Algorithms for Computational Linguistics

Kristina Striegnitz
Patrick Blackburn
Katrin Erk
Stephan Walter
Aljoscha Burchardt
Dimitra Tsovaltzi

MiLCA, SaarbrÃ¼cken

Abstract

The name Computational Linguistics already suggests that this displine comprises two related objects of research: natural language (NL) is studied and operational methods are developed. Both fields are investigated in their own right and divide into various topics. This course introduces a variety of NL phenomena together with appropriate implementations in the programming language Prolog. The topics dealt with are among others Morphology, Finite State Techniques, Syntax, Context Free Grammars, Parsing, and Semantics Construction.

MiLCA, Computerlinguistik, UniversitÃ?t des Saarlandes, SaarbrÃ¼cken, Germany
May 2003

This course is also available online:

`http://www.coli.uni-saarland.de/projects/milca/courses/coal/`

Contents

1 Finite State Automata **1**

1.1 Finite State Recognizers and Generators 1

1.1.1 A Simple Machine that can laugh 1

1.1.2 Finite State Automata 3

1.2 Some Examples 4

1.3 Deterministic vs. Non-deterministic FSAs 5

1.4 FSAs in Prolog 6

1.4.1 Representing FSAs in Prolog 6

1.4.2 A Recognizer and Generator for FSAs without Jump Arcs 8

1.4.3 A Recognizer and Generator for FSAs with Jump Arcs . 9

1.5 Finite State Methods in Computational Linguistics and their Limitations 9

1.6 The Prolog Files 10

1.7 Practical Session 10

1.8 Exercises and Solutions 11

1.8.1 Exercises 11

1.9 Further Reading 12

2 Finite State Parsers and Transducers **13**

2.1 Building Structure while Recognizing 13

2.1.1 Finite State Parsers 13

2.1.2 An Example 14

2.1.3 Separating out the Lexicon 15

2.2 Finite State Transducers 18

2.2.1 What are Finite State Transducers? 18

2.2.2 FSTs in Prolog 19

2.3 An Application: Morphology 21

2.3.1 Morphology 21

2.3.2 Morphological Parsing 22
2.3.3 From the Surface to the Intermediate Form 23
2.3.4 From the Intermediate Form to the Morphological Structure 24
2.3.5 Combining the two Transducers 26
2.3.6 Putting it in Prolog 26
2.3.7 Further Reading 27
2.4 The Complete Programs 27
2.5 Practical Session 27
2.6 Exercises 28

3 Regular Languages 29
3.1 Regular Languages and Relations 29
3.1.1 FSAs, Regular Expressions, and Regular Languages . . 29
3.1.2 Examples of Regular Expressions 29
3.1.3 Definition of Regular Expressions 30
3.1.4 Regular Expressions and FSAs 31
3.1.5 From Regular Expressions to FSAs 32
3.1.6 Regular Relations and Rewriting Rules 34
3.2 Properties of Regular Languages 35
3.2.1 Concatenation, Kleene Closure and Union 35
3.2.2 Intersection, Complementation 35
3.2.3 The ‘Pumping Lemma’ 37
3.3 Implementing Operations on FSAs 38
3.3.1 The `scan`-Predicate 38
3.3.2 Implementation of `scan/1` 39
3.3.3 `scan_state/2` 40
3.3.4 Recursion 41
3.3.5 Intersection 42
3.3.6 Union 44
3.4 The Complete Code 45
3.5 Practical Session 45
3.6 Exercises 46
3.7 Further Reading 48

4 Finite State Technologies for Dialogue Processing 49
4.1 Dialogue Processing 49
  4.1.1 Introduction 49
  4.1.2 General Dialogue characteristics 49
  4.1.3 General Dialogue characteristics: Examples 50
4.2 FSA-based dialogue systems 52
  4.2.1 Processing dialogue with finite state systems 52
  4.2.2 A simple FSA example 53
  4.2.3 Extending FSA 54
  4.2.4 Grounding extension 54
4.3 An in-depth look at the speaking elevator 58
  4.3.1 The SaarbrÃ¼cken CL department's Speaking Elevator 58
  4.3.2 How does the dialogue progress? 60
  4.3.3 An additional memory 60
  4.3.4 How long does the system wait for a response? 60
  4.3.5 How does the elevator deal with unrecognised utterances or inconsistent input? 61
  4.3.6 And what happens if the elevator does understand the input? 61
  4.3.7 How is the user input confirmed? 61
  4.3.8 During the trip... 61
  4.3.9 What happens when there is ambiguous input? 61
  4.3.10 Speaking elevator reference 62
4.4 Examples of required behaviour 62
  4.4.1 Turn-taking 62
  4.4.2 Adjacency pairs and insertions 62
  4.4.3 Grounding 62
  4.4.4 Dialogue context 63
  4.4.5 Ellipsis 63
  4.4.6 Reference resolution 63
  4.4.7 Mixed initiative 64
4.5 Dialogue characteristic and extension of the elevator 64
  4.5.1 Turn-taking 64
  4.5.2 Adjacency pairs and insertions 64
  4.5.3 Grounding 64

4.5.4 Dialogue context 65
4.5.5 Ellipsis 65
4.5.6 Reference resolution 65
4.5.7 Mixed initiative 65
4.6 Summary and Outlook 66
4.6.1 Advantages and disadvantages 66
4.6.2 The CSLU tool 67
4.6.3 Beyond finite state techniques 67
4.7 Exercises 68

5 Context Free Grammars 71
5.1 Context Free Grammars 71
5.1.1 The Basics 71
5.1.2 The Tree Admissibility Interpretation 73
5.1.3 A Little Grammar of English 73
5.2 DCGs 75
5.2.1 DCGs are a natural notation for context free grammars . 75
5.2.2 DCGs are really Syntactic Sugar for Difference Lists . . . 76
5.2.3 DCGs give us a Natural Notation for Features 76
5.3 DCGs for Long Distance Dependencies 78
5.3.1 Relative Clauses 78
5.3.2 A First DCG 80
5.3.3 A Second DCG 82
5.3.4 Gap Threading 84
5.3.5 Questions 86
5.3.6 Occurs-Check 87
5.4 Pros and Cons of DCGs 88
5.5 The Complete Code 90
5.6 Exercises 90

6 Parsing: Bottom-Up and Top-Down 91

6.1 Bottom-Up Parsing and Recognition 91
6.2 Bottom-Up Recognition in Prolog 92
6.3 An Example Run 96
6.4 How Well Have we Done? 97
  6.4.1 Implementation 97
  6.4.2 Algorithmic Problems 98
  6.4.3 [Sidetrack] Using a Chart 98
6.5 Top Down Parsing and Recognition 99
  6.5.1 The General Idea 99
  6.5.2 With Depth-First Search 99
  6.5.3 With Breadth-First Search 101
6.6 Top Down Recognition in Prolog 101
  6.6.1 The Code 101
  6.6.2 A Quick Evaluation 103
6.7 Top Down Parsing in Prolog 103
6.8 The Code 104
6.9 Practical Session 105
  6.9.1 Bottom Up 105
  6.9.2 Top Down 105
6.10 Exercises 105
  6.10.1 Bottom-Up 105
  6.10.2 Top-Down 106
  6.10.3 Midterm Project 107

7 Left-Corner Parsing 109

7.1 Introduction Left-Corner Parsing 109
  7.1.1 Going Wrong with Top-down Parsing 109
  7.1.2 Going Wrong with Bottom-up Parsing 110
  7.1.3 Combining Top-down and Bottom-up Information 111
7.2 A Left-Corner Recognizer in Prolog 113
7.3 Using Left-corner Tables 114
7.4 The Code 116
7.5 Practical Session 116
7.6 Exercises 116

8 Passive Chart Parsing **119**

8.1 Motivation 119

8.2 A Bottom-Up Recognizer Using a Passive Chart 121

8.2.1 An Example 121

8.2.2 Being Complete 123

8.3 Some Prolog Revision 123

8.3.1 Database manipulation 123

8.3.2 Failure Driven Loops 126

8.4 Passive Chart Recognition in Prolog 127

8.4.1 Representing the Chart 127

8.4.2 The Algorithm 128

8.4.3 An Example 130

8.5 The Code 131

8.6 Exercise 131

9 Active Chart Parsing **133**

9.1 Active Edges 133

9.2 The Fundamental Rule 134

9.3 Using an Agenda 136

9.4 A General Algorithm for Active Chart Parsing 136

9.5 Bottom-up Active Chart Parsing 137

9.5.1 An Example 137

9.5.2 An Example (continued) 138

9.5.3 Putting it into Prolog 144

9.6 The Code 148

9.7 Top-Down Active Chart Parsing 149

9.7.1 Top-down rule selection 149

9.7.2 Initializing Chart and Agenda 149

9.7.3 An Example 150

9.8 Exercise 155

10 Semantic Construction **157**

10.1 Introduction 157

10.2 First-Order Logic: Basic Concepts 158

10.2.1 Vocabularies 158

10.2.2 First-Order Languages 158

10.2.3 Building Formulas 159

10.2.4 Bound and Free Variables 160

10.2.5 Notation 161

10.2.6 Representing formulas in Prolog 162

10.3 Building Meaning Representations 163

10.3.1 Being Systematic 163

10.3.2 Being Systematic (II) 163

10.3.3 Three Tasks 164

10.3.4 From Syntax to Semantics 165

10.4 The Lambda Calculus 166

10.4.1 Lambda-Abstraction 166

10.4.2 Reducing Complex Expressions 167

10.4.3 Using Lambdas 168

10.4.4 Advanced Topics: Proper Names and Transitive Verbs . 170

10.4.5 The Moral 172

10.4.6 What's next 173

10.4.7 [Sidetrack:] Accidental Bindings 173

10.4.8 [Sidetrack:] Alpha-Conversion 174

10.5 Implementing Lambda Calculus 175

10.5.1 Representations 175

10.5.2 Extending the DCG 175

10.5.3 The Lexicon 176

10.5.4 A First Run 177

10.5.5 Beta-Conversion 177

10.5.6 Beta-Conversion Continued 178

10.5.7 Running the Program 179

10.6 Exercises 180

1

Finite State Automata

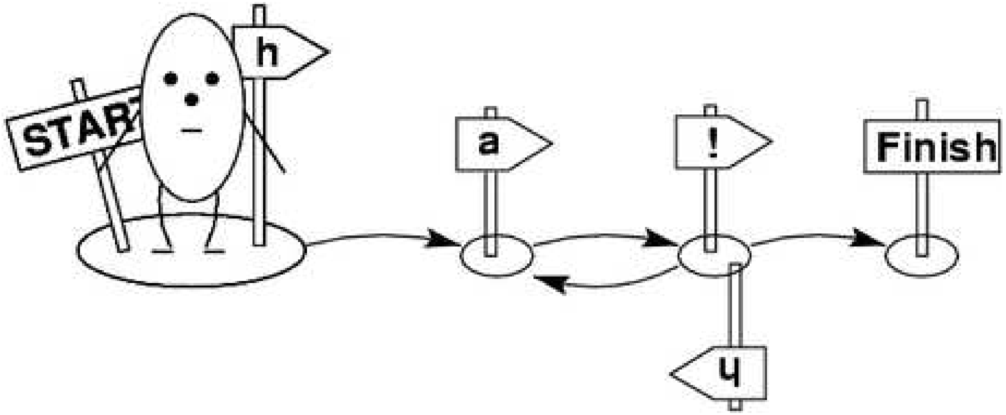
1.1 Finite State Recognizers and Generators

Finite state automata are used a lot for all kinds of things in computational linguistics. For example, they are used in morphology, phonology, text to speech, data mining ... Why are they so popular? Well, they are very simple (as you will see for yourself, soon) and extremely well understood mathematically. Furthermore, they are easy to implement (this, you will also see soon) and usually these implementations are very efficient. Therefore, finite state solutions tend to be good solutions. However, something so simple and efficient has to be restricted in what it is able to do in some way, which means that there isn't a finite state solution for every problem. We will look at some limitations of finite state methods later in this chapter. But first, let's see what finite state automata are and what they *can* do.

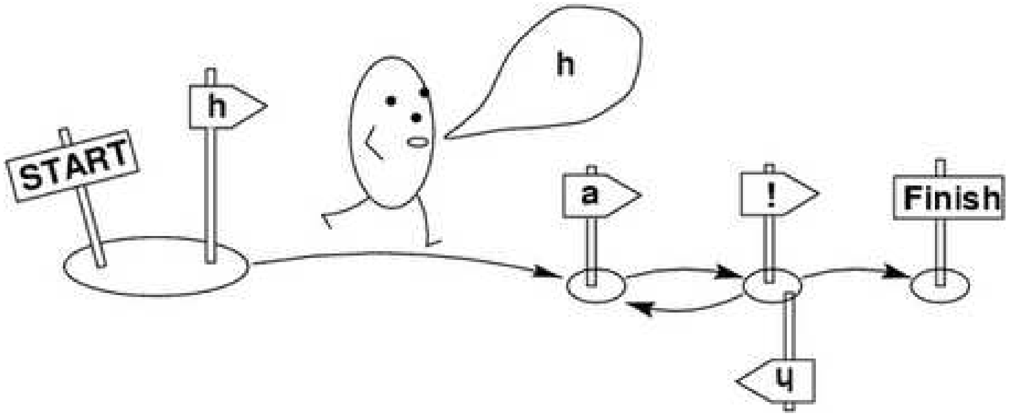
1.1.1 A Simple Machine that can laugh

A *finite state generator* is a simple computing machine that outputs a sequence of symbols.

It starts in some *start state*



and then tries to reach a final state by making *transition* s from one state to another. Every time it makes such a transition it *emits* (or *writes* or *generates*) a symbol.

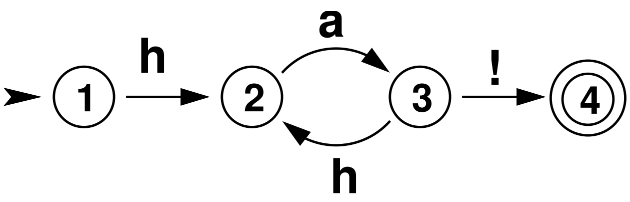


See movie in HTML version.

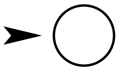
It has to keep doing this until it reaches a *final state* ; before that it cannot stop. All in all, finite state generators can only have a *finite* number of different states, that's where the name comes from. Another important property of finite state generators is that they only know the state they are currently in. That means they cannot look ahead at the states that come and also don't have any memory of the states they have been in before, or the symbols that they have emitted.

So, what does the generator in the pictures say? It laughs. It generates sequences of symbols of the form `ha!` or `haha!` or `hahaha!` or `hahahaha!` and so on. Why does it behave like that? Well, it first has to make a transition emitting `h`. The state that it reaches through this transition is not a final state. So, it has to keep on going emitting an `a`. Here, it has two possibilities: it can either follow the `!` arrow, emitting `!` and then stop in the final state (but remember, it can't look ahead to see that it would reach a final state with the `!` transition); or it can follow the `h` arrow emitting an `h` and going back to the state where it just came from.

Finite state generators can be thought of as directed graphs. And in fact finite state generators are usually drawn as directed graphs. Here is our laughing machine as we will from now on draw finite state generators:



The nodes of the graph are the states of the generator. We have numbered them, so that it is easier to talk about them. The arcs of the graph are the transitions, and the labels of the arcs are the symbols that the machine emits. A double circle indicates that this state is a final state. An arrow pointing to a state like this:



indicates that this state is a start state.

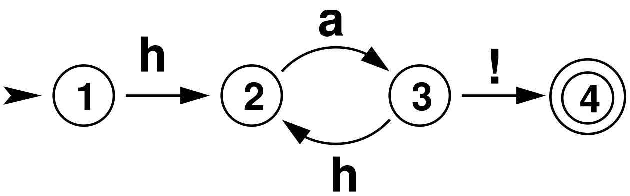
1.1.2 Finite State Automata

In the previous section, we have learned that *finite state generators* are simple computing machines that *output a sequence of symbols*. You will often find the notion of an *output tape* the symbols are written on. A finite state *recognizer* in turn is a simple computing machine that *reads* (or at least tries to read) a sequence of symbols from an *input tape* . That is only a small difference: Finite state generators and finite state recognizers are exactly the same kind of machine. Just that we are using them to output symbols in one case and to read symbols in the other case. The general term for such machines is *finite state automaton* (*FSA*) or *finite state machine* (*FSM*). But let's have a closer look at what it means for a finite state automaton to *recognize* a string of symbols.

An FSA recognizes (or accepts) a string of symbols (or word) $s_1, s_2, \ldots, s_n$ if starting in an intial state, it can read in the symbols one after the other while making transitions from one state to another such that the transition reading in the last symbol takes the machine into a final state. That means an FSA fails to recognize a string if:

1. it cannot reach a final state; or
2. it can reach a final state, but there are still unread symbols left over when it does

So, this machine



recognizes laughter. For example, it accepts the word `ha!` by going from state 1 via state 2 and state 3 to state 4. At that point it has read all of the input and is in a final state. It also accepts the word `haha!` by making the following sequence of transitions: state 1, state 2, state 3, state 2, state 3, state 4. Similarly, it accepts `hahaha!` and `hahahaha!` and so on. However, it doesn't accept the word `haha` – although it will be able to read the whole input (state 1, state 2, state 3, state 2, state 3). The problem is that it will end up in a non-final state without anything left to read that could take it into the final state. Also, it does not accept `hoho!`. The reason is that with this input, it won't be able to read the whole input (there is no transition that allows reading an `o`).

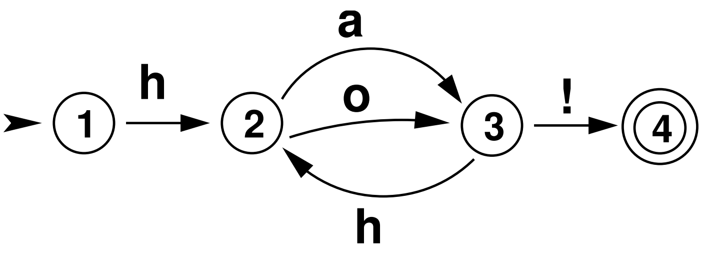
So, when used in recognition mode, this machine recognizes exactly the same words that it generates when used in generation mode. This is something which is true for all finite state automata and we can make it more precise:

- A *formal language* is a set of strings.
- The *language accepted* (or recognized) by an FSA is the set of all strings it recognizes when used in recognition mode.
- The *language generated* by an FSA is the set of all strings it can generate when used in generation mode.
- The language accepted and the language generated by an FSA are exactly the same.

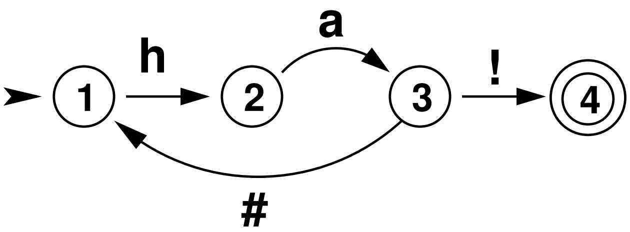
1.2 Some Examples

Let’s have a look at some more examples of what finite state automata can look like.

Assume that we want to extend our laughing machine so that it not only recognizes sequences of `ha` (followed by `!`) as laughter but also sequences of `ho` (followed by `!`) and sequences mixing `ha`s and `ho`s (followed by `!`). So, `a`s may now be replaced by `o`s. This means that in all states where the machine can make an `a` transition, it should now also be able to make an `o` transition. So, all we have to do to extend our machine in this way is to add an `o` transition from state 2 to state 3.

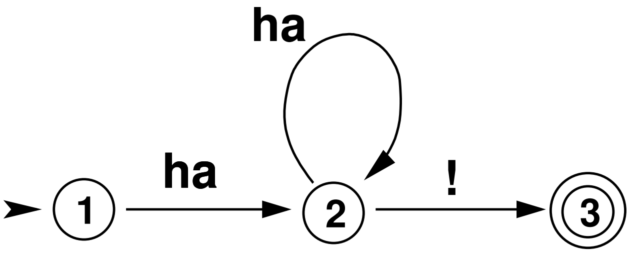


Now, look at the following FSA:



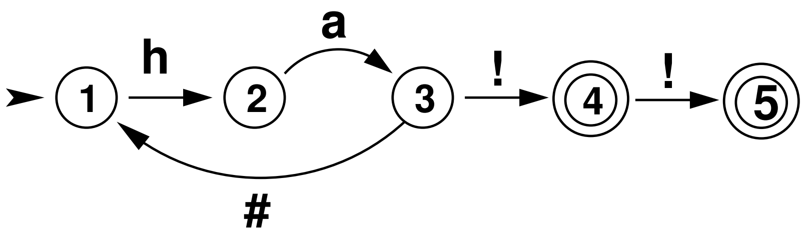
It has a strange transition from state 3 to state 1, which is reads/emits `#`. We will call transitions of this type *jump arc* s. Jump arcs let us jump from one state to another without emitting or reading a symbol. So, `#` is really just there to indicate that this is a jump arc and the machine does not read or write anything when making this transition. This FSA accepts/generates the same language as our first laughing machine, namely sequences of `ha` followed by a `!`.

And what language does this FSA accept/generate?



It also accepts/generates the same language as our first laughing machine. But the *alphabet* it uses is a bit different. Here `ha` counts as *one* symbol and can therefore be read by *one* transition. Furthermore, the FSA has a reflexive transition, going from state 2 back to itself.

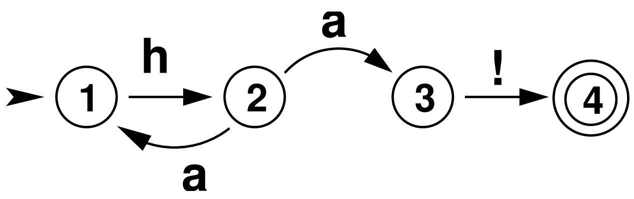
Finally, an FSA can have several intial and final states (it must have at least one initial and one final state, though). Here is a machine that has two final states. What language does it recognize?



The machine can stop after having generated one exclamation mark or after having generated two exclamation marks. So, the words of the language it accepts/generates all consist of a sequence of `ha` followed by either one `!` or two `!`.

1.3 Deterministic vs. Non-deterministic FSAs

Look at the following FSA:



It accepts/generates the same language as our first laughing machine, but it differs from the FSAs we have seen so far in a very important way. It has two arcs labelled with the same symbol (`a`) going out of one state (state 2) – this FSA is *non-deterministic* .

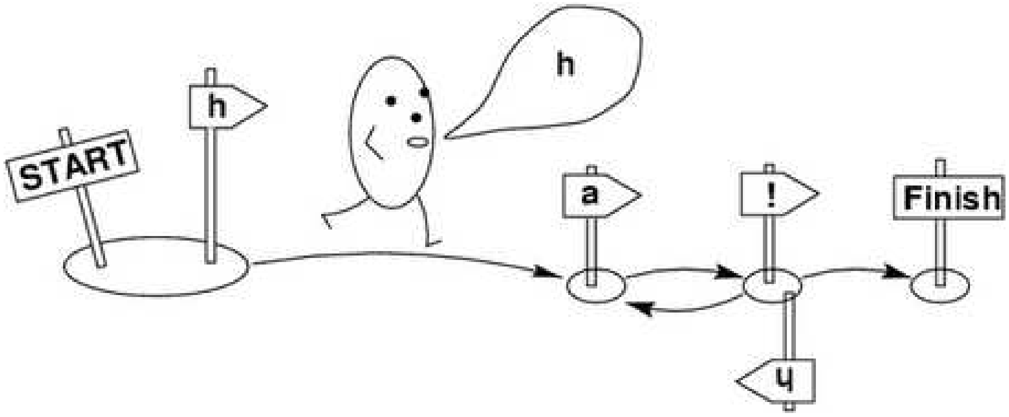
Mathematically speaking, non-determinism doesn't add anything to what FSAs can do. In fact, we can always find a deterministic automaton that recognizes/generates exactly the same language as a non-deterministic automaton. Also when using an automaton for generation, there is no big difference between deterministic and non-deterministic machines. You can think of the machine flipping a coin to decide which transitition to take whenever there is more than one arc going out of a state. Operationally however, there is a big difference when we want to use the machine for recognition. When the next symbol on the input tape is an `a`, the machine has to decide which transition to take and if it makes the wrong decision, it may fail to recognize the input string, even though it would have been able to, had it decided differently. That means that non-determinism brings with it the need to perform *search* .

Here is an example. Assume that we want to check whether the FSA above recognizes the word `ha!`. Starting from state 1 we have no choice but to go to state 2 via the `h` transition. But next, we have to read an `a` and there are two `a`-transitions going out of state 2. Which one should we take? Well, you can see of course that we would want to take the transition leading to state 3, because only when taking this transition will we be able to read the whole input and end in a final state. However, remember that the machine cannot look ahead and can therefore not base its decision on the rest of the input or the following states in the automaton. All it can do at this point is to arbitrarily decide for one transition, try it, and go back to try another transition in case this one turns out to not lead us to acceptance of the string. That means, we have to systematically try all alternatives and to do so, we have to keep track of what we have tried already.

1.4 FSAs in Prolog

1.4.1 Representing FSAs in Prolog

In our implementation, we are going to treat FSAs as passive data structures that are manipulated by other programs. In a way, the very first pictures of FSAs in this chapter showed a similar division into a *declarative* part and a *procedural* part: the network-like graph is a declarative representation of the FSA and the “egg” is the program. Depending on what instructions we give to the “egg”, we can use it to generate words (as in the picture) or to recognize words using the same FSA.



Separating declarative and procedural information is often a good idea. In many cases, it makes it easier to understand what is going on in a program, to maintain a program, and to reuse information. In the chapter on Context Free Grammars (Chapter 5), we will learn more about why the machine oriented view is unattractive.

We will use three predicates to represent FSAs:

- `start/2`
- `final/2`
- `trans/4`

The first argument in each of these predicates is simply an atom naming the automaton to be specified. Let’s start by specifying our first laughing machine, and call it `a1`. It will have the following Prolog representation:

```
start(a1,1).

final(a1,4).

trans(a1,1,2,h).

trans(a1,2,3,a).

trans(a1,3,4,!).
trans(a1,3,2,h).
```

`start(a1,1)`, for instance, says that `1` is the initial state of `a1`; and `final(a1,4)` says that `4` is its final state (to be precise: one final of its states. But incidentally, `a1` has only one final state). And `trans(a1,1,2,h)` says that there is a `h`-transition from state `1` to state `2` in `a1`.

We will, furthermore, use the atom `'#'` to mark jump arcs. Here is the laughing machine with the jump arc. We shall give it the name `a2`:

```
start(a2,1).

final(a2,4).

trans(a2,1,2,h).

trans(a2,2,3,a).

trans(a2,3,4,!).

trans(a2,3,1,'#').
```

And here's the non-deterministic version (under the name `a3`):

```
start(a3,1).

final(a3,4).

trans(a3,1,2,h).

trans(a3,2,3,a).

trans(a3,3,4,!).

trans(a3,2,1,a).
```

As you can see, the Prolog representation of these finite state machines is a straightforward translation of the graphs that we have been drawing in the previous sections.

1.4.2 A Recognizer and Generator for FSAs without Jump Arcs

Now that we know how to represent FSAs, we would of course like to do something with them; we want to use them to generate and recognize strings. That is, we need programs to work on top of FSA representations. Those programs should be general enough, so that we don't have to know anything about the structure of a certain FSA before using it – in particular, we would like these programs to be able to deal with deterministic as well as non-deterministic FSAs. Prolog helps us a lot with this point, because it's built-in backtracking mechanism provides us with the search tool that we need to deal with the non-determinism. Furthermore, Prolog is so declarative, that one and the same program can (up to a point) work as both a recognizer and a generator.

Let's first ignore the fact that there may be jump arcs and write a recognizer/generator for FSAs without jump arcs. We will define the predicate `recognize/3` which takes the name of the automaton to be used as first argument, the number of the node you want to start from as second argument and a list of symbols representing the string that you want to recognize as third argument:

```
recognize(A,Node,SymbolList) :- ...
```

For instance the query `recognize(a1,1,[h,a,h,a,!])` should succeed, if the list of symbols `[h,a,h,a,!]` can be recognized by the FSA called `a1` in Prolog's database starting from node `1` and ending in a final state.

We will define `recognize/3` as a recursive predicate, which first tries to find a transition in the automaton `A` from state `Node` to some other state reading the first symbol of the list `SymbolList` and then calls itself with the node it can reach through this transition and the tail of `SymbolList`. The code for the predicate is found in See file `recognize.pl`..

In the base case, `recognize/3` is called with an empty list, i.e. the whole input has been read. In this case, it succeeds if the `Node` is a final state in `A`:

```
recognize(A,Node,[]) :-
    final(A,Node).
```

In the case where `SymbolList` is not empty, we first retrieve a transition of `A` that starts from `Node`. Then we take this transition, thereby reading a symbol of the input, and recursively call `recognize/3` again.

```
recognize(A,Node_1,String) :-
    trans(A,Node_1,Node_2,Label),
    traverse(Label,String,NewString),
    recognize(A,Node_2,NewString).
```

The predicate `traverse/3` checks that we can indeed take this transition, i.e. that the label of the transition is the same as the first symbol of the input list, and returns the input without the symbol that we have just read.

```
traverse(Label,[Label|Symbols],Symbols).
```

Now, if Prolog should ever retrieve an arc from the database that later turns out to be a bad choice because it doesn't lead to success, it will (automatically) backtrack on the call to `trans/4` in the second clause of `recognize/3` and look for alternatives.

As promised, we can use `recognize/3` in two different modes. In recognition mode, we want to give a list of symbols `SymbolList` and want to find out whether there is an initial node `Node` in `A` such that the query `recognize(A,Node,SymbolList)` returns `yes`. Here is a driver predicate for the recognition mode:

```
test(A,Symbols) :-
    start(A,Node),
    recognize(A,Node,Symbols).
```

In generation mode, we want to get all lists of symbols which `recognize/3` can generate using `A` and starting from its initial node. For this, we can just call `test1/2` with an uninstantiated variable as second argument. `test1/2` then selects an initial node and calls `recognize/3` with this node as first argument and an uninstantiated second argument.

```
generate(A,X) :-
   test(A,X).
```

1.4.3 A Recognizer and Generator for FSAs with Jump Arcs

It is very easy to adapt the recognizer/generator of the previous section to be able to deal with jump arcs. All we have to do is to specify that if an arc is labelled with `'#'`, we can make a transition without doing anything to the input string. We do so by adding the following the following clause to the `traverse/3` predicate:

```
traverse('#',String,String).
```

1.5 Finite State Methods in Computational Linguistics and their Limitations

We have seen in this chapter that finite state machines are very simple. As a consequence, there are limitations to what they can do. It is, for example, not possible to write an FSA that generates the language a^nb^n, i.e. the set of all strings which consist of a (possibly empty) block of `a`s followed by a (possibly empty) block of `b`s of exactly the same length. Mathematically speaking, FSAs have certain expressive weaknesses. This also limits their expressive adequacy from a linguistic point of view, because many linguistic phenomena can only be described by languages which cannot be generated by FSAs. In fact, even the language a^nb^n is linguistically relevant, since many linguistic constructions require ‘balanced’ structures. We shall have a closer look at the expressive strength (or weakness) of FSAa in Chapter 3.

However, there are linguistic applications where the expressive power of finite state methods is just sufficient, and FSAs have been used and are used a lot for all kinds of tasks in computational linguistics; the flip side of their expressive weakness being that they usually behave very well computationally. If you can find a solution based on finite state methods, your implementation will probably be efficient.

Areas where finite state methods have been shown to be particularly useful are phonological and morphological processing. We will see some simple examples for that in the next chapter (Chapter 2). But finite state methods have also been applied to syntactic analysis. Although they are not expressive enough if a full syntactic analysis is required, there are many applications where a partial syntacic analysis of the input is sufficient. And such partial analyses can be constructed with cascades of finite state automata (or rather transducers, which we will learn about in the next chapter), where one machine is applied to the output of another. Furthermore, hidden markov models, which are very common for speech recognition and part of speech tagging, can be seen as a variant of FSAs which assigns probabiltites to transitions.

1.6 The Prolog Files

See file `haha.pl`.	Three 'laughing-machines'
See file `recognize.pl`.	The recognizer/generator for FSAs (with *or* without jump arcs).
See file `recognize1.pl`.	A version of the recognizer/generator that only works for FSAs without ju

1.7 Practical Session

We will start out with a couple of simple keyboard exercises so make sure that you understand how the Prolog programs that we saw in this section work.

Note: The file See file `recognize1.pl`. contains a version of our recognizer/generator that that only works for FSAs without jump arcs. It defines the same predicates as See file `recognize.pl`., but all predicate names end in a `1`.

1. Start Prolog and consult `recognize.pl`, `recognize1.pl`, and `haha.pl`. Using both `test1` (which does not deal with jump arcs) and `test` (which does) see whether various strings are accepted e.g `test1(a1,[h,a,!]).` `test(a1,[h,a,!]).` `test1(a1,[h,a,h,a,h,a,!]).`

   Make sure you understand why these strings are accepted. That is, use trace to step through the execution so that you understand exactly what is going on.

2. Now let's have a look at the automaton `a3`. The difference between `a1` and `a3` is that `haha5` is not deterministic. Try giving the following queries:

   `test(a3,[h,a,!]).` `test1(a3,[h,a,!]).` `test(a3,[h,a,h,a,h,a,!]).`

   Make sure you understand why these strings are accepted. That is, use trace to step through the execution so that you understand exactly what is going on.

3. We now turn to `a2`. The big difference between `a1` and `a2` is that `a2` contains a jump arc `#`. Now, `recognize1` does not handle jump arcs so it should not be able to handle `a2` correctly.

   Try out the following experiments, and make sure you understand why recognize1 gives the responses it gives: `test1(a2,[h,a,!]).` `test1(a2,[h,a,h,a,!]).` `test1(a2,[h,a,h,a,h,a,!]).`

Now try these examples with `test` instead of `test1`. Again, carry out traces.

4. Finally, make sure that you understand how the generate predicates are defined. Try using `generate1` to generate using `a2`. Now try using `generate` with `a2`. Why does `generate1` give a wrong response?

1.8 Exercises and Solutions

1.8.1 Exercises

Exercise 1.1 *Now, you are able to write your own automata. You can then use the* `recognize` *and* `generate` *predicates to test them. So, here is a task which is a bit more interesting: Write an FSA (*First*, as a graph and,* then*, in Prolog notation!) that recognizes English noun phrases constructed from the following words:* a, the, witch, wizard, Harry, Ron, Hermione, broomstick, brave, fast, with, rat. *E.g.*

- *the brave wizard*
- *the fast broomstick*
- *the wizard with the rat*
- *Ron*
- ...

Exercise 1.2 *In the lecture, we represented* `a1` *in Prolog as follows:*

```
start(a1,1).

final(a1,4).

trans(a1,1,2,h).
trans(a1,2,3,a).
trans(a1,3,4,!).
trans(a1,3,2,h).
```

But we could also have represented it like this:

```
start(a1,1).

final(a1,4).

trans(a1,1,2,h).
trans(a1,2,3,a).
trans(a1,3,2,h).
trans(a1,3,4,!).
```

Does it make any difference which way we represent it?

Exercise 1.3 *Here's the code for* `recognize1` *given in the lecture:*

```
recognize(A,Node,[]) :-
    final(A,Node).

recognize(A,Node_1,String) :-
    trans(A,Node_1,Node_2,Label),
    traverse(Label,String,NewString),
    recognize(A,Node_2,NewString).
```

Suppose we changed it to this:

```
recognize(A,Node,[]) :-
    final(A,Node),!.

recognize(A,Node_1,String) :-
    trans(A,Node_1,Node_2,Label),
    traverse(Label,String,NewString),!,
    recognize(A,Node_2,NewString).
```

What effect would this change have? What sort of `FSA`*s would not be affected by the change?*

Exercise 1.4 *Write* `FSA`*s that generate the following languages (where e.g. by* a^m *we mean the result of repeating a m-times):*

1. $a^m b^n$, *where* $m > 3, n > 2$
2. $a^m c^l b^n$, *where* $m > 1, l = 2, n \geq 3$

1.9 Further Reading

- If you want to learn more about the formal properties of finite state automata [4] and [7] provide (rather mathematical) introductions.
- [5] (chapter 2) give a less formal introduction to the mathematical properties of FSAs.
- In [5] you can also find more about applications of finite state automata in computational linguistics.

2

Finite State Parsers and Transducers

2.1 Building Structure while Recognizing

2.1.1 Finite State Parsers

So, in the case of a finite state parser the output should tell us which transitions had to be made in the underlying FSA when the input was recognized. That is, the output should be a sequence of nodes and arcs. For example, if we give the input `[h,a,h,a,!]` to a parser for our first laughing automaton (`a1`), it should return `[1,h,2,a,3,h,2,a,3,!,4]`.

There is a fairly standard technique in Prolog for turning a recognizer into a parser: add one or more extra arguments to keep track of the structure that was found. We will now use this technique to turn `recognize/3` of the last chapter into `parse/4`, i.e. an FSA-based parser.

In the base clause, when the input is read and the FSA is in a final state, all we have to do is record that final state. So, we turn

```
recognize(A,Node,[]) :-
    final(A,Node).
```

into

```
parse(A,Node,[],[Node]) :-
    final(A,Node).
```

Then let's look at the recursive clause. The recursive clause of `recognize/3` looked as follows:

```
recognize(A,Node_1,String) :-
    trans(A,Node_1,Node_2,Label),
    traverse(Label,String,NewString),
    recognize(A,Node_2,NewString).
```

And here is the recursive clause of `parse/4`:

```
parse(A,Node_1,String,[Node_1,Label|Path]) :-
    trans(A,Node_1,Node_2,Label),
    traverse(Label,String,NewString),
    parse(A,Node_2,NewString,Path).
```

The parser records the state the FSA is in and the symbol it is reading on the transition it is taking from this state. The rest of the path, i.e. the sequence of states and arcs that the FSA will take from `Node2` onwards, will be specified in the recursive call of `parse/4` and collected in the variable `Path`.

The only thing that's left to do is to adapt the driver predicates `testparse/2` and `generate/2`. The new driver predicates look as follows:

```
testparse(A,Symbols,Parse) :-
    start(A,Node),
    parse(A,Node,Symbols,Parse).


generate(A,Symbols,Parse) :-
   testparse(A,Symbols,Parse).
```

You can find the new predicates in See file `parse.pl`..

2.1.2 An Example

Now, let's step through an example to have a look at how the output is being built in the extra argument during recognition. Assume that we have loaded the Prolog representation of `a1` (our first laughing automaton) into the Prolog database. So the database contains the following facts:

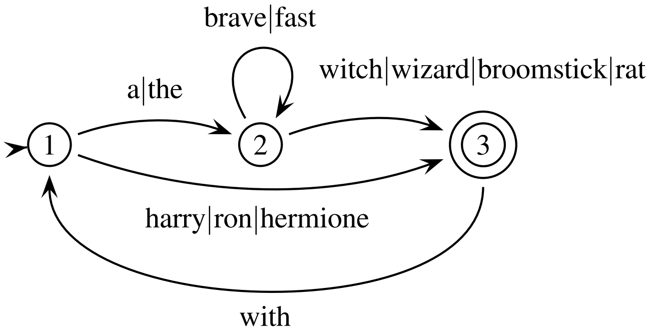
```
start(a1,1).

final(a1,4).

trans(a1,1,2,h).

trans(a1,2,3,a).

trans(a1,3,4,!).

trans(a1,3,2,h).
```

See file `parse.pl`.

We ask Prolog the following query: `testparse(a1,[h,a,!],Parse).` Prolog retrieves `1` as the only initial node in the FSA `a1` and calls `parse` instantiated as `parse(a1,1,[h,a,!],Parse` Next, Prolog has to retrieve arcs starting in node 1 from the database. It finds `trans(a1,1,2,h)`, which it can use because the first symbol in the input is `h` as well. So, `Parse` is unified with `[1,h|G67]` where `G67` is some Prolog internal variable. Prolog then makes a recursive call (the first recursive call) of `parse` with `parse(a1,2,[a,!],G67).` Now, Prolog finds `trans(a1,2,3,a)` in the database. So, `G67` gets unified with `[2,a|G68]` (`G68` again being some internal variable) and Prolog makes the second recursive call of `parse`: `parse(a1,3,[!],G68).` Using `trans(3,4,!)`, the last symbol of the input can be read and `G68` gets instantiated to `[3,!|G69]`. The next recursive call of `parse` (`parse(a1,4,[],G69)`) matches the base clause. Here, `G69` gets instantiated to `[4]`, instantiating `G68` to `[3,!,4]`, `G67` to `[2,a,3,!,4]`, and `Parse` to `[1,h,2,a,3,!,4]` as Prolog comes back out of the recursion. If you have trouble understanding how the output gets assembled, draw a search tree for the query `parse(a1,1,[h,a,!],Parse)`. Note, how with every recursive call of `parse` the third argument gets instantiated with a list. The first two elements of this list are the state the FSA is currently in and the next symbol it reads; the rest of the list is an uninstantiated variable at first, but gets further instantiated by the next recursive call of `parse`.

2.1.3 Separating out the Lexicon

In !!!UNEXPECTED PTR TO EX_FSA.EX1!!! you were asked to construct a finite state automaton recognizing those English noun phrases that can be built from the words `the`, `a`, `wizard`, `witch`, `broomstick`, `hermione`, `harry`, `ron`, `with`, `fast`. The FSA that you came up with probably looked similar to this:



Let's call this automaton `b1`. So it is

```
start(b1,1).


final(b1,3).


trans(b1,1,2,a).


trans(b1,1,2,the).
trans(b1,2,2,brave).

trans(b1,2,2,fast).

trans(b1,2,3,witch).

trans(b1,2,3,wizard).

trans(b1,2,3,broomstick).

trans(b1,2,3,rat).

trans(b1,1,3,harry).

trans(b1,1,3,ron).

trans(b1,1,3,hermione).

trans(b1,3,1,with).
```

in Prolog (this automaton and the next are specified in See file `harry.pl`.).

Now, what would Prolog answer if we used the parser of the previous section on this automaton? Let's parse the input `[the,fast,wizard]`: `testparse(b1,[the,fast,wizard],Parse).`

The call instantiates `Parse=[1,the,2,fast,2,wizard,3]`. This tells us how the FSA was traversed for recognizing that this input is indeed a noun phrase. But wouldn't it be even nicer if we got a more abstract explanation? E.g. one saying that `[the,fast,wizard]` is a noun phrase because it consists of a determiner, followed by an adjective, which is in turn followed by a common noun. That is, we would like the parser to return something like this:

```
[1,det,2,adj,2,noun,3].
```

Actually, you were probably already making a similar abstraction when you were thinking about how to construct that FSA. You were probably thinking: 'Well, a noun phrase starts with a determiner, which can be followed by zero or more adjectives, and it must end in a noun; 'the' and 'a' are the determiners that I have, so I need a 'the' and an 'a' transition from state 1 to state 2.' In fact, it would be a lot nicer if we could specify transitions in the FSA based on syntactic categories like determiner, common noun, and so on, and additionally give a separate lexicon specifying what words belong to each category. Like this, for example:

```
start(b2,1).

final(b2,3).
```

```
trans(b2,1,2,det).

trans(b2,2,2,adj).

trans(b2,2,3,cn).

trans(b2,1,3,pn).

trans(b2,3,1,prep).

lex(a,det).
lex(the,det).
lex(fast,adj).
lex(brave,adj).
lex(witch,cn).
lex(wizard,cn).
lex(broomstick,cn).
lex(rat,cn).
lex(harry,pn).
lex(hermione,pn).
lex(ron,pn).
```

It's not very difficult to change our recognizer to work with FSA specifications that, like the above, define their transitions in terms of categories instead of symbols and then use a lexicon to map those categories to symbols or the other way round. The only thing that changes is the definition of the `traverse` predicate. We don't simply compare the label of the transition with the next symbol of the input anymore, but have to access the lexicon to check whether the next symbol of the input is a word of the category specified by the label of the transition. That means, instead of

```
traverse('#',String,String).
traverse(Label,[Label|Symbols],Symbols).
```

we use

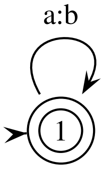
```
traverse('#',String,String).
traverse(Label,[Symbol|Symbols],Symbols) :-
   lex(Symbol,Label).
```

A recognizer for FSAs with categories (using the above version of `traverse/3` can be found in See file `cat_parse.pl`..

2.2 Finite State Transducers

2.2.1 What are Finite State Transducers?

A finite state transducer essentially is a finite state automaton that works on two (or more) *tape* s. The most common way to think about transducers is as a kind of "translating machine". They read from one of the tapes and write onto the other. This, for instance, is a transducer that translates `a`s into `b`s:



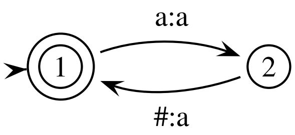
`a:b` at the arc means that in this transition the transducer reads `a` from the first tape and writes `b` onto the second.

Transducers can, however, be used in other modes than the translation mode as well: in the generation mode transducers write on both tapes and in the recognition mode they read from both tapes. Furthermore, the direction of translation can be turned around: i.e. `a:b` can not only be read as "read `a` from the first tape and write `b` onto the second tape", but also as "read `b` from the second tape and write `a` onto the first tape".

So, the above transducer behaves as follows in the different modes.

- generation mode: It writes a string of `a`s on one tape and a string `b`s on the other tape. Both strings have the same length.
- recognition mode: It accepts when the word on the first tape consists of exactly as many `a`s as the word on the second tape consists of `b`s.
- translation mode (left to right): It reads `a`s from the first tape and writes a `b` for every `a` that it reads onto the second tape. Try it: `testtrans(a2b,[a,a,a],Tape2).`
- translation mode (right to left): It reads `b`s from the second tape and writes an `a` for every `b` that it reads onto the first tape.

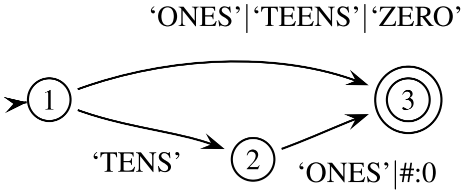
Transitions in transducers can make jumps going from one state to another without doing anything on either one or on both of the tapes. So, transitions of the form `a:#` or `#:a` or `#:#` are possible. Here is an example:



And what does this transducer do?

- generation mode: It writes twice as many `a`s onto the second tape as onto the first one.
- recognition mode: It accepts when the second tape has twice as many `a`s as the first one.
- translation mode (left to right): It reads `a`s from the first tape and writes twice as many onto the second tape.
- translation mode (right to left): It reads `a`s from the second tape and writes half as many onto the first one. Try it: `testtrans(adoubler,Tape1,[a,a,a,a,a,a,a,a]).`

Similar as with FSAs, we can also use categories to label the arcs and provide a kind of lexicon which translates these categories into real labels, i.e. labels of the form `X:Y`. Here is an example translating English number terms into numbers.



And here is the lexicon that maps the category labels to standard FST transition labels:

```
lex(one:1,'ONES').
lex(two:2,'ONES').
lex(three:3,'ONES').
lex(four:4,'ONES').
...
lex(eleven:11,'TEENS').
lex(twelve:12,'TEENS').
...
lex(twenty:2,'TENS').
lex(twenty:3,'TENS').
...
lex(zero:0,'ZERO').
```

An implementation of a transducer using such a lexicon can be found in See file `trans_lex.pl`.. We will not discuss it here in detail. If you have read the next section, you will easily understand it yourself.

2.2.2 FSTs in Prolog

In implementing finite state transducers in Prolog, we will follow the same strategy that we used for FSAs: we represent an FST as a static data structure that other programs manipulate.

Here is how we represent our first transducer, the `a` to `b` translator (found in See file `a2b.pl`.).

```
:- op(250,xfx,:).

start(a2b,1).

final(a2b,1).

trans(a2b,1,1,a:b).
```

To be able to write `a:b` as the label of the arc, we have to define `:` as an infix operator as is done by the operator definition.

Our second transducer, the `a` doubler (See file `adoubler.pl`.), looks like this in Prolog representation:

```
:- op(250,xfx,:).

start(adoubler,1).

final(adoubler,1).

trans(adoubler,1,2,a:a).

trans(adoubler,2,1,'#':a).
```

Now, we need a program that can manipulate these data structures and carry out the transduction. We will extend `recognize1` from Section 2.1.1 to work as a transducer (see See file `trans.pl`.).

The base case is that both tapes are empty and the FSA is in a final state. In Prolog:

```
transduce(A,Node,[],[]) :-
    final(Node).
```

In the recursive case, we make a transition from one node to another which is licensed by some `trans` definition in the database. As in the last chapter we define a predicate `traverse/6` to check that the transition is indeed licensed.

```
transduce(A,Node1,Tape1,Tape2) :-
    trans(Node1,Node2,Label),
    traverse(A,Label,Tape1,NewTape1,Tape2,NewTape2),
    transduce(A,Node2,NewTape1,NewTape2).

traverse(A,L1:L2,[L1|RestTape1],RestTape1,[L2|RestTape2],RestTape2).
```

Finally, we define the following driver predicate `testtrans/3`. It can be called with both arguements instantiated, only one of them instantiated, or both uninstantiated – depending on which mode we want to use the transducer in.

```
testtrans(A,Tape1,Tape2) :-
    start(Node),
    transduce(A,Node,Tape1,Tape2).
```

We can use this program to transduce `a`s to `b`s with our first transducer. To be able to use the second transducer, the `a` doubler, as well, we need a program that can handle transitions involving jumps. What do we have to change for that? Well, the only thing that changes is the way that the tapes are treated when making a transition. This is taken care of by the `traverse` predicate and this is all we have to adapt. (Remember that when extending the recognizer of the last chapter to handle jump arcs, we also only changed the `traverse` predicate.)

So, what are the possibilites how a tape can be affected by a transition? There are four:

- We have to jump on both tapes.
- We have to jump on the first but not on the second tape.
- We have to jump on the second but not on the first tape.
- We have to jump on neither tape (this is what the clause of `traverse/6` given above does).

The Prolog definition of `traverse` therefore has four clauses:

```
traverse('#':'#',Tape1,Tape1,Tape2,Tape2).
traverse('#':L2,Tape1,Tape1,[L2|RestTape2],RestTape2).
traverse(L1:'#',[L1|RestTape1],RestTape1,Tape2,Tape2).
traverse(L1:L2,[L1|RestTape1],RestTape1,[L2|RestTape2],RestTape2).
```

2.3 An Application: Morphology

2.3.1 Morphology

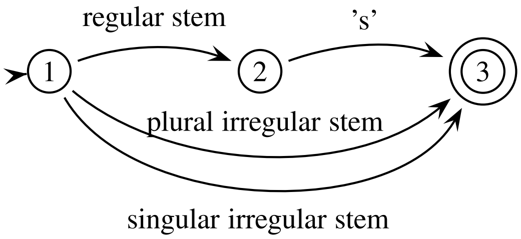
Morphology is about the inner structure of words. It is interested in what are the smallest units in word that bear some meaning and how can they be combined to form words.

First, what are meaning bearing units? You can say that the word *rabbits* has to two units which contribute to the meaning of the word: *rabbit* contributes the main meaning, and *s* adds the information that the word is plural. The smallest unit in a word that bear some meaning, such as *rabbit* and *s*, are called *morpheme* s. Morphemes like *rabbit* that contribute the main meaning of a noun, verb, etc. are also called *stem* s, while the other morphemes are also known as *affix* es.

The second question is then, how can morphemes be combined to form words that are legal in some language. Two kinds of processes can be distinguished here. They are called *inflection* and *derivation (morphological)* . Inflection is usually taken to be the process of adding a grammatical affix to a word stem, forming a word of the same class as the stem. Adding plural *s* to a noun stem, for example, is an inflectional process. Derivation is when adding an affix to a word stem results in a word with a class different from that of the stem. Making a noun out of a verb by adding *ation* to the verb is an example: *realize* + *ation* gives us *realization*.

Let's look at the example of inflection of nouns in English in more detail. Basically, English nouns only come in two forms: singular and plural. In the standard case, you get the plural form by adding an *s* to the end of the noun stem, which, at the same time, is the singular form of the noun: *rabbit* vs. *rabbits*. However, there are some exceptions, where the plural is not built by simply adding an *s* to the stem, but rather by changing the stem: *foot* vs. *feet*. So, valid English nouns consist of either the stem of a regular noun, or the singular stem of an irregular noun, or the plural stem of an irregular noun, or the stem of a regular noun plus an s.

The nice thing about these morphological rules, is that they can be captured using finite state techniques. Which means that you can draw a finite state automaton describing the inflectional morphology of English noun phrases. Here it is:



And this is even true for languages which have a much richer morphology than English, Finnish for example.

There is one more complication, however, and that is that sometimes, when two morphemes are combined, additional changes happen at the boundary. When combining the noun stem *fox*, for instance, with the plural morpheme *s* we get *foxes* instead of *foxs*. That is, an *e* slips in between the stem and the suffix. The same things happens with stem *kiss*, which becomes *kisses* when adding the plural morpheme. This inserting of *e*s at the morpheme boundary is not arbitrary though. It follows a rule, which says: "Insert an *e* when a morpheme ending in *s, x* or *z* is combined with the suffix *s*." As we shall see in the next section this kind of rules can also be expressed using finite state technology.

2.3.2 Morphological Parsing

The goal of *morphological parsing* is to find out what morphemes a given word is built from. For example, a morphological parser should be able to tell us that the word *cats* is the plural form of the noun stem *cat*, and that the word *mice* is the plural form of the noun stem *mouse*. So, given the string *cats* as input, a morphological parser should produce an output that looks similar to *cat N PL*. Here are some more examples:

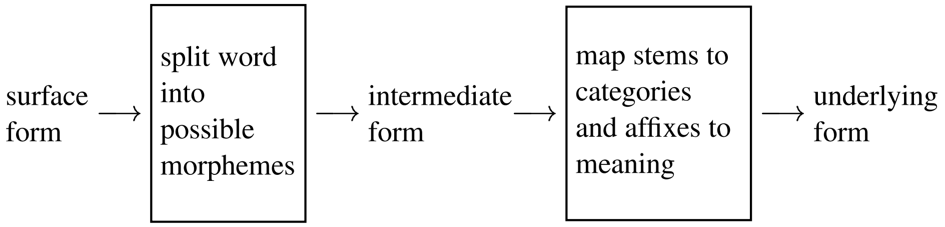
mouse	==>	*mouse N SG*
mice	==>	*mouse N PL*
foxes	==>	*fox N PL*

Morphological parsing yields information that is useful in many NLP applications. In parsing, e.g., it helps to know the agreement features of words. Similarly, grammar checkers need to know agreement information to detect such mistakes. But morphological information also helps spell checkers to decide whether something is a possible word or not, and in information retrieval it is used to search not only *cats*, if that's the user's input, but also for *cat*.

To get from the surface form of a word to its morphological analysis, we are going to proceed in two steps. First, we are going to split the words up into its possible components. So, we will make *cat + s* out of *cats*, using + to indicate morpheme boundaries. In this step, we will also take spelling rules into account, so that there are two possible ways of splitting up *foxes*, namely *foxe + s* and *fox + s*. The first one assumes that *foxe* is a stem and *s* the suffix, while the second one assumes that the stem is *fox* and that the *e* has been introduced due to the spelling rule that we saw above.

In the second step, we will use a lexicon of stems and affixes to look up the categories of the stems and the meaning of the affixes. So, *cat + s* will get mapped to *cat NP PL*, and *fox + s* to *fox N PL*. We will also find out now that *foxe* is not a legal stem. This tells us that splitting *foxes* into *foxe + s* was actually an incorrect way of splitting *foxes*, which should be discarded. But note that for the word *houses* splitting it into *house + s* is correct.

Here is a picture illustrating the two steps of our morphological parser with some examples.

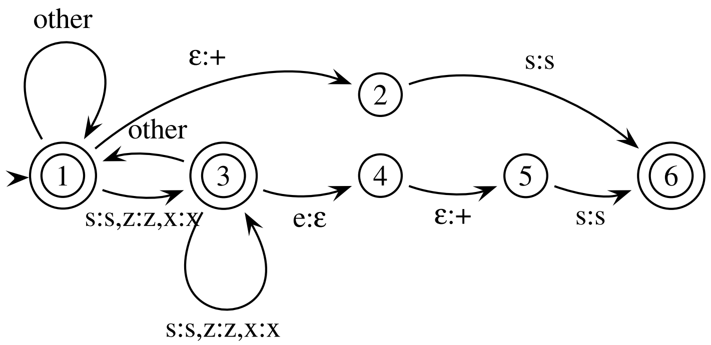

surface form
split word into possible morphemes
intermediate form
map stems to categories and affixes to meaning
underlying form

cat	*cat*	*cat N SG*
cats	*cat + s*	*cat N PL*
mouse	*mouse*	*mouse N SG*
mice	*mice*	*mouse N PL*
foxes	*fox + s*	*fox N PL*

We will now build two transducers: one to do the mapping from the surface form to the intermediate form and the other one to do the mapping from the intermediate form to the underlying form.

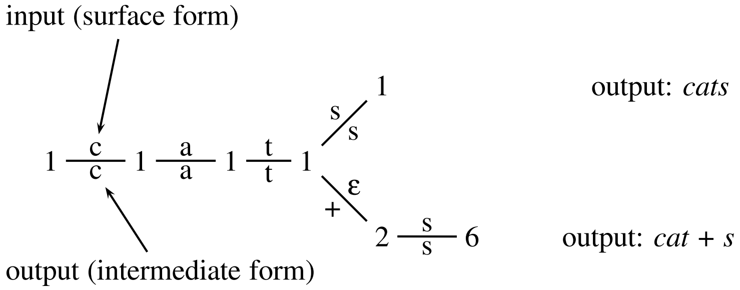
2.3.3 From the Surface to the Intermediate Form

To do morphological parsing this transducer has to map from the surface form to the intermediate form. For now, we just want to cover the cases of English singular and plural nouns that we have seen above. This means that the transducer may or may not insert a morpheme boundary if the word ends in *s*. There may be singular words that end in *s* (e.g. *kiss*). That's why we don't want to make the insertion of a morpheme boundary obligatory. If the word ends in *ses*, *xes* or *zes*, it may furthermore delete the *e* when introducing a morpheme boundary. Here is a transducer that does this. The "other" arc in this transducer stands for a transition that maps all symbols except for *s, z, x* to themselves.

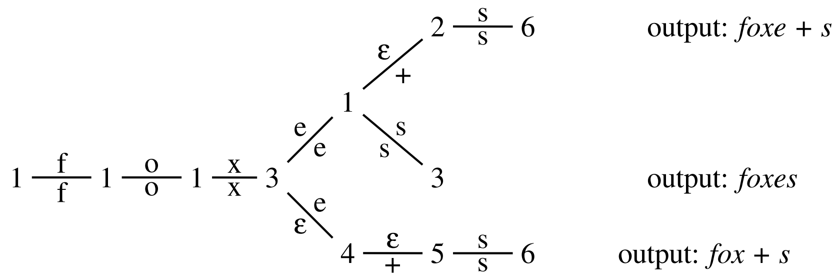


Let's see how this transducer deals with some of our examples. The following graphs show the possible sequences of states that the transducer can go through given the surface forms *cats* and *foxes* as input.

input *cats*:



input *foxes*:



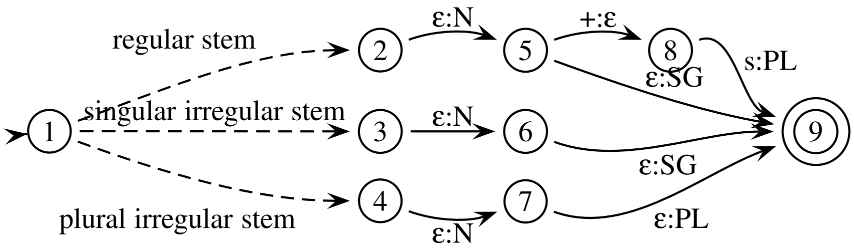
2.3.4 From the Intermediate Form to the Morphological Structure

Now, we want to take the intermediate form that we produced in the previous section and map it to the underlying form. The input that this transducer has to accept is of one of the following forms:

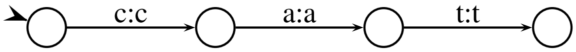
1. regular noun stem, e.g. *cat*
2. regular noun stem + s, e.g. *cat + s*
3. singular irregular noun stem, e.g. *mouse*

4. plural irregular noun stem, e.g. *mice*

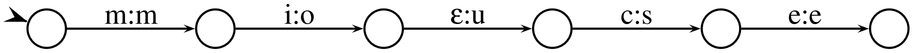
In the first case, the transducer has to map all symbols of the stem to themselves and then output *N* and *SG*. In the second case, it maps all symbols of the stem to themselves, but then outputs *N* and replaces *PL* with *s*. In the third case, it does the same as in the first case. Finally, in the fourth case, the transducer should map the irregular plural noun stem to the corresponding singular stem (e.g. *mice* to *mouse*) and then it should add *N* and *PL*. So, the general structure of this transducer looks like this:



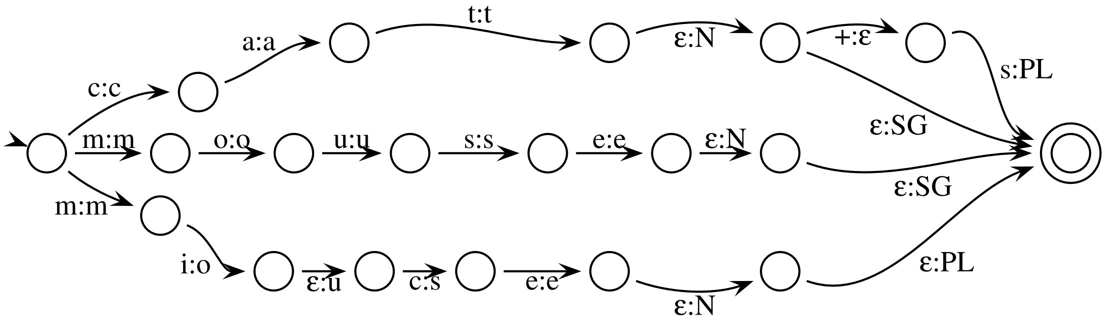
What still needs to be specified is how exactly the parts between state 1 and states 2,3, and 4 respectively look like. Here, we need to recognize noun stems and decide whether they are regular or not. We do this be encoding a lexicon in the following way. The transducer part that recognizes *cat*, for instance, looks like this:



And the transducer part mapping *mice* to *mouse* can be specified as follows:



Plugging these (partial) transducers into the transducer given above we get a transducer that checks that input has the right form and adds category and numerus information.



2.3.5 Combining the two Transducers

If we now let the two transducers for mapping from the surface to the intermediate form and for mapping from the intermediate to the underlying form run in a cascade (i.e. we let the second transducer run on the output of the first one), we can do a morphological parse of (some) English noun phrases. However, we can also use this transducer for generating a surface form from an underlying form. Remember that we can change the direction of translation when using a transducer in translation mode.

Now, consider the input *berries*. What will our cascaded transducers make out of it? The first one will return two possible splittings, *berries* and *berrie + s*, but the one that we would want, *berry + s*, is not one of them. The reason for this is that there is another spelling rule at work, here, which we haven't taken into account at all. This rule is saying that "*y* changes to *ie* before *s*". So, in the first step there may be more than one spelling rules that all have to be applied.

There are basically two ways of dealing with this. First, we can formulate the transducers for each of the rules in such a way that they can be run in a cascade. Another possibility is to specify the transducers in such a way that they can be applied in parallel.

There are algorithms for combining several cascaded tranducers or several transducers that are supposed to be applied in parallel into a single transducer. However, these algorithms only work, if the individual transducers obey some restrictions so that we have to take some care when specifying them.

2.3.6 Putting it in Prolog

If you want to implement the small morphological parser that we have seen in the previous section, all you really have to do is to translate the transducer specifications into the Prolog format that we used in the last lecture. Then, you can use last lecture's transducer program to let them run.

We won't show in detail what the transducers look like in Prolog, but we want to have a quick look at the *e* insertion transducer, because it has one interesting feature; namely, the *other*- transition. How can we represent this in Prolog?

Assuming the transducer is called `c1`, here are all transitions that go out of state 6 in Prolog notation, except for the *other*-transition.

```
trans(c1,6,2,z:z).
trans(c1,6,2,s:s).
trans(c1,6,2,x:x).
trans(c1,6,3,'^':'^').
```

Now, the *other*-transition should translate *any* symbol except for *z*, *s*, *x*, ^ to itself. In Prolog, we can express this using cuts and exploiting the fact that Prolog searches the database top down:

```
trans(c1,6,2,z:z) :- !.
trans(c1,6,2,s:s) :- !.
trans(c1,6,2,x:x) :- !.
trans(c1,6,3,'^':'^') :- !.
trans(c1,6,1,X:X) :- !.
```

2.3.7 Further Reading

This lecture is partly based on Chapter 3 of [5]. There you can read more about morphology and finite state transducers and find some more details about the mathematical properties of FSTs.

- The introduction to morphology in this lecture was very quick and superficial. If you want to know more, any introductory text to linguistics should help.
- If you are interested in more details about the relation between transducers, regular relations, and rewriting grammars, have a look at [6].
- Gert-Jan van Noord has developed a system called FSA Utilities which let's you specify regular expressions and builds the corresponding automaton for you. It can also do various operations on automata (minimization, determinization, union ...)

2.4 The Complete Programs

See file `parse.pl`.	An FSA-based parser.
See file `cat_parse.pl`.	FSA-based parser for FSA with categories.
See file `haha.pl`.	'laughing-machines' from Section 1.4.1.
See file `harry.pl`.	FSAs for 'Harry Potter-phrases' from !!!UNEXPECTED PTR TO EX_FSA
See file `trans.pl`.	A driver for transducers.
See file `trans_lex.pl`.	A driver for transducers using lexicon entries (`lex/2`)
See file `a2b.pl`.	A transducer that translates `a`s into `b`s
See file `adoubler.pl`.	A transducer that doubles the number of `a`s that it reads on the first tape and

2.5 Practical Session

Now, we will again do some simple exercises going through the programs of this section step for step.

1. Start Prolog and consult `parse.pl`, and `haha.pl`. Use `trace` to step through some examples so you understand clearly what the additional argument is doing.
2. Restart Prolog and consult `trans.pl`, and `a2b.pl`. Use trace to step through some examples so you understand clearly how `trans.pl` works.
3. Consult `adoubler.pl`. Use `trace` again and step through some examples so you understand clearly how. Look how `#`-symbols are handled.
4. When you are sure you understand `trans.pl` properly, extend it so that it can cope with categories. This is not difficult: all you have to do is extend the definition of the `traverse/5`- predicate.
5. Then put the morphological transducer (which parses English noun phrases, see Section 2.3.4) into Prolog notation and test it with your new predicates.

6. Finally, pick another phenomenon of inflectional morphology in some language you know. Try to write down a transducer as a graph. Then put it in Prolog and test it with the programs provided in this section.

2.6 Exercises

Exercise 2.1 *Answer "true" or "false" to the following questions:*

1. *It is possible to write a finite state automaton (FSA) to generate/recognize any formal language.*
2. *It is not possible to write an FSA that generates/recognizes the formal language* $a^n b^n$.
3. *It is not possible to write an FSA that generates/recognizes the formal language* $a^n b^m$.
4. *We can generate more languages using non-deterministic finite automata than we can using deterministic finite automata.*
5. *A finite state transducer (FST) is essentially an FSA that works with 2 (or more) tapes.*

Exercise 2.2 *Write an FST that transduces between strings of* a*s (of length m), and strings consisting of a* c*, followed by m* b*s, followed by another* c*. For example, your FST should transduce between* aaa *and* cbbbc*. Write down your FST in both graphical and Prolog notation.*

Exercise 2.3 *Write an FST that, given a string of* a*s,* b*s, and* c*s, deletes all the* a*s, turns the* b*s to* d*s, and leaves the* c*s alone. For example, given the input* abbaabcbca*, your FST should transduce return* dddcdc. Write down your FST in both graphical and Prolog notation.

Exercise 2.4 *Design an FST that transduces number names in written form between two languages you know (for all numbers from 0 to 100). For example, if you write a French_English FST, this should transduce between 'ninety four' and 'quatre vingt quatorze', and if you write a Dutch_German FST, this should transduce between 'een en twintig' and 'ein und zwanzig'. You have seen a similar transducer in Section 2.2.1.*

Exercise 2.5 *In Lecture 1, we claimed that the formal language* $a^n b^n$ *was relevant to natural language, because natural languages contain various "balanced" syntactic constructions. Give an example from some natural language you are familiar with (German, Irish, Russian, French, English, Japanese, Chinese, Tagalog,...). Make sure you explain why your answer is relevant.*

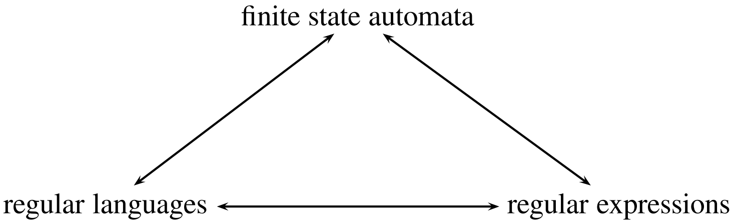
3

Regular Languages

3.1 Regular Languages and Relations

3.1.1 FSAs, Regular Expressions, and Regular Languages

First, let's go back to finite state automata. We said in Chapter 1 that the languages that FSAs can recognize are called *regular languages*. But there is another way of defining regular languages: Regular languages are exactly those languages that can be represented by *regular expressions*. And from this it follows that every automaton corresponds to a regular expression and vice versa. So, we get the following picture:

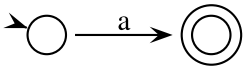


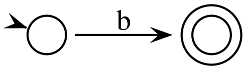
3.1.2 Examples of Regular Expressions

But what are regular expressions? Here are four examples:

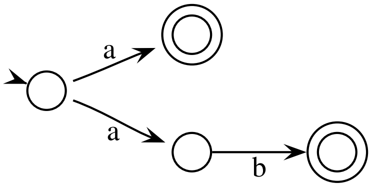
- a
- b
- a|ab
- (a|b)*

The building stones of regular expressions are symbols. These symbols can then be connected in a number of ways to represent sets of strings. Before we go into the details, let's look more closely at the above examples. a and b are regular expressions representing the singleton sets of strings $\{a\}$ and $\{b\}$. They correspond to the following automata:

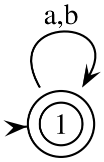




Regular expressions can be concatenated. The regular expression `ab` represents the (also singleton) set of strings $\{ab\}$. It should be clear what the corresponding automaton looks like. We can furthermore combine regular expressions by disjunction. The regular expression `a|(ab)` represents the set of strings $\{a, ab\}$ and corresponds to this automaton:



Finally, $(\mathsf{a}|\mathsf{b})^*$ is the set of all strings that consist of *a*s and *b*s in any order. The empty word is also accepted. The automaton looks as follows:



From the examples you might already have guessed that there is a systematic way of translating regular expressions into finite state automata, which means that we never actually have to specify automata - we only have to write regular expressions.

3.1.3 Definition of Regular Expressions

But before we have a closer look at this systematic relation between regular expressions and automata, let's give a more formal statement of what a regular expression is and what language it denotes.

First here's some notation to (independently, in terms of sets) describe languages. Let L, L_1 and L_2 be languages over a (finite) alphabet Σ (i.e., subsets of the set of all strings that can be formed from Σ). Then

L_1L_2	denotes the *concatenation* of these two languages, that is the language that contains all strings formed by *concatenating* a string from L_2 to a string from L_1 (picking a string from L_1 and following it by one from L_2).
L^*	denotes the *Kleene closure* of L. It contains all strings obtained by concatenating any number of strings from L. In particular it contains the empty string ε as the result of concatenating *zero* strings from L.

Definition

Now we're ready to turn to *regular expression* s themselves. Again given an alphabet Σ:

$\emptyset$ **and** ε	are regular expressions denoting the empty language, $\{\}$, and the language containing only the empty string, $\{\varepsilon\}$. Note that these two languages are *different*.
a	for each $a \in \Sigma$ is a regular expression and denotes $\{a\}$, the language consisting of the single symbol a.
rs, r|s **and** r*	are regular expressions, provided r and s are. If r denotes the language R and s denotes the language S, then rs denotes their concatenation RS, r|s denotes their union $R \cup S$, and r* denotes the Kleene closure of R, R^*.

3.1.4 Regular Expressions and FSAs

Now that we know precisely what regular expressions are, let us look systematically at their relation to FSAs. As we've already mentioned, both formalisms are equivalent. They both characterize the same set of languages known as the *regular language* s. We shall now prove one direction of this equivalence: If r is a regular expression, then there exists a FSA that accepts the language characterized by r. Proving this basically means stating more formally what we illustrated by examples in Section 3.1.2. To do so let us first introduce a canonical notation for FSAs. We write a FSA as a quintuple:

$$(Q, \Sigma, \delta, s, F)$$

In such a quintuple

Q	is the set of states of the automaton.
Σ	is the alphabet underlying the language characterized by the automaton.
δ	specifies what transitions can be made in the automaton
s	is the start state of the automaton.
F	is the set of final states of the automaton.

How does δ specify what transitions can be made? It states for each state-symbol-pair in the automaton, where the transition from the respective state over the symbol leads. So δ is a *relation* over $(Q \times \Sigma) \times Q$. We can also require that δ be a function, making the automaton deterministic.

Let's compare our canonical way of specifying a FSA to our Prolog notation from Section 1.4: There, we specified possible transitions (i.e. δ) by giving `trans/4`-terms. The initial state (s) was given as a `start/2`-term, and the final states (F) in the form of `final/2`-terms. The set of states Q as well as the alphabet Σ were left implicit. Here's how we formally specify the laughing machine from Section 1.3:

$$(\{1,2,3,4\}, \{a,h,!\}, \{((1,h),2), ((2,a),3), ((3,!),4), ((3,h),2)\}, 1, \{4\})$$

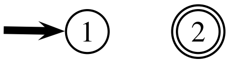
3.1.5 From Regular Expressions to FSAs

We now prove one direction of the equivalence between regular expressions and FSAs: We will show that there is a FSA for every regular expression. The nice thing about this proof is that it is contructive - it proceeds by showing us what such an automaton will actually look like.

Our proof is by induction over the number of operators (that is, occurences of concatenation, | and *) in a regular expression. We first show that there are automata for all possible singleton regular expressions. Then we assume that there are automata for the regular expressions with less than *n* operators, and show that there is one for each regular expression with n operators.

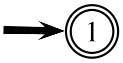
Basis

We first have to look at regular expressions without any operators. This means, we are talking about $\emptyset$, ε or a for some $a \in \Sigma$. For $\emptyset$ we can use the following automaton:



Obviously, we can never get to the final state in this automaton, therefore *no word* is accepted.

For ε, the following automaton can be given:



In this automaton we are at the final state at the moment we begin our computation (and we cannot get out of it). Therefore the automaton accepts only 'input' having *no symbol* (that is: the empty word).

We've already seen (in Section 3.1.2) the pattern of how to construct automata for a for any $a \in \Sigma$.

Induction

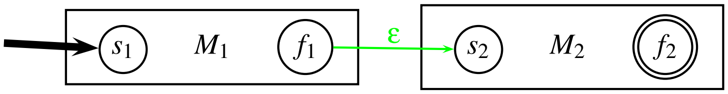
Now let's turn to complex regular expressions, i.e. ones that do contain operators. More precisely, we assume that there are automata for the regular expressions with fewer than *i* operators (with $i \geq 1$), and take a regular expression *r* with *i* operators. We have to look at three cases.

1. Concatenation: $r = r_1 r_2$
2. Union: $r = r_1 | r_2$
3. Kleene Closure: $r = r_1^*$

In all three cases, r_1 and (where applicable) r_2 have $i-1$ operators. Therefore there are FSAs for them by our induction hypothesis. We now have to build an automaton for r on the basis of these given automata. All we have to do is make a few additions according to the additional operator used in r. We will basically apply the ideas already seen in Section 3.1.2.

Let's go through the cases one by one. We assume that M_1 is an automaton for r_1, M_2 corresponds to r_2, and that the states in M_1 are disjoint from those in M_2 (we can always achieve this by renaming):

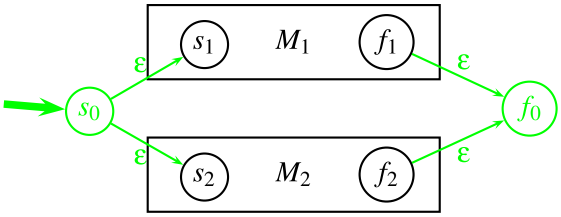
1. *Concatenation ($r = r_1 r_2$):* We construct an automaton M that consists of M_1 and M_2 *in series*. That is, M will accept a word by running M_1 on the first part, then jumping to M_2 and running it on the second part of the word. M will look like this:



The start state of M_1 is also the start state of M, and the final state of M_2 is also the final state of M. The final state of M_1 and the start state of M_2 have become 'normal' states, but are now connected by a jump arc. Formally (where indices 1 and 2 indicate that an entity originally belongs to M_1 or M_2):

$$M = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2 \cup \{((f_1, \varepsilon), s_2)\}, s_1, \{f_2\})$$

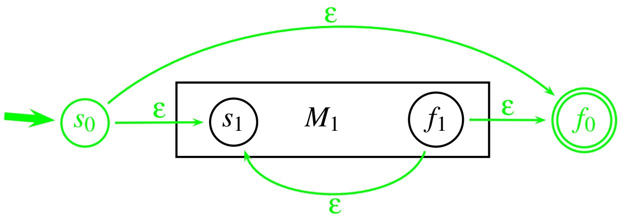
2. *Union ($r = r_1|r_2$):* We construct an automaton M that consists of M_1 and M_2 *in parallel*. That is, M will accept a word by jumping into *either* M_1 *or* M_2, running it on the word, and then jumping to a new final state. M will look like this:



The new start state s_0 is connected by jump arcs to the former start states of M_1 and M_2. Thus it serves as the common start state for going through M_1 *or* M_2. Similarly, the former final states of M_1 and M_2 are now both connected to a new final state f_0 by jump arcs. This new final state serves as the common final state for going through M_1 *or* M_2. Formally, the new automaton looks like this:

$$M = (Q_1 \cup Q_2 \cup \{s_0, f_0\}, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2 \cup \{((s_0, \varepsilon), s_1), ((s_0, \varepsilon), s_2), ((f_1, \varepsilon), f_0), ((f_2, \varepsilon), f_0)\}, s_0, \{f_0\}$$

3. *Kleene Closure ($r = r_1^*$:* We build a new automaton M that 'embeds' M_1. In M we can either bypass M_1 and jump directly from the start state to the final state, or loop by jumping into M_1, going through M_1, and then jumping back to the (former) start state of M_1. After looping an arbitrary number of times on M_1, we can jump to the final state. Thus M accepts ε (via the bypassing jump arc) as well as all combinations of words that are accepted by M_1 (by looping on the M_1-part of M). Here's the picture:



We had to add two states (the new start and final states s_0 and f_0), and four jump edges (one into M_1, one out of M_1, one to bypass it and one to loop). Thus this is the formal characterization of the new automaton M:

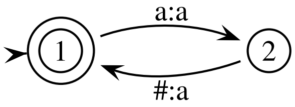
$$M = (Q_1 \cup \{s_0, f_0\}, \Sigma_1, \delta_1 \cup \{((s_0,\varepsilon),s_1),((f_1,\varepsilon),f_0),((s_0,\varepsilon),f_0),((f_1,\varepsilon),s_1)\}, s_0, \{f_0\})$$

So we've shown that for any regular expression there is an FSA that characterizes the same language. And in fact we've also shown how to construct such an automaton from any given regular expression. We just have to formulate a recursive procedure based on our inductive proof. We won't give an implementation that builds FSAs based on regular expressions here. But parts of the proof we've just seen will be very useful in Section 3.3, where we implement a tool for performing operations on given FSAs.

3.1.6 Regular Relations and Rewriting Rules

Before we turn to a closer examination of the class of regular languages, let us have a short look at finite state transducers. Is there anything to do the job of regular expressions for finite state transducers? And what corresponds to regular languages in this case?

Finite state transducers recognize tuples of strings. A set of tuples of strings that can be recognized by an FST is called a regular relation. So, regular relations are to FSTs what regular languages are to FSAs. The following transducer (we have already seen it in Section 2.2.1), for instance, recognizes the regular relation $\{\langle\varepsilon,\varepsilon\rangle, \langle a, aa\rangle, \langle aa, aaaa\rangle, \ldots\}$.



Regular relations can be specified using (ordered sets of) *rewriting rules*. The rewriting rules

$$\varepsilon \rightarrow e / s + \_ s\%,$$
$$\varepsilon \rightarrow e / z + \_ s\%,$$
$$\varepsilon \rightarrow e / x + \_ s\%,$$

for instance, express the *e*-insertion rule. The first one is read as "replace nothing (ε) with e in the context of $s+$ (s and a morpheme boundary, $+$) to the left and $s\%$ (s and a word boundary, $\%$) to the right. There are algorithms for translating such rule systems (at least as long as they obey certain restrictions) into transducers.

3.2 Properties of Regular Languages

3.2.1 Concatenation, Kleene Closure and Union

It follows from the definition of the operators of *concatenation*, * and $|$ that the set of regular languages is closed under concatenation, union and Kleene closure:

- If r is a regular expression and L is the regular language it denotes, then L^* is denoted by the regular expression r^* and hence also regular.
- If r_1 and r_2 are regular expressions denoting L_1 and L_2 respectively, then $L_1 \cup L_2$ is denoted by the regular expression $r_1|r_2$ and hence also regular.
- If r_1 and r_2 are regular expressions denoting L_1 and L_2 respectively, then L_1L_2 is denoted by the regular expression r_1r_2 and hence itself regular.

The rules for constructing FSAs based on these closure properties can be read off the respective parts of the inductive step of the proof in Section 3.1.5.

3.2.2 Intersection, Complementation

But the class of regular languages also has closure properties that are not obvious from the definition of regular expressions. We shall look at closure under *intersection* and *complementation*.

Intersection

If L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$. We will now construct a FSA M for $L_1 \cap L_2$ from two FSAs M_1 and M_2 for L_1 and L_2. The idea is to construct an automaton that simulates going through M_1 and M_2 synchronously, step by step in parallel. Formally (we use indices 1 and 2 to refer to elements of M_1 or M_2 respectively, and assume that both automata share the same alphabet):

$$M = (Q_1 \times Q_2, \Sigma, \delta, (s_1, s_2), F_1 \times F_2)$$

The states of the new automaton will are all pairs of states from M_1 with states from M_2.

The transitions will be such that the paths through M are the paths through M_1 if we only look at the first element of each state-pair, and are the paths through M_1 if we only look at second elements:

$$\delta = \{(((p_1,p_2),\ a),\ (q_1,q_2))\ | p_1,q_1 \in Q_1, p_2,q_2 \in Q_2, a \in \Sigma, ((p_1,a),q_1) \in \delta_1, ((p_2,a),q_2) \in \delta_2\}$$

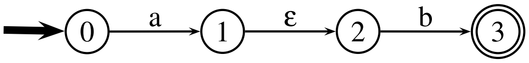
But our new automaton will accept a word only if the computation ends on a pair where *both* elements are final states in their ‘original automata’ (the set of final states is specified as $F_1 \times F_2$). This means that we can reach a final state with a word in M iff we could reach a final state with it in M_1 *and* in M_2, hence iff the word is in $L_1 \cap L_2$.

Complementation

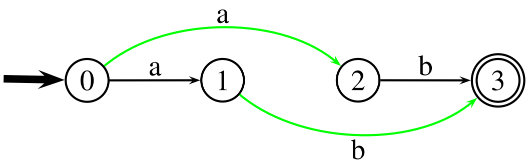
If L is a regular language over an alphabet Σ, then so is $\overline{L}$ (i.e. the language consisting of all strings from Σ^* that are *not* in L). We will again show how to construct an automaton M for $\overline{L}$ from an automaton M_1 for L. Yet this time we have to make three additional assumptions:

1. M_1 is ε-free, i.e. contains no jump arcs.
2. M_1 is complete, i.e. there is a transition from each state in Q over each symbol in Σ (note that none of the automata we've seen so far has had this property).
3. M_1 is deterministic.

However, these assumptions are quite harmless. We can get rid of jump arcs by systematically substituting direct transitions over the symbols read before and after each jump. For example, look at the following very simple automaton with one jump arc:

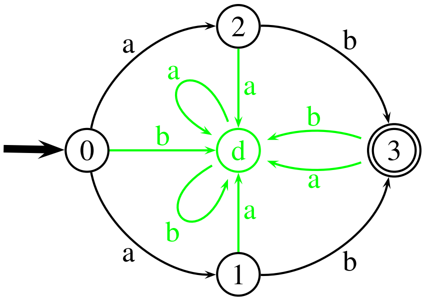


To get rid of the jump arc between states 1 and 2, we transform it into the following automaton:



The two transitions we've added so to speak bypass the former jump arc.

To *complete* an automaton we do as follows: We add one 'dead' state (a new state that has no outgoing transitions to any other states), and let all the 'missing' transitions point to it. For instance, here's a completed version of our ε-free automaton from above:



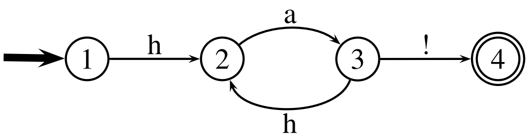
Finally, as we've already mentioned, non-determinism doesn't add anything substantial to FSAs, and every non-deterministic FSA can be transformed into a deterministic one. The algorithm fo this purpose is a bit more involved and we can present an easy example here. The basic idea is to build a new automaton that has as states all *sets* of states in the old one.

So for the construction of a complement-automaton, let's assume that M_1 is a determinstistic ε-free complete FSA for the language L. Then we construct an FSA M for $\overline{L_1}$ by copying M_1 and making all and only its non-final states final states of M. M will reach a final state on a word (and thus accept it) if and only if that word would have taken M_1 into a non-final state. Hence M will accept all and only those words from Σ^* that are not accepted by M_1.

Notice that once we know that regular languages are closed under union and complementation, closure under intersection follows from the set theoretic fact that $\overline{\overline{L_1} \cup \overline{L_1}} = L_1 \cap L_2$. An automaton for any intersection language can be contructed accordingly by nesting the construction methods for union and complementation. But because to prove closure under complementation, we had to make assumptions that we don't need for the direct construction method for intersection, we've chosen to present this direct method first.

3.2.3 The 'Pumping Lemma'

We will now look at another interesting property of regular languages that often allows us to tell that a language is *not* regular. We will first illustrate the basic idea. Consider our simple 'laughing machine' (already well known from Section 1.1.1):



The automaton has four states. This means that the only word it can accept without visiting any state more than once is the three-symbol word *ha!*. All other words from the 'laughing language' require it to visit some state(s) more than once. We can easily see from the diagram that these are the states 2 and 3. They will be visited twice when accepting *haha!*, three times when accepting *hahaha!*, etc.

This observation can be generalized: Let M be an automaton with n states that accepts a language L. Then for any word $w \in L$ that consists of n or more symbols, there must be at least one state in M that is visited more than once when w is processed. Now if a state (or sequence of states) in a FSA can be visited more than once, it can be visited any number of times. This is so becaues such a FSA must contain a loop, and we are free to go through it as often as we like. Let's suppose M reads the word v while going through the loop. So we know that w can be split in parts as follows: $w = w_1 v w_2$. But this means that, by looping on v, M will as well accept $w_1 vv w_2$, $w_1 vvv w_2$, etc.

Take for instance our laughing machine above. We had to go through 2 and 3 twice to accept *haha!*, using the loop from 2 to 3 and back. On our way through this loop we read *ha* (the second occurence, to be precise). And we can go through this loop any number of times. Looping like this we accept *hahaha!*, *hahahaha!* etc.

The above considerations lead to the following lemma, known as the 'Pumping Lemma', because it says that we can 'pump' up words in regular languages under certain conditions. (By w^i we mean the result of repeating w i-times; $w^i = \varepsilon$. By $|w|$ is the length of w.):

Let L be a regular set. There is a constant n such that if w is any word that is in L and is of length greater n, then we can split w in parts w_1vw_2 with $|w_1v| \leq n$ and $|v| \geq 1$, and $w_1v^iw_2$ is in L, too for all $i \geq 0$. Moreover, we know that n is no greater than the number of states in the smallest FSA accepting L.

The main use of the pumping lemma is in showing that a given language is *not* regular. Consider for instance the language L_p of all palindromes ('symmetrical' words that are the same whether you read them forward or backwards) that can be formed over some alphabet containing more than one symbol (say $\{a,h\}$). L_p contains words of any finite length. Now if L_p was regular, all of its words exceding a certain length n could be 'pumped' on, and all the resulting words would also be in L_p. In particular, this would be

Now take the palindrome a^nh^n. The 'Pumping Lemma' states that there is some substring within the first n symbols, thus a substring consisting entirely of as, that we should be allowed to repeat arbitrarily often without leaving L_p. Yet obviously any such repetition of as would destroy the symmmetry in p and the resulting word would no longer be a palindrome.

3.3 Implementing Operations on FSAs

3.3.1 The `scan`-Predicate

All operations on FSAs that we are going to implement will be carried out within one general framework: We will define a predicate `scan/1`. This predicate takes as input a symbolic specification of the operation to be performed. Then it traverses the FSA(s) referred to in its input and calls helper predicates according to the operation that has been requested. We will use the Prolog operators `;` and `,` to specify union and intersection in the input to `scan/1`. But how can we refer to the FSAs ot be operated on? Recall that our Prolog notation (see Section 1.4) requires us to give a name to each FSA that we put in the database (the first argument of the clauses that make up the automaton). And it is these names that we use in the operation specifications for the `scan/1`-predicate.

Here are two examples. Let's assume our database contains two automata named `a1` and `a2` (accepting L_1 and L_2, respectively). Then the specification used to request the union-automaton (for $L_1 \cup L_2$) will be:

```
a1;a2
```

To specify an automaton for $L_1 \cap L_2$ we write:

```
a1,a2
```

As indicated, our `scan/1`-predicate will have only one argument. Now if this single argument is to be filled by the specification of the operation to be performed, there's no argument left for output. Of course we don't want to lose the automaton resulting from performing the requested operations. But we won't use an additional argument for output. Instead, we will implement an operation `cp` that saves an automaton under a new name. For instance calling

```
scan(cp(a1,b1)).
```

results in the automaton `a1` being copied to an automaton `b1`. If for instance `a1` is our first 'laughing machine' from Section 1.4.1, we will have the following additional database entries after running `scan(cp(a1,b1)).`:

```
start(a1,1).

final(a1,4).

trans(a1,1,2,h).

trans(a1,2,3,a).

trans(a1,3,4,!).

trans(a1,3,2,h).
```

The `cp`-operation is used to save the output of other operations. For instance to intersect `a1` and `a2` *and* save the result as `b2`, we call:

```
scan(cp((a1,a2),b2)).
```

3.3.2 Implementation of `scan/1`

Now we will have a look at the code of `scan/1` and its helpers. We will again assume that our first laughing machine (see Section 1.4.1) is in the database under the name `a1` and discuss what happens when we copy it to `b1` by calling `scan(cp(a1,b1)).`

Here's the code of the toplevel predicate `scan/1`. It has only one clause:

```
scan(A):-
  retractall(state(A,_)),        % retract state markers
  start(A,S),                    % find start state
  scan_state(A,S).               % start scanning from there
```

Line 1 contains the head of the unary predicate `scan(A)`. Its single argument is the specification of the operation to be carried out. Hence in the case of our example, `A` will be `cp(a1,b1)`. Line 2 then removes all database entries with the pattern `state(A,_)`. As we will see soon, our system uses such entries to remember what has already been done. Some such `state/2`-entries might still be in the database from earlier runs of `scan/1`. So it's a good idea to start by removing all of them.

Line 3 consists of a call to `start/2`. Here, the execution of our program branches according to the operation specified as input. Different clauses of `start/2` match for different properties. They call different other predicates, but there is one invariant: All of them output a state name in `S`, namely the name of the start state of the automaton resulting from the requested operation. So in our example with `A=cp(a1,b1)`, `S` will be `1`. To see how this is achieved and what else happens, let's look at the relevant clause of `start/2`:

```
start(cp(A1,A2),S):-
  start(A1,S),
  retractall(start(A2,_)),
  retractall(trans(A2,_,_,_)),
  retractall(final(A2,_)),
  assert(start(A2,S)).
```

In our example `A1` and `A2` are unified with `a1` and `b1`. First of all in the body, the name of the start state of `a1` is retrieved (resulting in the instantiation `S=1`). Next any potential remains of other automata name `b1` are removed. Finally, and a start state of the name just retrieved is asserted for `a2`.

3.3.3 `scan_state/2`

Now control returns to the top-level call of `scan/1`, the database contains a new entry `start(a2,1)` and `S` is instantiated to `1`. Next thing in `scan/1` is a call to `scan_state/2`. This predicate is the most important part of our system. Here's the code:

```
scan_state(A,S):-
  state(A,S),!.                 % state already scanned?
```

The predicate consists of two clauses. The second one is the one that succeeds in our example, so let's look at it first. After unifying the two arguments `A` and `S` respectively with the the instruction `cp(a1,b1)` and the state name `1` in the head, the call to `assert/1` in the first line of the body adds an entry `state(cp(a1,b1),1)` to the database. This marks state `1` as processed in copying `cp(a1,b1)`. Next, it is checked if `S` is a final state. This is done using `check_final/2`. We will look at the code later. Suffice it to say for now that `check_final/2` will not do anything and just succeed trivially in our example because state `1` isn't a final state.

The final lines of `scan_state/2`-predicate make use of a special Prolog construction similar to 'if-then-else'-constructions in imperative languages like Basic. This construction has the form `(IF -> THEN; ELSE)`.

So in the `scan_state/2`-predicate, the `IF` part contains a call to `setof/3`. Basically, this call gives us all states `D` which are the destination of a transition originating from state `S` over any symbol `X` in the automaton resulting from the operation that's just being performed. The result is a list bound to the variable `Ds` ('destinations'). Just as in `a1` itself, we can only reach state `2` from `1` in the copy that we are making. So `Ds=[2]`.

If `setof/3` does not find any destinations, it fails, the 'if-then-else'-construction activates the `ELSE`-part. This part contains only the atom `true`, letting `scan_state/2` succeed. But if `setof/3` finds any destination-states, the list containing them is further processed.

Before we discuss how this is done, let's have a closer look at what happens 'within' the `setof/3`-call in our example. Using the instantiations we've collected, the call looks as follows:

```
setof(D,X^trans(cp(a1,b1),1,X,D),Ds)
```

This means that `trans(cp(a1,b1),1,X,D)` is called and re-done until no further solutions are available. Like `start/2`, `/trans/4` is implemented differently for all kinds of operations that can be specified. Here is the clause for `cp`:

```
trans(cp(A1,A2),S,D,X):-
  trans(A1,S,D,X),
  assert(trans(A2,S,D,X)).
```

If the original automaton `A1` has a transition from `S` over `X` to `D`, then an according transition is asserted for its copy `A2`. The solution for `D` is the destination state of the transition found in the original automaton. In our example, a copy of the transition `trans(a1,1,2,h)` is asserted as `trans(b1,1,2,h)`. And because `trans(a1,1,2,h)` is the only transition out of state `1` in `a1`, `setof/3` has already finished its job and `Ds` is instantiated to `[2]` (as we've already said above).

3.3.4 Recursion

Now finally we come to the `THEN`-part in `scan_state/2`. It consists in calling `scan_states/2` (plural!) on the `Ds` produced by the preceding `setof/3`-call. The `scan_states/2`-predicate recursively calls `scan_state/2` (singular!) on every state on `Ds`:

```
scan_states(A,[S|Ss]):-
  scan_state(A,S),              % scan state from state list
  scan_states(A,Ss).            % go to next state from state list

scan_states(_,[]).              % state list already emptied?
```

In our example, this means that the transition `trans(a1,2,3,a)` is copied to `trans(a2,2,3,a)`. Then the next level of the recursion is entered by calling `scan_states/2` again, this time on `Ds=[3]`. Now there are two transitions. This means that `Ds` will have two members: `[2,4]`. State `2` has already been processed, and state `4` for has no outgoing transitions. This indicates that `a1` has been fully traversed and the next call to `scan_states/2` should end the recursion.

How is this done? For state `2`, the one that has already been processed, the first clause of `scan_state/2` will succeed, because the database contains the entry `state(cp(a1,b1),2)` that was asserted when state `2` was processed for the first time. This means that no further recursive level will be entered. For state `4`, recursion will end simply because `setof/3` in `scan_state/2` won't find any transitions and so `Ds` will be empty.

Finally notice that state `4` is the final state of `a1`. For this reason, the call to check `check_final/2` in the second clause of `scan_state/2` will not just trivially succeed. Rather it will really do something before it succeeds. Let's have a look at the code:

```
check_final(A,S):-
  final(A,S),!.                 % final states receive special treatment
```

```
check_final(_,_).              % non-final states: no processing necessary -
```

The first clause contains a call to `final/2`. This predicate is again implemented differently for `cp`, intersection and union. Here's the clause for `cp`:

```
final(cp(A1,A2),S):-
  final(A1,S),
  assert(final(A2,S)).
```

If `S` is a final state in the original `A1`, an according final state is asserted for the copy `A2`. In our example, the entry `final(b1,2)` will be asserted. For all non-final states, the above clause of `final/2` will fail and the second clause of `check_final/2` will succeed without any further actions.

Summing up, after running `scan(cp(a1,b1))`, the following will have happened:

- Due to the call to `start/2` in `scan/1`, the database will contain the entry:

  ```
  start(b1,1)
  ```

- The calls to `trans/2` in the second clause of `scan_state/2` will have added:

  ```
  trans(b1,1,2,h).
  trans(b1,2,3,a).
  trans(b1,3,4,!).
  trans(b1,3,2,h).
  ```

- The call to `final/2` in `check_final/2` will have added:

  ```
  final(b1,4).
  ```

That's a full copy of `a1`. Additionally, the database will contain the entries:

```
state(cp(a1,b1),1).
state(cp(a1,b1),2).
state(cp(a1,b1),3).
state(cp(a1,b1),4).
```

These were asserted in `scan_state/2` as control information and can be retracted.

3.3.5 Intersection

We now turn to the more interesting operations of union and intersection. We implement them simply by giving additional clauses for `start/2`, `trans/2` and `final/2`. Let's look at an example call to see how this works. We assume that our database contains two automata named a1 and a2 (accepting L_1 and L_2, respectively). As said above we call

```
scan(cp((a1,a2),b2)).
```

to build an automaton for $L_1 \cap L_2$ and save it as `b1`.

Just as when we were simply copying `a1`, the first substantial thing to happen is a call to `start/2`, like this:

```
start(cp((a1,a2),b2),S)
```

Again, this leads to another call to *start*/2. But this time, this call will not directly match a start state specification of any of the automata in the database. Rather it will look like this:

```
start((a1,a2),S)
```

The first argument contains our intersection symbol, the operator ‘`,`’. And we provide the following clause of `start/2` to match in this case:

```
start((A1,A2),S1/S2):-
  start(A1,S1),
  start(A2,S2).
```

This clause produces the start state of our intersection automaton. The start state has to be the pair of the start states of `a1` and `a2` according to the construction method discussed in Section 3.2.2. We represent this in Prolog using the `/`-operator.

Next, control returns to the embedding call `start(cp((a1,a2),b2),S)`, and the start state of the new `b1` is asserted to the database.

Similarly, the calls to `trans/4` issued in `scan_state/2` now lead to one further call, which processes the intersection operator. Here’s how we implement `trans/4` for intersection:

```
trans((A1,A2),S1/S2,D1/D2,X):-
  trans(A1,S1,D1,X),
  trans(A2,S2,D2,X).
```

This code translates our ideas from Section 3.2.2: If a transition over some symbol is present in `A1` *and* `A2`, the intersection-automaton should contain a transition over that symbol between the corresponding pair-states. The above code ‘builds’ this transition. It is then asserted when control passes back to the embedding call that performs the `cp`-operation.

The third predicate that has to be extended is `final/2` (called by `check_final/2` in `scan_state/2`). Here is the clause we will add:

```
final((A1,A2),S1/S2):-
  final(A1,S1),
  final(A2,S2).
```

Any pair of final states of `A1` *and* `A2` is a final state of the intersection automaton. Again, the new final states are asserted by the embedding `final/2`-call processing the `cp`-instruction.

3.3.6 Union

To obtain a method for constructing union-automata, we have to translate some of the ideas underlying our proof of the equivalence between regular expressions and FSAs in Section 3.1.5. We proceed again by giving further clauses of `start/2`, `trans/4` and `final/2`.

Remember that we gave the automaton for the union of two regular languages a brand new start state. This is encoded in the clause we add to `start/2`:

start((_;_),start).

Assume that we are building a union-automaton from `a1` and `a2`. The first `setof/3`-call will then search for transitions out of the new start state as follows:

```
setof(D,X^trans(cp((a1;a2),b1),start,X,D),Ds)
```

This leads to a call:

```
trans((a1;a2),start,X,D)
```

As we've seen in Section 3.1.5, our union-automaton should contain jump arcs from the new start state to (copies of) the start states of `a1` and `a2`. We add the following clauses to `trans/4` for this purpose:

```
trans((A1;_),start,1/S1,'#'):-
  start(A1,S1).

trans((_;A2),start,2/S2,'#'):-
  start(A2,S2).
```

When we were building a union-automaton from two automata M_1 and M_2 in Section 3.1.5, we assumed that the states in both automata were disjoint. To the same effect we will generally give states that come from M_1 a prefix `1/` and states that come from M_2 a prefix `2/` in our implementation of the union-operation. So even if M_1 and M_2 have some state names in common, the corresponding states in our union automaton will have different names.

The interior of our union automaton is build up by taking over all transitions from the automata to be united and prefixing their start and end states by `1/` or `2/`:

```
trans((A1;_),1/S1,1/D1,X):-
  trans(A1,S1,D1,X).

trans((_;A2),2/S2,2/D2,X):-
  trans(A2,S2,D2,X).
```

For the final states, we deviate a little from the method discussed in Section 3.1.5. There we introduced *one* new final state and added jump arcs into it from all former final states. But we don't have to insist on having only one final state in our union-automaton. So we can simply prefix and take over the final states from the two input automata. Hence we just add the following two clauses to `final/2`:

```
final((A1;_),1/S1):-
  final(A1,S1).

final((_;A2),2/S2):-
  final(A2,S2).
```

3.4 The Complete Code

See file `operations.pl`.	The complete program for operations on FSAs
See file `hahuho.pl`.	Further 'laughing-machines' needed in the Practical session.
See file `deterministic.pl`.	Deterministic and ε free 'laughing machine' for use in the Exercises.
See file `recognize.pl`.	The recognizer/generator for FSAs with or without jump arcs (well kn

3.5 Practical Session

Hint: In newer versions of SWI-Prolog (at least) you can use the built-in predicate `atom_chars/2` to convert an atom into a list of one-character atoms. For instance calling

```
atom_chars(hallo, X).
```

yields:

```
X = [h, a, l, l, o]
```

So you can use `atom_chars/2` to prepare the input for your FSA like this:

```
atom_chars('haha!', X), test(a1,X).
```

Note that if the word you want to split in single letter contains an exclamation mark, you have to enclose the whole word in single quotes (as seen in the example).

Now, let's see if our program for manipulating FSAs really works as it should.

1. Have a look at `hahuho.pl`. This file contains two automata, named `hahu` and `haho`. They recognize different kinds of laughter. For `hahu`, the laughter can be any sequence of 'ha' and 'hu', ended by an exclamation mark. For `haho` it has to consist of 'ha' and 'ho' and also end in an exclamation mark. These are the automata we will test our new program on, so make sure you understand them.

2. Start Prolog and consult `recognize.pl`, `hahuho.pl` and `operations.pl`. (You may see some warnings. Don't mind...) Use `test/2` to test `hahu` and `haho` on some laughter of your choice.

3. Think about how to characterize the language that the union-automaton of `hahu` and `haho` should accept.

4. Build and test the automaton. First request our program to build the union-automaton of `hahu` and `haho` (indicated by `;`) and then save it under the name `u` (using `cp`), as follows: `scan(cp((hahu;haho), u)).`

   Now you can use `test/2` to test `u` on any input that you think should or shouldn't be accepted. For instance try `test(u, [h,a,h,u,!]).`, `test(u, [h,a,h,o,!]).`, and `test(u, [h,u,h,o,!]).`.

5. Describe (for yourself...) the language that the intersection-automaton of `hahu` and `haho` should accept.

6. Build the intersection-automaton (using `,`) and save it under the name `s`. Then test `s` on several inputs that it should or shouldn't accept.

7. Finally, use `generate/2` to generate with `hahu`, `haho`, `s` and `u` (if you don't remember how to do this, look it up in Section 1.4.2). What do you see? Have a look at the code to explain your observations.

3.6 Exercises

Exercise 3.1 *What language does the regular expression* $(\mathsf{hi}|(\mathsf{ha}\ (\mathsf{ho})^*))^*$ *describe? Sketch the step-by-setp construction of an automaton along the lines of our proof in Section 3.1.5. To make this task a bit less cumbersome, here are some simplifications that you can make:*

- *You don't have to treat* hi, ha, *and* ho *as complex, that is you can read each of them as a whole word over one edge.*
- *You don't have to give names to the states. Just draw empty circles.*
- *The solution you hand in needn't contain all intermediate step. That means you can re-use for the next one what you've drawn in one step and don't have to copy over and over.*

Exercise 3.2 *Now draw an automaton for* $(\mathsf{hi}|(\mathsf{ha}\ (\mathsf{ho})^*))^*$ *as you would normally do it (that is, don't stick to the proof in Section 3.1.5). Compare!*

Exercise 3.3 *In Section 3.1.5 we looked at one direction of the equivalence between regular expressions and FSAs. We showed that every language that can be characterized by a regular expression is also accepted by some FSA. Now let's look at the other direction and prove that for every FSA there is a regular expression that characterizes the same language.*

Let

$$M = (\{q_1, \ldots, q_n\},\ \Sigma,\ \delta,\ q_1,\ F)$$

be a deterministic FSA, and L be the language accepted by it. We are going to show that there is also a regular expression for L.

Before we begin, let's define the following notion: R_{ij}^k *are those strings that M reads when it goes from i to j without passing through a state numbered higher than k (by passing through, we mean entering* and *leaving again). So i and j may have numbers greater than k. We will use* R_{ij}^k *to 'zoom' in and out of our automaton, determining that only a part of it may be used.*

Now our induction will be over k. Starting from single states we successively allow ourselves to use larger parts of our automaton. We have to show that (given any choice of $i, j \leq n$*):*

Basis *There is a regular expression that characterizes* R_{ij}^0*. This is the language of all strings that take M from i to j without going through any state numbered higher than 0 (that is infact, without going* through *any state at all).*

Induction *There is a regular expression for* R_{ij}^k*, given that there is one for* R_{lm}^{k-1} *for any choice of* $l, m \leq n$*.*

Give proofs for basis and induction! To prove the basis, you can of course simply enumerate as a disjunction all the words that can be read in M in one step from i to j. For the inductive step, think about the following: If a word can be read from i to j in M without going through any state higher than k, it can always be split into a sequence of parts none of which takes M through a state higher than $k-1$ *(in the sense of entering* and *leaving). How can this be done?*

Exercise 3.4 *Add a 'pretty-print' operation (you can call it* `pr`*) to the operations defined in Section 3.3. It should use the backbone of the* `scan/1`*-predicate to crawl along an automaton and print out messages with information about any state it visits and about any edge it passes.*

For instance loading See file `haha.pl`*. and then calling* `scan(pr(a1)).` *should result in output like the following:*

```
automaton a1:

initial state 1

from 1 to 2 over h
etc...

final state 4

yes
```

Exercise 3.5 *In Section 3.2.2 we saw how to build complement automata, that is how to make an automaton M that accepts some language L into an automaton* $\overline{M}$ *that accepts* $\overline{L}$*.*

The method we discussed there works for FSAs that are complete, deterministic and ε*-free. We did not implement a complementation-operation in our Prolog system in Section 3.3 because we did not want to bother with establishing these properties.*

Add a complementation-operation `comp` *to the implemetation. Your operation may simply assume that the input FSA is complete, deterministic and* ε*-free. The file See file* `deterministic.pl`*. contains a complete, deterministic (and of course* ε*-free) version of our very first laughing machine. Use this file to test you implementation.*

3.7 Further Reading

- References already mentioned in Section 1.9 are [4], [7], and [5]. This chapter follows the exposition of FSAs and regular languages given in Chapters 2 and 3 of [4].
- The implementation presented in the third part of this chapter is based on the Fistame system implemented[1] and documented[2] by Ralph Debusmann. Fistame uses ideas from a course by Martin Kay and Ronald Kaplan. The slides from this course are available online[3].

---

[1] `http://pictures_eps/www.ps.uni-sb.de/~rade/papers/fistame.tar.gz`

[2] `http://pictures_eps/www.ps.uni-sb.de/~rade/papers/fistame.pdf`

[3] `http://www.coli.uni-sb.de/~kay/algorithms.pdf`

4

Finite State Technologies for Dialogue Processing

4.1 Dialogue Processing

4.1.1 Introduction

Dialogue is at the same time the most fundamental and broadly used form of language, as well as the most complex one. And most of all, dialogue is the most natural medium of communication for human beings. It is these aspects of dialogue that make its modelling a research area of its own, and one of greatest interest at that. In many cases of interaction between humans and machines, using dialogue may enables better results with regard to the target each time.

We now consider some general desired characteristics that any dialogue system should have, and give examples in the form of dialogue fragments. This will serve as background when we look at a real life-application of dialogue processing with finite state techniques, namely at the speaking elevator of the Computational Linguistics department of Saarland University.

But before we do any of that, have a look at the following.

ELIZA

Eliza (to be found here[1]) is one of the first dialogue systems. You can regard it as your friend. Tell it your problems and it will try to help you. It cannot really understand what you tell it. It picks on words that it then maps onto specific output. Play with it for fun. Do try to work out how it operates!

4.1.2 General Dialogue characteristics

We'll now give some general characteristics that should be handled by a dialogue manager in some way or another. Besides giving insight into how dialogues work and how they're structured, these characteristics serve as beackround for the evaluation and classification of dialogue systems. We will use them when we discuss the dialogue system of the Saarbrücken speaking elevator later in this chapter, and you will also have the opportunity to apply them to further systems in your exercises.

---

[1] `http://www-ai.ijs.si/eliza/eliza.html`

So what are the general characteristics of dialogues that a dialogue manager should be able to handle?

1. *Turn-taking:* When, how, and for how long should each dialogue participant talk? (Example ((page 50)), Elevator characteristic (Section 4.5.1)).
2. *Adjacency pairs and insertions:* What is the appropriate way of responding to a given input? How can one correct what one has said, ask for additional information in order to and before one can provide the appropriate response, etc.?(Example ((page 51)), Elevator characteristic (Section 4.5.2)).
3. *Grounding:* How does a dialogue participant make sure that their contribution is rightly understood, or that they themselves have understood correctly a previous contribution? How can misinterpretations be corrected? (Examples ((page 51)), ((page 51)), ((page 51)), Elevator characteristic (Section 4.5.3).)
4. *Dialogue context:* How do dialogue participants use the contributions and the conclusions previously drawn to interpret the current contribution and decide on their next contribution? (Example ((page 51)), Elevator characteristic (Section 4.5.4).)
5. *Ellipsis:* How do dialogue participants make sense of fragmentary responses that are often provided in dialogue situations?(Example ((page 52)), Elevator characteristic (Section 4.5.5).)
6. *Reference resolution:* How can participants disambiguate what referring expressions refer to? (Example ((page 52)), Elevator characteristic (Section 4.5.6).)
7. *Mixed initiative:* What is the amount, quality and level of contribution of every participant? (Example ((page 52)), Elevator characteristic (Section 4.5.7).)

4.1.3 General Dialogue characteristics: Examples

This was a bit abstract. So next, let's look at some examples.

Turn-taking

A: Is there something bothering you or not?
(1.0)
A: Yes or no?
(1.5)
A: Eh?
B: NO

A asks a question and that signals a point where the turn passes on to B. Since there is a significant amount of time where no response is produced by B, A reclaims the turn and repeats the question. That happens twice, until B actually responds.

Adjacency pairs and insertions

A: Where are you going?
B: Why?
A: I thought I'd come with you.
B: I'm going to the supermarket

One of the commonest adjacency pairs is the question-answer pair. This means that whenever a question has been asked, an answer is to be expected, or is appropriate. In this example, B does not answer the question A has asked. That, however, is all right, since B is not ignoring the question. He is just initiating a sub-dialogue in order to clarify the reasons as to A's asking the question and be able to better respond.

Grounding

A: OK. I'll take the 5ish flight on the night before the 11th.
B: On the 11th? OK. Departing at 5:55pm arrives Seattle at 8pm, U.S. Air flight 115.
A: OK.

A gives a departure date. B briefly states what he has understood the departing date to be and goes ahead to give the details of the itinerary on that date.

What if A changed his mind and wanted to take a later flight, after all?

A: Hm, actually, the 8ish flight would be better.

Or, if the system hadn't understood him correctly, for example, due to mis-hearing?

B: On the 10th? OK. ...

A dialogue system must be able to deal with these cases, especially as with the current state of the art in speech recognition there is a lot of room for misrecognition. That means, that the system should be able to delete information and resume the dialogue at the right point.

Dialogue context

A: That's the telephone.
B: I'm in the bath.
A: OK.

B gives an answer which is not at first glance relevant to A's assertion that the telephone is ringing. If interpreted in context, though, A's original assertion must have also been a prompt for B to pick the telephone up. Therefore, B's utterance is interpreted as stating B's inability to pick up the phone. This particular example requires a lot of world knowledge as well as dialogue context to be taken into account. World knowledge, for instance, is used when A realises that B cannot pick up the phone. A is aware of the fact that getting out of the bath is not what most people enjoy doing in order to answer a telephone call.

Ellipsis

A: Which day would you like to travel?
B: Friday.
A: You want to travel on Friday.

B is able from the fragmentary response ‘Friday’, to reconstruct the semantics of something like ‘I want to travel on Friday’.

Reference resolution

A: We need to get the woman from Penfield to Strong.
B: OK.
A: What vehicles are available?
B: There are ambulances in Pittsford and Webster.
A: OK. Use one from Pittsford.

B is able to map ‘vehicles’ onto the vehicles in Penfield or near Penfield, as A has previously specified the place where the woman has to be transfered from to be Penfield and, thus, that that will be the place of departure. B also infers that the vehicles should be ambulances, since this kind of vehicle is normally used for transferring people. The interpretation here is made easier because B can also infer that the woman is injured, given the domain. This example is taken from the TRIPS system, a quite sophisticated system which we will look at briefly later in this lecture.

Mixed initiative

B: There are ambulances in Pittsford and Webster.
A: OK. Use one from Pittsford.
B: Do you know that Route 96 is blocked due to construction?
A: Oh.
A: Let’s use the interstate instead.
B: OK, I’ll dispatch the crew.

A’s suggestion to use the particular ambulances makes the system infer that Route 96 needs to be used. Thereupon, it takes the initiative and informs A that that route is blocked, implying that it is not a good choice. In the final turn, A takes the initiative again. It offers to dispatch the crew of the ambulance once the route has been decided on.

4.2 FSA-based dialogue systems

4.2.1 Processing dialogue with finite state systems

At the heart of the speaking elevator’s dialogue processing capabilities, there’s a FSA much like the ones we’ve looked at in the last lectures. We will now have a closer look at how finite state automata can be adapted for dialogue processing. When modeling a dialogue with an automaton, the key idea is to think of the states of that automaton as standing for different states of the dialogue, and of its edges as corresponding to things that happen in the dialogue.

So states are always defined with certain expectations as to what the system can have had as input, and what else can have happened at certain stages of the dialogue. For instance, the initial state of a dialogue automaton is naturally the beginning of the dialogue, and final states will normally correspond to possible end-points of a dialogue.

Edges of a dialogue automaton may for instance be defined to take the user input into account in order to decide what the new system state is, and to activate certain procedures, for instance output or movements. That way whenever a particular input is recognised by the system, a predefined behaviour and the way to realise it can be produced next.

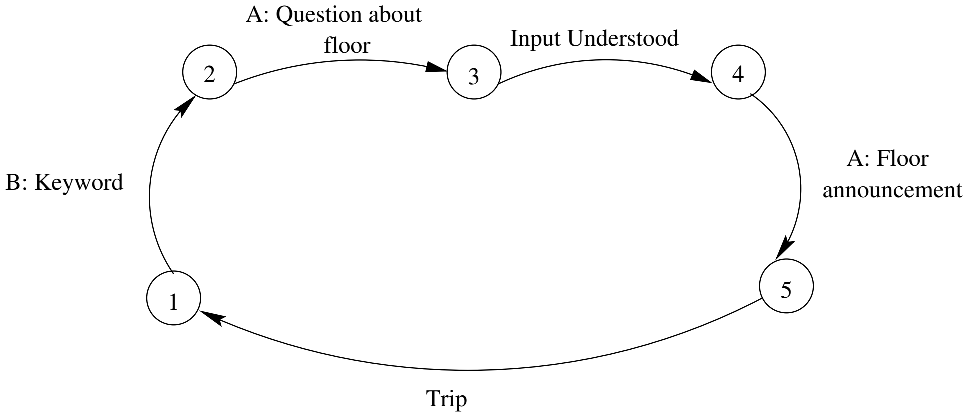
So one adaption that has to be made to use FSA for dialogue processing is to allow various kinds of actions to be connected to edges, other than just consuming given input. Although this step is formally quite far-reaching, it is conceptually simple. For the practical purposes we're going to look at, the striking simplicity of FSAs remains untouched.

4.2.2 A simple FSA example

The dialogue module of the speaking elevator is modeled by a finite state automaton. It handles input and output. For this purpose it communicates with a speech recogniser and a text to speech synthesiser. Specific kinds of spoken input are expected according to the current state. Besides that, it also communicates with the hardware and controls mechanical actions.

As a first example let us look at a FSA dialogue manager that could be used to produce very simple elevator dialogues. As we will soon see, the real speaking elevator's FSA is somewhat more complex. Still it is based on the same structure. The dialogue module of the speaking elevator handles input and output and it controls mechanical actions. It communicates with a speech recogniser and a text to speech synthesiser. Specific kinds of spoken input are expected according to the current state. It also communicates with the hardware. So a very simple FSA for elevator dialogues consists of five states and five arcs and looks as follows:

A simple finite state automaton



State 1 is the initiating state. In the elevator it is expecting a particular keyword. When the keyword is recognised, the system is set to State 2. A question about which floor the user wants to go to is produced, which sets the system to state 3. The user's input is recognised, which sets the system to state 4. The system informs the user of the floor he is being taken to and the state after that is State 5. After State 5, the trip begins and the system is reset.

From this picture we can see another difference between finite state systems used in dialogue processing and FSAs in the strict sense as we have seen them in previous lectures: Dialogue automata don't always have final states. This is of course because dialogues systems - like for instance in an elevator - often need to be ready for new input immediately when one dialogue is over, and shouldn't be 'switched off' after one interactive sequence. This can be symbolised by simply looping back to the intial state. In the above picture, we could just as well regard state 5 as final state and view the trip-and-reset as external to the automaton.

Example dialogue

Our simple automaton is able to generate only dialogues of the following kind.

User:	Elevator.
System:	Which floor do you want?
User:	Professor Bill Barry.
System:	I'm taking you to the Fifth floor.
	(Mechanical command execution follows)

One thing that this dialogue does not allow the user is to confirm or correct the system in case of a misrecognition. We will now look at various ways of mending this shortcoming.

4.2.3 Extending FSA

The kind of dialogue that the automaton we've just seen allows is only an abstraction of the expected dialogues. It covers so few of the characteristics that make dialogue an efficient way of communicating that it almost defeats the purpose of using dialogue at all. A drop-down menu would probably be equally efficient.

Let us see, then, what a programmer has to do in order to extend a FSA dialogue manager in order to encapsulate the dialogue characteristics considered in Section 4.1.2. We will do that by looking at an example, namely, handling grounding.

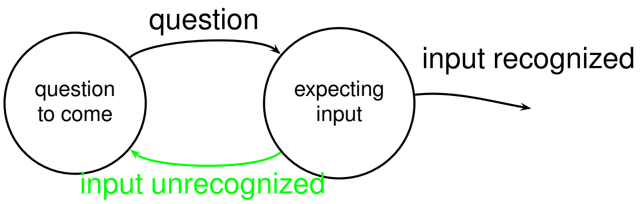
4.2.4 Grounding extension

So let's look at the possibilities of extending the simple automaton in Section 4.2.2 to cover different forms of grounding. Due to the low performance of the state of the art speech recognisers, it is absolutely necessary for a system to model this feature in order to prevent irrecoverable errors.

Grounding Extension 1

An obvious solution for modeling grounding would be to add a backward-transition from every state where input is expected, pointing to the state before it, where the corresponding question is to be produced. This new edge should then be taken when the input given by the user is not understood. Here's the general picture:

Adding a transition



The resulting dialogue would be as follows:

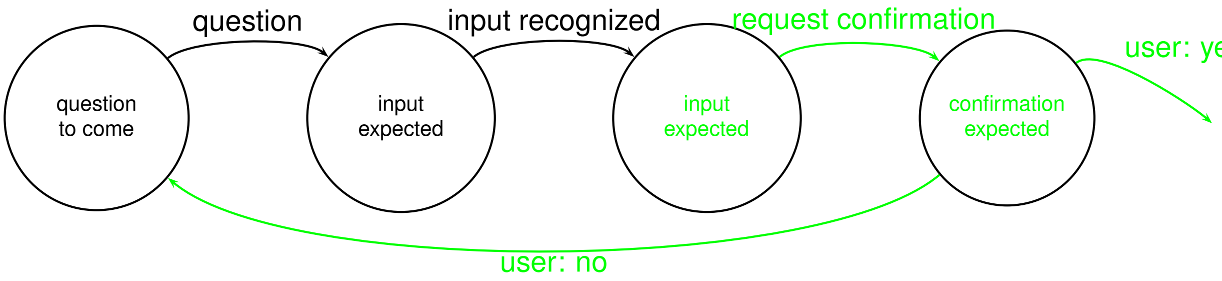
User:	Elevator.
System:	Which floor do you want?
User:	Professor Bill Barry.
(input unrecognised)	
System:	Which floor do you want?
User:	Professor Bill Barry.
(input unrecognised)	
System:	Which floor do you want?
...	...

This is a likely result, as the user cannot know why the failure to recognise the input occurred in the first place.

Grounding Extension 2

The grounding method we've just seen covers (in a very trivial way) the case that the system doesn't recognize the user's input at all. Yet even if the system does recognize the user's input, grounding is important to make sure that the input has been understood correctly. For this, one solution is to add extra states to every state that expects input. Those states are responsible for asking for explicit or implicit confirmation based on what the proposed recognition is.

Adding states



The resulting dialogues would be as follows:

User:	Elevator.
System:	Which floor do you want?
User:	Professor Bill Barry.
System:	Did you say professor Barry?
User:	Yes
System:	[Trip...]

Or:

User:	Elevator.
System:	Which floor do you want?
User:	Professor Bill Barry.
System:	Did you say professor Barry?
User:	No
System:	Which floor do you want?

It is of course quite aggravating for the user to have to confirm what he has said every time he says something. That would amount to almost double the number of turns that humans need to hold a dialogue with the same outcome. If the number of the original questions that the system needs to ask is more than a couple, which is true for most domains, this becomes a significant problem.

Grounding Extension 3

Delayed Confirmation

A third option for systems that request a lot of information from the user is to have delayed implicit requests for confirmation. That means adding for some states other states that asks the user to confirm all the information that the system has elicited in the previous states. This method avoids prolonging dialogues too much.

A dialogue with such a system might look as follows:

User: Elevator.
System: Please specify the city for which you want a weather forecast.
User: SaarbrÃ$\frac{1}{4}$cken.
System: OK, For when?
User: Tomorrow afternoon.
System: So, you want a weather forecast for SaarbrÃ$\frac{1}{4}$cken for tomorrow afternoon.
User: Okay.
System: [...]

Or:

User: Elevator.
System: Please specify the city for which you want a weather forecast.
User: SaarbrÃ$\frac{1}{4}$cken.
System: OK, For when?
User: Tomorrow afternoon.
System: So, you want a weather forecast for Stuttgart for tomorrow afternoon.
User: No, that's wrong.
System: Please specify the city for which you want a weather forecast.

The system first collects some user input and then requires confirmation for all the collected information at once. This method is already quite advanced, since it requires some kind of way for storing all of the previous answers recognised before the confirmation is done.

And what about Repair?

The last two examples involve defining a lot of states and transitions. Yet note again that they do not take into account the case where the input is not understood at all. In order to do that, we could first implement the first method we saw in Section 4.2.4. That is, adding a transition to return to the state before the corresponding question if the input is not understood.

Even this combination of methods would not allow the user to correct the input directly after the request for confirmation. If the information gathered by the system is inadequate, the user has to go through answering all questions again. Instead it would be nice to allow for more targeted corrections by the user, for example corrections like:

...
System: So, you want a weather forecast for Stuttgart for tomorrow afternoon.
User: No, I said SaarbrÃ$\frac{1}{4}$cken.
System: So, you want a weather forecast for SaarbrÃ$\frac{1}{4}$cken for tomorrow afternoon.

Or:

...
System: So, you want a weather forecast for SaarbrÃ¼cken for tomorrow morning.
User: Wrong time.
System: So, you want a weather forecast for SaarbrÃ¼cken for tomorrow afternoon.
System: OK, For when?
...

To allow for this kind of dialogues, we have to add still more states and transitions, to distinguish between different kinds of user-reactions to the system's confirmation request and repeat different questions based on what recognition errors the user points out.

?- Question!

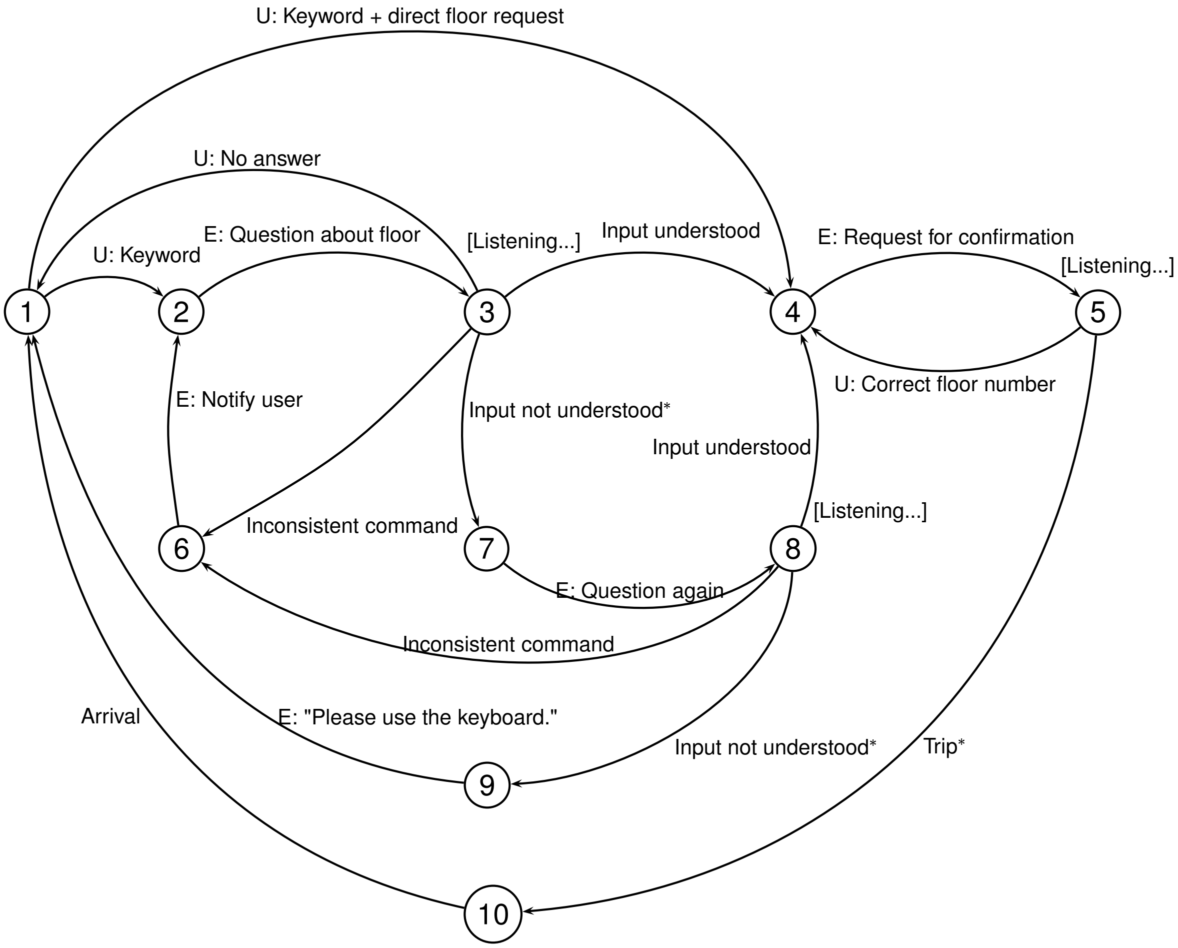
What about including insertions, for example clarification sub-dialogues, and still allowing for grounding within them in the manner described above?

4.3 An in-depth look at the speaking elevator

4.3.1 The SaarbrÃ¼cken CL department's Speaking Elevator

Now let's have a closer look at the dialogue engine of the SaarbrÃ¼cken CL department's speaking elevator. In the following we will assume that the automaton at the heart of the speaking elevator looks like this:

Elevator dialogue finite state automaton



?- Question!

Compare this automaton to the simple one in Section 4.2.2. Do you recognize the additions that correspond to the grounding techniques we've just discussed?

See movie in HTML version.

Try different trips interactively!

4.3.2 How does the dialogue progress?

The initiation of the dialogue occurs via a keyword, namely, either ‘Fahrstuhl’ or ‘Aufzug’, that the user has to be aware of and utter. There is also the possibility to skip the second and third state and go directly to state 4, if the user uses the keyword and makes a travel request at the same time. This possibility is defined separately from the single keywords in the recognition grammar. It is only an added feature for quicker interaction for the people familiar with the system.

When one of the keywords is recognised, the elevator asks in German ‘Which floor do you want to go to?’, which corresponds to transition 2–>3 in the model. By that, the system is set to the state of expecting an answer, which corresponds to state 3. The answers are all in the recognition grammar of the speech recognizer and they include numbers of floors as well as descriptions that can be mapped onto floor numbers, e.g. names of professors that sit on that floor. Turns are strictly defined: The elevator prompts for information, the user answers, the elevator prompts again, and so on.

4.3.3 An additional memory

There is a memory that is used for storing global variable values, which are accessible from all states. One thing such variables do is providing a mechanism to have *parametrized* actions connected to edges instead of fully instantiated ones. This extends the possibility of things that can happen over one and the same transition from one state to the next. So that way, the memory allows defining less states explicitly, as the actual arguments are defined via global variables instead of states. For instance we will see that there is no need to add explicit states for every different target floor in order to produce correct confirmation sentences. The floor requested is looked up in the memory and is added to the standard confirmation utterance.

The memory is also used for various other purposes that we will soon come across. It is important for the time management. The request of the selection function to the speech recogniser must be assigned a point in time when a relevant utterance from the user is to be expected. Relevant here means, an utterance that matches one of the arguments that the selection function is expecting. And the memory also adds dialogue context by keeping track of already recognised targets, which floors the elevator has been on, and the current floor. The global memory is reset when the automaton reaches the starting state again at the end of a trip or when it has given up understanding the user.

4.3.4 How long does the system wait for a response?

The system waits for responses for a pre-defined specific amount of time (this happens at states 3, 5, and 8. We have indicated such ‘time windows’ in the diagram by writing [Listening...]). If there is no answer within the time limit at a state with a ‘time window’, the system carries out a specified default action. We indicate default actions by marking the corresponding edge label with a little asterisk *.

So for instance at state 3 the system waits for user input for some time. If, however, there is no input within that time, the system goes to state 7 because that is the default action. From there it moves to state 8 and asks for a floor specification again. If there is no response in due time after that second prompt in a row, the elevator asks the user to use the keyboard and it is reset to the starting point.

4.3.5 How does the elevator deal with unrecognised utterances or inconsistent input?

If the user utterance is not understood, a transition is chosen which corresponds to the elevator stating so, ‘I have not understood.’. If the second try by the user is still not understood, there is a request that the user uses the buttons instead, and the elevator is reset. (Since the *Input not understood*-edges are marked as default, this is of course exactly the same course of things that we just discussed for time-outs at the ‘time windows’ at states 3 and 8).

Due to its global memory, the elevator also has a modest possibilty to check the (understood) user input for consistency. It compares the floor number that it understood from the request to that of the floor it’s on, which it finds in its memory. If these are the same, the automaton moves to state 6, then to back to state 2 and informs the user: ‘We are already on floor x’. Being in state 2, the elevator asks for instructions again.

4.3.6 And what happens if the elevator does understand the input?

If however the user’s utterance is understood at state 3 (that is, if a floor number is understood that makes sense as destination floor), the elevator moves to state 4. From there a transition follows for the confirmation of the floor understood. After this, the elevator is in state 5. If then the user corrects the elevator, we are back at state 4 (confirmation to come). This can end up in an infinite loop.

?- Question!

Can you trace the loop in the dialogue model in figure ((page 59))?

4.3.7 How is the user input confirmed?

User input is confirmed indirectly. The elevator informs the user of the floor it is about to start traveling to. If there is no correction or if the input is ‘Thank you’, the elevator starts the trip.

4.3.8 During the trip...

The dialogue cannot continue during or after the trip to the requested floor. It has to start from state 1 again.

When the trip starts, the automaton moves to state 10 and waits until the mechanical command has been executed. The user is then informed that they have arrived at the particular floor (transition 10–>1). This transition resets the automaton to the starting state.

4.3.9 What happens when there is ambiguous input?

The SaarbrÃ$\frac{1}{4}$cken Phonetics department is in the same building as the CL department. They use the same elevator, and because they are on floors 4 and 5, they probably use it quite often. Now someone might ask the elevator: ‘Take me to the phonetics department!’ If there is such an ambiguous request, the elevator should ask a clarification question and be reset to state 3. For example, the elevator should ask ‘Do you want the 4th or the 5th floor?’.

?- Question!

How would our automaton have to be extended to deal with this situation?

4.3.10 Speaking elevator reference

- Download here.[2]

4.4 Examples of required behaviour

4.4.1 Turn-taking

System:	Which floor do you want?
User:	(no input)
System:	I'm sorry I didn't understand you. Please specify a floor number, a person or a department you

The elevator prompts in a more elaborate way when there is no response by the user after a pre-defined amount of time. It gives information on what kind of answer is expected.

Current state and possibility of extension: See Section 4.5.2. See also description of elevator, Section 4.3.

4.4.2 Adjacency pairs and insertions

System:	Which floor do you want?
User:	Which floor is Malte Gabsdil on?
System:	Malte Gabsdil is on the first floor.
User:	Then, first floor, please.

The user requires more information in order to decide which floor they want to be taken to. The elevator provides it. When the user makes their mind up, based on the additional information, the elevator is able to understand that the last utterance is a response to the first question.

Current state and possibility of extension: See Section 4.5.2. See also description of elevator, Section 4.3.

4.4.3 Grounding

System:	Which floor do you want?
User:	Third floor.
System:	I'm taking you to the fourth floor.
User:	No, third floor.
System:	I'm taking you to the third floor.

The elevator misrecognises the requested floor. The user corrects it and the elevator takes the correction correctly into account.

Current state and possibility of extension: See Section 4.5.3. See also description of elevator, Section 4.3.

---

[2] `speaking-elevator.ps.gz`

4.4.4 Dialogue context

(user gets in on third floor)
System: Which floor do you want?
User: Ground floor.
System: I'm taking you to the ground floor.
(trip starts)
User: I left my bag in the office. Can we go back?
System: Yes. I'm taking you to the third floor.

The user specifies a target floor. After the trip has started he says that he wants to go 'back'. The system interprets 'back' in context and understands that the user wants to go to the third floor, where the trip started from.

Current state and possibility of extension: See Section 4.5.4. See also description of elevator, Section 4.3.

4.4.5 Ellipsis

System: Which floor do you want?
User: Fifth.
System: I'm taking you to the fifth floor.

The user gives a fragmentary response. Instead of saying 'I want to go to the fifth floor.', he says 'fifth'. The elvator still understands that the fifth floor is where the user wants to go.

Current state and possibility of extension: See Section 4.5.5. See also description of elevator, Section 4.3.

4.4.6 Reference resolution

System: Which floor do you want?
User: The top floor.
System: I'm taking you to the 5th floor.

The elevator does not give a floor number but a referring expression for it. The elevator is able to resolve the referring expression and realise that 'The top floor' is the fifth floor.

System: Which floor do you want?
User: Phonetics
System: Do you want the 4th or the 5th floor?
User: The fifth floor, please.

The same here. The system resolves 'Phonetics' to be referring to either the fourth or the fifth floor. Being aware of the ambiguity, it requests a clarification.

Current state and possibility of extension: See Section 4.5.6. See also description of elevator, Section 4.3.

4.4.7 Mixed initiative

User: Third floor quickly!
System: I'm taking you to the third floor.

The user should be able to initiate the dialogue in the way they choose. In this example no keyword has been used, but the elevator still responds appropriately.

System: Which floor do you want?
User: I want to go to the second floor to professor Pinkal's office.
System: Professor Pinkal's office is on the first floor. Shall I take you to the first floor?
User: Yes.

The user makes an inconsistent request. The elevator recognises that, lets the user know, and offers an alternative.

Current state and possibility of extension: See Section 4.5.7. See also description of elevator, Section 4.3.

4.5 Dialogue characteristic and extension of the elevator

4.5.1 Turn-taking

Turn-taking is strict. It has to be explicitly defined in states. The system asks the user something and waits for the user's contribution for a defined amount of time. An empty turn, where the user does not say anything, or not within the time limit, is also defined as a turn.

4.5.2 Adjacency pairs and insertions

Insertion dialogues and adjacency pairs have to be defined explicitly as possible transitions from every state. This would lead to computational explosion! Try to work out what the dialogue model would look like if you wanted to add some clarification dialogues.

4.5.3 Grounding

Grounding in the speaking elevator is *implicit*. Traversing the edge 4->5, the floor requested is looked up in the system's globl memory and is then mentioned in a standard confirmation utterance: 'I'm taking you to the x-th floor.', which the user can then object to.

As we've discussed above, by using an extra global memory to store the floor requested we can keep the number of states and edges needed for grounding quite small: There is no need to add explicit states for every different target floor in order to produce the correct confirmation sentence. A nice extension would be to remember the floor that was misunderstood, so that it does not recognise it wrongly twice, that is, if the user has already stated that the original recognition was wrong.

Repair is also possible, although in a very limited fashion in the depicted dialogue model. It is defined as a possible transition from the state after the confirmation utterance has been produced (5->4). The elevator can only recognize correct floor requests over this transition. Everything else activates the *Trip* edge. That is, you can only correct it by giving alternative (and consistent) floor requests, not by saying for instance ‘No!’ or ‘Stop!’.

?- Question!

How would the speaking elevator-automaton have to be changed to allow for more natural corrections by the user?

4.5.4 Dialogue context

Almost all dialogue context is hard-coded in the states and transitions. It is not explicitly represented, with the exception of storing the floor requested and the floor currently on. That makes it difficult to change general aspects of the dialogue. All states have to be re-defined.

4.5.5 Ellipsis

Ellipsis can be dealt with only in the very particular context of the current state. Recognition is based on keyword spotting. So, as long as elliptical phrases are present in the recognition grammar, they will be recognised. The recognition grammar is the same for every state. Therefore, the elevator cannot be extended by allowing a different grammar to be called for every state. There are other similar systems that do make use of that possibility.

4.5.6 Reference resolution

There is no possibility of extending the elevator to handle reference resolution apart from explicitly including the referring expressions in the recognition grammar. That can only be done for a tiny amount of referring expressions out of the, in principle, unlimited possible ones.

4.5.7 Mixed initiative

Only the system has initiative, apart from the initialisation of the interaction. Still that can only be done by one of the pre-specified keywords. If something goes wrong, the system is reset and the dialogue has to start anew, with loss of all information. The system prompts the user for a specific kind of contribution and waits for a relevant utterance in the very limited context of the current state. The user cannot provide more information that could be helpful, or underspecify a parameter required by the system, as only the utterances defined in the recognition grammar can be recognised. Users cannot even choose to answer questions still in the pre-defined way, but in an order that better fits their purpose. In addition, the system cannot reason about anything that is not hard-coded. It cannot make suggestions based on context, for example.

There is no possibility of extending the elevator for mixed initiative other than defining states that allow the system to take further initiative itself. An extension to accept user initiative can also partially be handled in the same way. It is, however, more problematic as there is no upper limit as to what the user can say, or request. This is a general problem with allowing a high degree of user initiative - even sophisticated plan recognition techniques are faced with it.

A fairly easy extension would be for the system to inform the user of the options they have at each point, so that the dialogue does not come to a break. In a way, this is modeled now by the system telling the user to use the keyboard after two misrecognitions in a row.

4.6 Summary and Outlook

4.6.1 Advantages and disadvantages

Dialogue structure

Dialogue management with FSA is simplified. That means that they are easy and quick to develop, as long as the domain is well-structured. The developer can define the states and transitions necessary based on the structure already present in the domain. This also presupposes that only the system has the initiative, because as soon as the user is allowed to take the initiative the input and structure cannot be controlled. Moreover, when there are more than just the basic dialogue requirements for a domain, FSA become an effort-some and time-consuming dialogue modeling method that is moreover not very robust. The developer has to hard-code every single behaviour while at the same time he always runs the risk of leaving something out, which can eventually cause the system to break down. In other words, there is no clear way of capturing a general conceptualization that makes a system less bug-prompt.

User input

Another advantage exemplified by FSA is that speech recognition and interpretation are simplified, because of the predefined user input. As far as the latter is concerned, keyword spotting is enough. That means that only the important semantic entities in the user's answer need to be recognised and interpreted. That, however, amounts to the restriction of the input the user is supposed to give in a very unnatural mode and any information based on structure of the input is lost. Moreover, the user cannot give more information than what has been asked for explicitly each time, since there is no handling of over-answering.

Conclusion

The above drawbacks make dialogue modeling with FSA appropriate only for very simple domains with flat structures. Any domain that involves possible sub-tasks, and especially in an unpredictable order, would require a lot of expensive backtracking in order to recover the state before the sub-task was initiated.

Summing up

In summary, the *advantages* of FSA are:

- Quick and easy to develop
- Controlled user input
- Simple speech recognition
- Simple interpretation
- Appropriate for well-structured domains

The *disadvantages* of FSA are:

- Need to hard code dialogue specifications
- Need to predict dialogue possibilities
- Inflexible structure
- Prompt to break-downs
- Inappropriate for more complicated domains

4.6.2 The CSLU tool

One rather practical advantage of finite state techniques for dialogue processing is that there are various tools for designing dialogue automata that are readily available. If you are keen on finding out more about FSA designing tools, here[3] is the link to the CSLU tool webpage. That includes information on the toolkit and tutorials on use. The tool runs only on Windows.

4.6.3 Beyond finite state techniques

As we've just seen, using finite state techniques in dialogue processing may lead to good results with relatively little effort, but only under certain conditions. If for instance great flexibility in a complicated domains is what is needed, finite state techniques often don't lead to satisfying solutions.

In such cases, one uses methods that are more sophisticated linguistically as well as technically. Such methods often involve a fair amount of *reasoning*, for instance about plans and communicative intentions. A key idea is that of representing the user's and the system's knowledge state explicitely and employ reasoning techniques on these state representations. Fully-fledged reasoning also allows for the consideration of domain knowldege to the extent needed for solving even quite complex tasks. Another concept of great importance in designing advanced dialogue systems is that of *speech act* s or *dialogue act* s, defining what role a each utterance plays within a dialogue.

Two examples of quite ambitious dialogue systems are the TRAINS system and its successor, the TRIPS system. Both deal with helping humans in logistics tasks. The basic dialogue scenario that TRAINS deals with is route-planning in a transportation domain. TRIPS collaborates with the user in disaster relief management.

---

[3] `http://cslu.cse.ogi.edu/toolkit/docs/index.html`

A TRAINS example dialogue

Here’s an example of a dialogue with the trains system:

User[1]: I need to get to Detroit, Syracuse and Montreal.
Syst[2]: Ok. [highlights the three cities]
User[3]: Let’s send the trains from Chicago to Detroit.
Syst[4]: Ok. [draws route from Chicago to Detroit via Toledo] But the terminal at Toledo is delaying
User[5]: That’s ok. Now let’s go from Atlanta up to Syracuse.
Syst[6]: Please tell me a route to use to get from Atlanta to Syracuse [highlights engine at Atlanta and
User[7]: Go via Charleston and Pittsburgh.
Syst[8]: Ok. [draws route via Cincinnati] But traffic through Cincinnati is being delayed due to constr
User[9]: Ok, let’s go via Charleston and Buffalo instead.
Syst[10]: No problem. [draws route]
User[11]: That’s great.
Syst[12]: I hope it was helpful.
User[13]: Finally, I want to go from Montreal to Philadelphia.
Syst[14]: Ok. [draws route]
User[15]: That’s great. I’m done.

As you can see from this small dialogue already, the TRAINS system allows for much more flexible interaction than for instance the speaking elevator. Dialogues start by either the system presenting the user with a problem, or the user presenting the system with their goal. The participants then have to define tasks in order to solve the original problem and talk about them. Interactions are interpreted with reference to all previous interactions.

TRAINS was built with the objective of allowing the user and the system to collaborate towards solving the problem in a way that humans collaborate with each other. Each participant is responsible for the part of the task that they can perform better. That means that the user is responsible for the top level goals of how to attack a problem. The system constantly tries to infer from the user’s input what the user’s goals are, with respect to the task. Based on that, it draws the user’s attention to possible problems and makes relevant suggestions. Due to the aimed behaviour for the system, a lot of planning and plan recognition becomes necessary. The TRAINS system uses domain-specific reasoning techniques to make plan recognition less computationally expensive. As far as planning is concerned, the system is deliberately weak, so that more collaboration with the human user is provoked.

Demo videos

This website[4] has further information (including demo videos with extensive dialogues) on the TRAINS and TRIPS systems.

4.7 Exercises

Exercise 4.1 *Watch the TRIPS/TRAINS-demo videos referenced in Section 4.6.3. Use the criteria given in Section 4.1.2 to evaluate the systems based on these videos and the example dialogue in the same section.*

---

[4] `http://www.cs.rochester.edu/research/cisd/projects/trains/tripscompare.html`

Exercise 4.2 *The Philips train information system is a system that gives information on train travels. In this exercise you're asked to experiment with this system.*

1. *Dial the following phone number to be connected with the system:*

   ```
   0241 604020
   ```

   We suggest you do that in groups! Protocol your dialogues.

2. *Try to systematically evaluate the system with regard to the characteristics discussed in Section 4.1.2 and the desired behaviour you would like it to depict. Take notes of your results.*

3. *Which of the dialogue systems that we have looked into is it closer to; the speaking elevator or the TRAINS system? Justify your opinion by specific examples, where possible.*

4. *Try if you can draw an automaton that handles some or even all of your protocoled dialogues.*

Exercise 4.3 *Give a Prolog-specification of the simple dialogue automaton shown in Section 4.2.2, using the following extensions to our ususal representation scheme:*

- *Use* `write('blabla blubb')` *as label for edges meant to produce the output 'blabla blubb'.*
- *Use* `read('blabla blubb')` *as label for edges meant to (possibly) ask for input and then to be traversed if the input was 'blabla blubb'.*
- *Use* `call(trip(FloorNumber))` *as label for edges meant to trigger an action named* `trip(FloorNumber)`.

For this exercise you can assume that the edge labeled 'Input understood' in the picture somehow sets a global variable named `FloorNumber` *that can then be accessed by all edges traversed later on.*

How would you describe the language accepted by the FSA you've specified? (Just look at it as if it was one of the usual FSAs that you know from the previous lectures. Can you give a regular expression for it?).

Describe what extensions to the predicates in See file `recognize.pl`. *become neccessary in order to use them for produce a dialogue based on the automaton you've specified.*

Exercise 4.4 *[Mid-term project]*

Adapt the predicates in See file `recognize.pl`. *so that they can process specifications of dialogue automata as described above in !!!UNEXPECTED PTR TO EX_DIALOGUE.EX.SPECIFY!!!. You can use the predicate* `readAtom/1` *from the file See file* `readAtom.pl`. *to read input from the keyboard into an atom.*

See file `readAtom.pl`. *Predicate* `readAtom/1` *for reading keyboard input into an atom.*

Here's an example of what this predicate does:

```
2 ?- readAtom(A).
|: hallo spencer.

A = 'hallo spencer.'

Yes
3 ?-
```

Include the following code before your program to be able to use the predicate:

```
:- use_module(readAtom,[readAtom/1]).
```

Please let us know[5] *if you would like to do this exercise as your mid-term project.*

---

[5] `mailto:stwa@coli.uni-sb.de`

5

Context Free Grammars

5.1 Context Free Grammars

5.1.1 The Basics

context free grammar s (CFGs) describe a larger class of formal languages than regular languages do. As we will see below, the language $a^n b^n$ can in fact be described by a CFG. Even more, many natural language structures can be neatly described with context free grammars. This makes them especially useful for writing grammars for NLP applications. Another reason why CFGs are widely used in the field of NLP is the availability of many efficient tools (compilers, parsers, etc.).

Here's a simple CFG:

$$
\begin{array}{lcl}
S & \longrightarrow & A\ B \\
S & \longrightarrow & A\ S\ B \\
A & \longrightarrow & a \\
B & \longrightarrow & b
\end{array}
$$

Some Terminology

The ⟶ is called the *rewrite arrow* , and the four expressions listed above are called *context free rule* s (you may also see them called *rewrite rule* s, or something similar).

Apart from the ⟶, there are two sorts of symbols in this grammar: *terminal symbol* s and *non-terminal symbol* s. In the above grammar, the terminal symbols are ‘a’ and ‘b’, and the non-terminal symbols are ‘S’, ‘A’, and ‘B’.

Terminal symbols never occur to the left of a rewrite arrow.

Non-terminal symbols may occur either to the right or the left of the rewrite arrow (for example, the ‘S’ in the above grammar occurs both to the right and the left of ⟶ in the second rule).

Every context free grammar has one special symbol, the *start symbol* (or *sentence symbol*), which is usually written ‘S’. Some context free grammars also use another special symbol, namely ε, for the empty string. The ε symbol can only occur on the right hand side of context free rules. For example, we could add the rule $S \longrightarrow \varepsilon$ to the above grammar, but we could *not* add $\varepsilon \longrightarrow A$.

Rewrite Interpretation

The simplest interpretation of context free grammars is the *rewrite interpretation* . This views CFGs as rules which let us rewrite the start symbol to other strings: each rule is interpreted as saying that we can replace an occurrence of the symbol on the left side of the rule by the symbol(s) on the right side.

For example, the above grammar lets us rewrite ‘S’ to ‘aabb’:

S	
ASB	Rule 2
aSB	Rule 3
aSb	Rule 4
aABb	Rule 1
aaBb	Rule 3
aabb	Rule 4

Such a sequence is called a *derivation* of the symbols in the last row. For example, the above sequence is a derivation of the string ‘aabb’. Note that there may be many derivations of the same string. For example,

S	
ASB	Rule 2
ASb	Rule 4
aSb	Rule 3
aABb	Rule 1
aAbb	Rule 4
aabb	Rule 3

is another derivation of ‘aabb’.

Why are such grammars called ‘context free’? *Because all rules contain only one symbol on the left hand side — and wherever we see that symbol while doing a derivation, we are free to replace it with the stuff on the right hand side.* That is, the ‘context’ in which a symbol on the left hand side of a rule occurs is unimportant — we can *always* use the rule to make the rewrite while doing a derivation. [1]

The *language generated by a context free grammar* is the set of terminal symbols that can be derived starting from the start symbol ‘S’. For example, the above grammar generates the language $a^n b^n \backslash \{\varepsilon\}$ (the language consisting of all strings consisting of a block of ’a’s followed by a block of ’b’s of equal length, except the empty string). And if we added the rule $S \longrightarrow \varepsilon$ to this grammar we would generate the language $a^n b^n$.

A language is called *context free* if it is generated by some context free grammar. For example, the language $a^n b^n \backslash \{\varepsilon\}$ is context free, and so is the language $a^n b^n$, for we have just seen that these can be produced by CFGs. Not all languages are context free. For example, $a^n b^n c^n$ is not. That is, no matter how hard you try to find CFG rules that generate this language, you won’t succeed. No CFG can do the job.

---

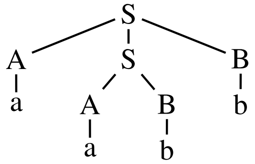
[1] There are more complex kinds of grammars, with more than one symbol on the left hand side of the rewrite arrow, in which the symbols to the right and left have to be taken into account before making a rewrite. Such grammars can be linguistically important, but we won’t study them in this course.

5.1.2 The Tree Admissibility Interpretation

While the rewrite interpretation of CFGs is important, there is a more fundamental way of thinking about CFGs, namely as *tree admissibility rule* s.

A *parse tree* is a finite tree all of whose interior nodes (that is, nodes with daughters) are *licensed (admitted) by a grammar rule*. What does this mean? Let's look at an example:

A Parse Tree



This tree is licensed by our grammar — that is, the presence of every node with daughters can be justified by some rule. For example. the top node is ok, because under the tree admissibility interpretation we read $S \longrightarrow A\ S\ B$ as saying: *A tree node labelled S can have exactly three daughters, labelled (reading left to right) A, S, and B.* The node labelled A is ok, because under the tree admissibility interpretation we read the rule $A \longrightarrow a$ as saying: *A tree node labelled A can have exactly one daughter which is labelled a.* Similarly, all the other nodes with daughters are 'justified' or 'admitted' by other rules (the terminal nodes of the tree don't need to be justified).

If you think about it, you will see that there is a close correspondence between parse trees and derivations: every derivation corresponds to a parse tree, and every parse tree corresponds to (maybe many) derivations. For example, the tree above corresponds to both our earlier derivations of 'aabb'.

The tree admissibility interpretation is important for two reasons. First, it is the linguistically most fundamental interpretation. We don't think of natural language as a bunch of strings that need rewriting — we think of natural language sentences, and noun phrases and so on as having structure, tree structure. Thus we think of CFG rules as telling us which tree structures we have.

Ambiguity

Second, the idea of parse trees brings us to an important concept: *ambiguity* . *A string is ambiguous if it has two distinct parse trees.* Note: We said 'parse trees', not 'derivations'. For example, we derived the string 'aabb' in two different ways from our little grammar — but both ways correspond to the same derivation tree. That is, the string 'aabb' is *not* ambiguous in this grammar. And in fact, no string this grammar generates is ambiguous. But we will see that some grammars *do* generate ambiguous strings.

5.1.3 A Little Grammar of English

The above ideas adapt straightforwardly to natural languages. But with natural languages it is useful to draw a distinction between rules which have syntactic categories on the right hand side, and rules which have only words on the right hand side. The rules are known as *phrase structure rule* s, and *lexical rule* s, respectively.

A Phrase Structure Grammar

Here’s a simple so-called *phrase structure grammar* (PSG) of English. We have phrase structure rules:

$$\begin{array}{l}
S \longrightarrow NP\,VP \\
NP \longrightarrow PN \\
NP \longrightarrow PN\ Rel \\
NP \longrightarrow Det\ Nbar \\
NBar \longrightarrow N \\
NBar \longrightarrow N\ Rel \\
Rel \longrightarrow Wh\,VP \\
VP \longrightarrow IV \\
VP \longrightarrow TV\ NP \\
VP \longrightarrow DV\ NP\ PP \\
VP \longrightarrow SV\ S \\
PP \longrightarrow P\ NP
\end{array}$$

...and lexical rules:

$$\begin{array}{l}
PN \longrightarrow vincent \\
PN \longrightarrow mia \\
PN \longrightarrow marsellus \\
PN \longrightarrow jules \\
Det \longrightarrow a \\
Det \longrightarrow the \\
Det \longrightarrow her \\
Det \longrightarrow his \\
N \longrightarrow gun \\
N \longrightarrow robber \\
N \longrightarrow man \\
N \longrightarrow woman \\
Wh \longrightarrow who \\
Wh \longrightarrow that \\
P \longrightarrow to \\
IV \longrightarrow died \\
IV \longrightarrow fell \\
TV \longrightarrow loved \\
TV \longrightarrow shot \\
TV \longrightarrow knew \\
DV \longrightarrow gave \\
DV \longrightarrow handed \\
SV \longrightarrow knew \\
SV \longrightarrow believed
\end{array}$$

So in this grammar, the terminal symbols are English words. There is a special word for the symbols (such as N, PN, and Det) which occur in lexical rules: they are called *preterminal symbol* s.

This grammar is unambiguous. That is no string has two distinct parse trees. (Incidentally, this means that it is not a realistic grammar of English: all natural languages are highly ambiguous.) But it does display an interesting (and troublesome) phenomenon called *local ambiguity* .

Local Ambiguity

Consider the sentence ‘The robber knew Vincent shot Marsellus’. This has a unique parse tree in this grammar. But now look at the first part of it: ‘The robber knew Vincent’. This is also a sentence according to this grammar — but *not* when considered as a part of ‘The robber knew Vincent shot Marsellus’. This can be a problem for a parser. Locally we have something that looks like a sentence — and a parser may prove that this part really is a sentence according to the grammar. But this information does not play a role in analyzing the larger sentence of which it is a part. Keep this in mind. It will become important again when we build a parser using this grammar in Section 6.1.

5.2 DCGs

5.2.1 DCGs are a natural notation for context free grammars

Suppose we are working with a context free grammar, for example, this:

S ⟶ NP VP
NP ⟶ Det N
VP ⟶ V NP
Det ⟶ the
Det ⟶ a
N ⟶ witch
N ⟶ wizard
V ⟶ curses

We can immediately turn this into the following DCG (found in See file `dCGExample.pl`.):

```
s --> np,vp.


np --> det,n.


vp --> v,np.


det --> [the].


det --> [a].


n --> [witch].


n --> [wizard].


v --> [curses].
```

The link between the two formalisms is obvious. About the only thing you have to remember is that in DCG notation, lexical items (that is, the words) have to be written as elements of a one-element list.

What's not quite so obvious is the way DCGs are used. For example, to see whether 'A witch curses the wizard' is accepted by the grammar we need to pose the following query: `s([a,witch,curses,the,wizard],[]).`

That is, we have to give *two* arguments, one of which is the empty list. Similarly, to see if 'the witch' is a noun phrase in this grammar, we would pose the query `np([the,witch],[]).`

DCGs can be used to generate (a very useful property). For example, to see all the sentences generated by our little grammar, we would give the following query: `s(X,[]),write(X),nl,fail.`

To see all the noun phrases produced by the grammar we would pose the query `np(X,[]),write(X),nl,fa`

5.2.2 DCGs are really Syntactic Sugar for Difference Lists

[Background]

Why the two arguments when we make queries? Because DCGs are really "syntactic sugar" for something else. They are a nice user friendly notation for grammars written in terms of difference lists. For example, when the above grammar is given to Prolog, the Prolog interpreter will immediately translate the first rule into something like this:

```
s(A,B) :-
    np(A,C),
    vp(C,B).
```

The lexical rule for 'the' will translate into something like this

```
det([the|W],W).
```

This is not the place to go into how this works, but we will make two remarks: First, the difference list representation of grammars, while a bit complicated at first sight, is a very efficient way of representing grammars. In particular, the use of difference lists allows us to avoid using the `append/3` predicate, thus DCGs are a useful way of writing even quite large grammars. Second, notice that the translated clauses all have *two* arguments — which explains why the above queries needed two arguments.

Summing up: DCG notation is a natural notation that lets us write context free grammars in a natural way. DCGs translate into a difference list representation that allows far more efficient processing than a naive single-list representation that relies on using `append/3`.

5.2.3 DCGs give us a Natural Notation for Features

But we can do more with DCGs. For a start, we are able to add extra arguments to our DCG rules. And this lets us do some linguistically useful things — and in particular it lets us use *feature* s.

Suppose we wanted to deal with sentences like ‘She shoots him’, and ‘He shoots her’. What should we do? Well, obviously we should add rules saying that ‘he’, ‘she’, ‘him’, and ‘her’ are pronouns:

```
pro --> [he].
pro --> [she].
pro --> [him].
pro --> [her].
```

Furthermore, we should add a rule saying that noun phrases can be pronouns:

```
np --> pro.
```

This new DCG (kind of) works. For example:

```
s([she,shoots,him],[]).
yes
```

But there are obvious problems. The DCG will also accept a lot of sentences that are clearly wrong, such as ‘A witch curses she’, ‘Her curses a wizard’, and ‘Her shoots she’.

Here’s a *bad* way of repairing the problem: add more rules. For example, we might rewrite the DCG as follows:

```
s --> np_subject,vp.
np_subject --> det,n.
np_object  --> det,n.
np_subject --> pro_subject.
np_object  --> pro_object.
vp --> v,np_object.

vp --> v.
det --> [the].
det --> [a].
n --> [witch].
n --> [wizard].
pro_subject --> [he].
pro_subject --> [she].
pro_object --> [him].
pro_object --> [her].
v --> [curses].
```

This is awful. Basically, we’ve had to make a big change to the grammar to cope with a very small set of facts. After all, let’s face it: ‘I’ and ‘me’ are pretty much the same — they just differ with respect to their case property and the way they sound. By marking information with features, we can do a much neater job:

```
s --> np(subject),vp.
np(_) --> det,n.
np(X) --> pro(X).
vp --> v,np(object).
vp --> v.

det --> [the].
det --> [a].
n --> [witch].
n --> [wizard].
v --> [curse].
pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

The extra argument — the feature — is simply passed up the tree by ordinary unification. And, depending on whether it can correctly unify or not, this feature controls the facts of English case neatly and simply.

Summing up: features let us get rid of lots of unnecessary rules in a natural way. And DCGs enable us to implement rules with feature information in a natural way.

?- Question!

Above, we said that DCGs are formally more powerful than CFGs. Did the addition of features in our case leave the expressivity of CFGs?

[Background]

One last remark. Note that extra arguments, really are just plain old Prolog arguments. For example

```
s --> np(subject),vp.
```

translates into something like.

```
s(A,B) :-
    np(subject,A,C),
    vp(C,B).
```

5.3 DCGs for Long Distance Dependencies

5.3.1 Relative Clauses

Consider these two English NPs:

‘The witch who Harry likes.’

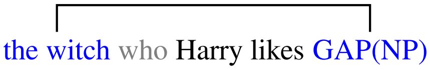
This NP contains a relative clause where the relative pronoun is the direct object. Next, an NP with a relative pronoun in subject position:

‘Harry, who likes the witch.’

What is the syntax of such NPs? That is, how do we build them? We are going to start off by giving a fairly traditional explanation in terms of *movement* , *gap* s, *extraction* , and so on. As we’ll soon see, it’s pretty easy to think of these ideas in terms of features, and to implement them using DCGs.

The Traditional Explanation

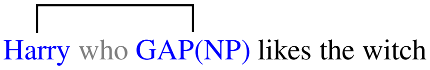
The traditional explanation basically goes like this. We start from the following sentence: ‘Harry likes the witch’ Now we can think of the NP with the object relative clause as follows.



the witch who Harry likes GAP(NP)

That is, we can think of it as being formed from the original sentence by (1) extracting the NP ‘the witch’ from its original position, thereby leaving behind an empty NP, or an NP-gap, and (2) moving it to the front, and (3) inserting relative pronoun ‘who’ between it and the gap-containing sentence.

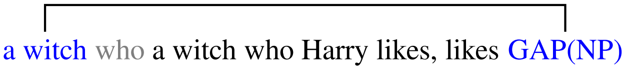
What about the subject relative clause example? We can think of this as follows:



Harry who GAP(NP) likes the witch

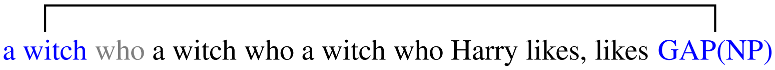
That is, we have (1) extracted the NP ‘Harry’ from the subject position, leaving behind an NP-gap, (2) moved it to the front, and (3) placed the relative pronoun ‘who’ between it and the gap-containing sentence.

But why are relative clause constructions an example of unbounded dependencies? The word ‘dependency’ indicates that the moved NP is linked, or depends on, its original position. The word ‘unbounded’ is there because this “extracting and moving” operation can take place across arbitrary amounts of material. For example, from ‘A witch who Harry likes, likes a witch.’ we can form



a witch who a witch who Harry likes, likes GAP(NP)

And we can iterate such constructions indefinitely — albeit, producing some pretty hard to understand sentences. For example, from: ‘ A witch who a witch who Harry likes, likes a witch.’ we can form



a witch who a witch who a witch who Harry likes, likes GAP(NP)

In what follows we shall see that, by using features, we can give a nice declarative account of the basic facts of English relative clauses. Indeed, we won’t even have to think in terms of movement. We'll just need to think about what information is missing from where, and how to keep track of it.

5.3.2 A First DCG

Let's now sit down and write a DCG for simple English relative clauses. In this DCG (found in See file `dCG4Gaps.pl`.) we will only deal with transitive verbs (verbs that take a single NP as argument — or to use the proper linguistic terminology, verbs that *subcategorize* for a single NP). Further, to keep things simple, the only relative pronoun we shall deal with is 'who'.

Let's first look at the NP rules:

```
np(nogap) --> det, n.
np(nogap) --> det, n, rel.
np(nogap) --> pn.
np(nogap) --> pn, rel.
np(gap(np)) --> [].
```

The first four rules are probably familiar. They say that an English NP can consist of a determiner and a noun (for example: 'a witch'), or a determiner and a noun followed by a relative clause (for example: 'a witch who likes Harry'), or a proper name (for example: 'Harry'), or a proper name followed by a relative clause (for example: 'Harry, who likes a house-elf'). All these NPs are 'complete' or 'ordinary' NPs. Nothing is missing from them. That is why the extra argument of `np` contains the value `nogap`.

What about the fifth rule? This tells us that an NP can also be realized as an empty string — that is, as nothing at all. Obviously this is a special rule: it's the one that lets us introduce gaps. It says: we are free to use 'empty' NPs, but such NPs have to be marked by a feature which says that they are are special. Hence in this rule, the value of the extra argument is `gap(np)`. This tells us that we are dealing with a special NP — one in which the usual NP information is absent.

A Feature-Based Approach

The use of features to keep track of whether or not information is missing, and if so, what kind of information is missing, is the crucial idea underlying grammar-based treatments of all sorts of long distance dependency phenomena, not just relative clauses. Usually such features keep track of a lot more information than our simple `nogap` versus `gap(np)` distinction — but as we are only looking at very simple relative clauses, this is all we'll need for now.

Now for the S and VP rules.

```
s(Gap) --> np(nogap),vp(Gap).

vp(Gap) --> v(1), np(Gap).
```

The first rule says that an S consists of an NP and a VP. Note that the NP must have the feature `nogap`. This simply records the fact that in English the NP in subject position cannot be realized by the empty string (in some languages, for example Italian, this is possible in some circumstances). Moreover, note that the value of the `Gap` variable carried by the VP (which will be either `nogap` or `gap(np)`, depending on whether the VP contains empty NPs) is unified with the value of the `Gap` variable on the S. That is, we have here an example of feature passing: the record of the missing information in the verb phrase (if any) is passed up to the sentential level, so that we have a record of exactly which information is missing in the sentence.

The second rule says that a VP can consist of an ordinary transitive verb together with an NP. Note that instead of using the symbol `tv` for transitive verbs, we use the symbol `v` marked with an extra feature (the `1`). (In the following section we shall introduce a second type of verb, which we will call `v(2)` verbs.) Also, note that this rule also performs feature passing: it passes the value of `Gap` variable up from the NP to the VP. So the VP will know whether the NP carries the value `nogap` or the value `gap(np)`.

Now for the relativization rules:

```
rel --> prorel, s(gap(np)).
rel --> prorel, vp(nogap).
```

The first rule deals with relativization in object position — for example, the clause 'who Harry likes' in 'The witch who Harry likes'. The clause 'who Harry likes' is made up of the relative pronoun 'who' (that is, a `prorel`) followed by 'Harry likes'. What is 'Harry likes'? It's a sentence that is missing its object NP — that is, it is a `s(gap(np))`, which is precisely what the first relativization rule demands.

Incidentally — historically, this sort of analysis, which is due to Gerald Gazdar, is extremely important. Note that the analysis we've just given doesn't talk about moving bits of sentences round. It just talks about missing information, and says which kind of information needs to be shared with the mother category. The link with the movement story should be clear: but the new information-based story is simpler, clearer and more precise.

The second rule deals with relativization in subject position — for example, the clause 'who likes the witch' in 'Harry, who likes the witch'. The clause 'who likes the witch' is made up of the relative pronoun 'who' (that is, a `prorel`) followed by 'likes the witch'. What is 'likes the witch'? Just an ordinary VP — that is to say, a `vp(nogap)` just as the second relativization rule demands.

And that's basically it. We re-use the lexical rules from above (page 75) and add some new ones:

```
n --> [house-elf].
pn --> [harry].
v(1) --> [likes].
v(1) --> [watches].
prorel --> [who].
```

Let's look at some examples. First, let's check that this little DCG handles ordinary sentences:

`s(_,[harry,likes,the,witch],[]).`

Let's now check that we can build relative clauses. First, object position relativization:

```
np(_,[the,witch,who,harry,likes],[]).
```

Now subject position relativization:

```
np(_,[harry,who,likes,the,witch],[]).
```

And of course, there's no need to stop there — we can happily embed such constructions. For example, combining the last two examples we get:

```
np(_,[the,witch,who,harry,who,likes,the,witch,likes],[]).
```

And indeed, we really are correctly handling an *unbounded* construction. For example, we can form:

```
np(_,[a,witch,who,a,witch,who,harry,likes,likes],[]).
```

And we go on to add another level:

```
np(_,[a,witch,who,a,witch,who,a,witch,who,harry,likes,likes,likes],[]).
```

But this is getting hard to understand — so let's simply check that we can make sentences containing relative clauses and then move on:

```
s(_,[a,house-elf,likes,a,house-elf,who,a,witch,likes],[]).
```

5.3.3 A Second DCG

Our first DCG works, but it only covers a tiny fragment of English. So let's try and improve it a bit.

Now, one obvious thing we could do is to add further relative pronouns, like 'that' and 'which', so that we could deal with such sentences as:

- 'The broomstick which was in Harry's room flew down to the lake'.
- 'The castle that the four wizards built is called Hogwarts',
- 'The house-elf that was in the kitchen cried'.

But making this extension is not particularly difficult. The basic point is that 'who' can only be be used with conscious entities (such as wizards, witches, and house-elves), while 'which' has to be used with entities that lack consciousness (such as broomsticks), while 'that' can be used with both. So all we'd need to do is introduce a feature which could take the values `consc` and `unconsc`, mark the nouns in the lexicon appropriately, and then extend the DCG so that this information was taken into account when forming relative clauses. It's a good exercise, but We won't discuss it further.

A far more interesting extension is to add new kinds of verbs — that is, verbs with different subcategorization patterns. In what follows we'll discuss verbs such as 'give' which subcategorize for an NP followed by a PP.

Consider the following sentence:

'The witch gave the house-elf to Harry'.

We can relativize the NP argument of 'give' (that is, the NP 'the house-elf') to form:

‘the house-elf who the witch gave to Harry’.

Moreover, we can also relativize the NP that occurs in the PP ‘to Harry’. That is, we can take ‘Harry’ out of this position, leaving the ‘to’, behind to form:

‘Harry, who the witch gave the house-elf to’.

We would like our DCG to handle such examples. But note — there are also some things that the DCG should *not* do, namely perform *multiple extraction*. There are now two possible NPs that can be moved: the first argument NP, and the NP in the PP. Can both be moved at once? In some languages, yes. In English, no. That is, in English

∗ ‘who the witch gave to’,

is *not* a well-formed relative clause (at least not if we view it as resulting from the extraction of two NPs. It could be short for ‘who the witch gave something to’. In this case, the direct object would have been dropped and not extracted). So when we write our DCG, not only do we have to make sure we generate the NPs we want, we also have to make sure that we don’t build NPs using multiple extraction.

Now, we can develop our previous DCG to do this — but the result (found in See file `dCG4Gaps2.pl`.) is *not* something a computational linguist should be proud of. Let’s take a closer look.

As we are going to need to build prepositional phrases, we need a rule to build them:

```
pp(Gap) --> p,np(Gap).
```

This says: a prepositional phrase can be built out of a preposition followed by an NP. We have used the extra argument to pass up the value of the *Gap* feature from the NP to the PP. So the PP will know whether the NP is an ordinary one or a gap.

We’ll also need a larger lexicon: we’ll add the verb ‘gave’ and the preposition ‘to’:

```
v(2) --> [gave].

p --> [to].
```

Now comes the crucial part: the new VP rules. We need to allow single extractions, and to rule out double extractions. Here’s how this can be done — and this is the part linguists *won’t* like:

```
vp(Gap) --> v(2), np(nogap),pp(Gap).

vp(Gap) --> v(2), np(Gap),pp(nogap).
```

We have added two VP rules. The first rule insists that the NP argument be gap-free, but allows the possibility of a gap in the PP. The second rule does the reverse: the NP may contain a gap, but the PP may not. Either way, at least one of the VPs two arguments must be gap-free, so multiple extractions are ruled out.

Now, this does work. For example it generates such sentences as: `s(_,[the,witch,gave,the,house-elf,to,harr`

And we can relativize in the NP argument: `np(_,[the,house-elf,who,the,witch,gave,to,harry],[]).`

And in the PP argument: `np(_,[harry,who,the,witch,gave,the,house-elf,to],[]).`

Moreover, the DCG refuses to accept multiple extractions, just as we wanted: `np(_,[the,house-elf,who,the,wi`

So why would a linguist not approve of this DCG? *Because we are handling one construction — the formation of VPs using V(2) verbs — with two rules.*

The role of syntactical rules is to make a structural claim about the combination possibilities in our language. Thus there should be *one* rule for building VPs out of V(2) verbs, for there is only *one* structural possibility: V(2) verbs take an NP and a PP argument, in that order. We used two rules not because there were two possibilities, but simply to do a bit of 'feature hacking': by writing two rules, we found a cheap way to rule out multiple extractions. And this is a silly way to write a grammar. As we saw when we discussed the case example, one role of features is precisely to help us minimize the number of rules we need — so to add extra rules to control the features is sheer craziness!

There are also practical disadvantages to the use of two rules. For a start, many unambiguous sentences now receive two analyses. For example, 'The witch gave the house-elf to Harry' is analyzed in two distinct ways.

Such a *spurious analysis* is a real nuisance — natural language is ambiguous enough anyway. We certainly *don't* need to add to our troubles by writing DCGs in a way that guarantees that we generate too many analyses!

Furthermore, adding extra rules is just not going to work in the long run. As we add more verb types, we are going to need more and more duplicated rules. The result will be a mess, and the grammar will be ugly and hard to maintain. We need a better solution — and there is one.

5.3.4 Gap Threading

The basic idea underlying gap threading is that instead of using simple feature values such as `nogap` and `gap(np)` we are going to work with more structure. In particular, the value of the `Gap` feature is going to be a difference list. Think of the first item (list) as 'what we have before we analyze this category' and the second one as 'what we have afterwards'. Or more simply: think of the list as the 'in value', and the second item as the 'out value'.

Here are the new NP rules (found in See file `dCG4GapThreading.pl`.):

```
np(F-F) --> det,n.
np(F-F) --> det,n,rel.
np(F-F) --> pn.
np(F-F) --> pn,rel.
np([gap(np)|F]-F) --> [].
```

Note that in the first four rules, the difference list `F-F` is doing the work that `nogap` used to do. And this is the way it should be. After all, `F-F` is a difference list representation of the empty list. So, for example, the first rule says that an NP can consist of a proper name, and when an NP is built that way, no material has been removed. (That is: the in value is the same as the out value.)

What about the last rule? This says that we can build empty NPs, but that when we do so, we have to add `gap(np)` to the first list. That is, in this case there is a difference between the in value and the out value: the difference is precisely the `gap(np)` value.

The S rule is analogous to our old one:

```
s(F-G) --> np(F-F),vp(F-G).
```

This says that the subject NP must be an ordinary NP (recall: `F-F` is essentially the same as `nogap`) and that the difference list associated with the VP is passed up to the S. That is, just as before we are performing feature passing, except that now the `Gap` feature is a difference list.

Thus the rules for PPs and relativization should not be puzzling:

```
pp(F-G) --> p,np(F-G).


rel --> prorel,s([gap(np)|F] - F).
rel --> prorel,vp(F-F).
```

Once again, these are exact analogs of our earlier rules — except that we are using complex features.

So we come at last to the critical part: how do we handle VPs? As follows:

```
vp(F-G) --> v(1),np(F-G).
vp(F-G) --> v(2),np(F-H),pp(H-G).
 % lexicon
```

This looks nice: we have one rule for each of the verb types. The first rule is is analogous to our earlier `V(1)` rule. But what about the second rule? Why do we only need one for `V(2)` verbs?

Think about it. The most complicated feature we have is

```
[gap(np)|F]-F]
```

and this indicates that precisely `one` element is missing. So when we enter the second `VP` rule, there is only one missing item and two arguments. It may end up in the first position, or the second, but (as there is only one) it cannot end up in both.

This DCG is well worth studying. For a start, you should carefully compare it with our previous DCG (See file `dCG4Gaps2.pl`.). Note the *systematic* link between the two. Second, you really should play with lots of traces so you can see how the gap is threaded in and out of the various categories. Start by running our example queries from before (page 83).

This threading technique can be used to thread other things besides gaps. For example, it can be used in computational semantics. There is a simple way of implementing Discourse Representation Theory (DRT): one threads the semantic representations (the DRSs) through the parse tree. In this way, each part of the sentence gets to make its own semantic contribution. You can find a discussion of this in Volume 2 of the textbook on semantics by Blackburn and Bos, which is available at *www.comsem.org*.

5.3.5 Questions

As we have promised above, we now apply the technique of gap threading to another phenomenon: wh-questions. Let us look at an example:

- 'Harry likes the witch.'
- 'Who likes the witch?'

So we can think of this wh-question as a sentence with a gap:

Who GAP(NP) likes the witch?

If we want to ask about the witch, not about Harry, we form the following question:

Who does Harry like GAP(NP)?

So this looks very much like the analysis of relative clauses that we had in the previous paragraph, except for one small complication: When we ask about the object, we have to use an auxiliary and the infinite form of the verb ('does...like'). Apart from that, the heart of the new DCG (found in See file `dCG4Questions.pl`.) contains nothing really new:

```
%subject interrogative

s(F-F) --> wh, vp(F-F,fin).

%object interrogative

s(F-F) --> wh, aux, np(F-F), vp([gap(np)|F]-F, inf).

vp(F-G, FIN) --> v(1, FIN),np(F-G).

vp(F-G, FIN) --> v(2, FIN),np(F-H),pp(H-G).
```

Auxiliary and a wh-pronouns are added to the lexicon straight forward:

```
% auxiliary
```

```
aux --> [does].

% wh-pronoun

wh --> [who].
```

The annotation of verbs as being finite or not is achieved by adding a new feature:

```
v(1,inf) --> [like].

v(1,fin) --> [likes].

...
```

Finally, we have said that relative constructions are a case of *unbounded dependencies*: an arbitrary amount of material may stand between the gap and the extracted and moved noun phrase. The same holds for wh-questions: From a sentence like 'Harry said that a witch likes Ron.' we can get to the question

Who did Harry say that a witch likes GAP(NP)?

And of course we can make the question even longer, for example like this:

Who did Hermione say that Harry said that a witch likes GAP(NP)?

5.3.6 Occurs-Check

There is one thing about Prolog we can learn using the previous DCG (See file `dCG4Questions.pl`.). If you feed this DCG to Prolog and play with it, you will see that it accepts sentences (questions) that should not be accepted:

```
2 ?- s(_,[who,likes],[]).

Yes
```

What happens here is that NPs can be empty that should not be allowed to. But our DCG is correct. So how can this be? The problem is an unwanted interaction between Prolog's unification mechanism and the way we represent empty difference lists. We write `np(F-F)` to make sure that the list containing the gaps of an NP is empty. But in fact, Prolog can unify such an NP with an NP having a gap:

```
?- np(F-F) = np([gap(np)|G]-G).

F = [gap(np), gap(np), gap(np), gap(np), gap(np), gap(np), gap(np), gap(np), g
G = [gap(np), gap(np), gap(np), gap(np), gap(np), gap(np), gap(np), gap(np), g

Yes
```

The reason lies in Prolog's standard unification mechanism. For reasons of efficiency, it leaves out the so-called *Occurs-Check* . This plays no role most of the time. But in some cases it does. Let's look at a canonical example to see what's going wrong:

```
?- A = f(A).

A = f(f(f(f(f(f(f(f(f(f(...))))))))))

Yes
```

If you ask Prolog to unify a variable with a complex term, it does not check whether this variable *occurs* inside the term. Therefore you can unify the variable and the term in the example above resulting in an infinite structure. You can think of what is happening as follows: First, `A` on the left matches `f(A)` on the right and becomes `f(A)`. At the same time, `f(A)` on the right becomes `f(f(A))`. This matches `f(A)` on the left again and becomes `f(f((A)))`, and so on.

The simplest solution to this is not to use Prolog's standard unification. For example in SWI Prolog there is a special predicate `unify_with_occurs_check/2`:

```
?- unify_with_occurs_check(A,f(A)).

No.
```

Now that was the true story. It looks like we will have to use unification-with-occurs-check, instead of the standard built-in Prolog unification. Luckily, there's another, much simpler, solution in our case: Nothing hinges on how exactly we represent the empty difference list. We don't have to use variables for this purpose (in particular, we needn't be as unspecific as in the calls shown above, where we gave one anonymous variable _ for the whole difference list). Instead of using variables, we can also use a non-variable '`-`'-term, as long as we make sure that the difference between both sides of the '`-`' is the empty list. For instance, we can use the following to symbolize an empty difference list: `[]-[]`.

Indeed `np([]-[])` and `np([gap(np)|G]-G)` don't unify, in contrast to `np(F-F)` and `np([gap(np)|G]-G)`. So calling `s([]-[],[who,likes],[])` instead of `s(_,[who,likes],[])` should solve our problem - and so it does.

5.4 Pros and Cons of DCGs

We have seen that we can do a lot with DCGs, some of it quite surprising. For instance, the feature passing mechanism of DCGs makes it possible to give a surprisingly straightforward account of long distance dependencies, especially when more complex features values (and in particular, difference lists) are used. So: just how good are DCGs? Let's start by thinking about their good points:

1. It seems fair to say that DCGs are neat in a way. Certainly in the form we have been writing them, in which we just have simple context free rules, regulated by feature agreement. Of course, in DCGs we are free to add arbitrary pieces

of code to our rules — and if we do this, we are working with a full-powered programming language with all its complexities. But it is clear from our work that we don't have to do this. Simple feature agreement can already perform a lot of useful linguistic work for us.

2. Secondly, DCGs are declarative (as opposed to procedural). Certainly the rules we have written above have a clear declarative interpretation: they simply license certain configurations of features and categories. We don't *have* to think about them in terms of procedures that transform structures (e.g. movement).

But DCGs have certain disadvantages:

1. For a start, they lock us into one way of handling grammars. DCGs carry our recognition and parsing using top-down depth-first search — for that's the way the Prolog search strategy works. This won't always be the best way to work (sometimes a bottom-up, or left-corner, strategy might be better). But if we decide to use DCGs, we're forced to accept the top down approach. As computational linguists, it really is our business to know a lot about grammars and how to compute with them — we shouldn't accept the off-the-shelf solution offered by DCGs just because it's there.
2. Our DCGs make it very clear that "grammars + features" is potentially a very powerful strategy. But our use of the idea has been highly Prolog specific. What exactly is a feature? How should they be combined? Is the idea of gap threading really fundamental as a way of handling long-distance dependencies, or is that just neat a Prolog implementation idea? These are important questions, and we should be looking for answers to them.
3. Finally, it should be remarked that DCGs can quickly get clumsy to write. DCGs with six features are OK; DCGs with 30 features, are painful. It would be nice if we had a more friendly notation for working with features.

This discussion pretty much tells us what we should do next. For the remainder of the course, we will examine the two components (grammars and features) in isolation, and only right at the end will we put them back together. Most of our time will be spent looking at context free grammars and how to work with them computationally. Once we have a good understanding of what is involved in recognizing/parsing context free languages, we will take a closer look at features. Finally, we'll bring them together again. This will result in a system that has all the advantages of DCGs, and none of the disadvantages.

5.5 The Complete Code

See file `dCGExample.pl`.	A small DCG for very simple English sentences (see Section 5.
See file `wrongDCG4Pronouns.pl`.	A DCG for simple English sentences with pronouns that overge
See file `badDCG4Pronouns.pl`.	A DCG for simple English sentences with pronouns that uses a
See file `dCG4Pronouns.pl`.	A DCG for simple English sentences with pronouns using featu
See file `dCG4Gaps.pl`.	A DCG for simple English relative clauses (see Section 5.3.2).
See file `dCG4Gaps2.pl`.	A not very promising extension of See file `dCG4Gaps.pl`. (se
See file `dCG4GapThreading.pl`.	A DCG for simple English relative clauses using gap threading
See file `dCG4Questions.pl`.	A DCG for simple English questions using gap threading (see S

5.6 Exercises

Exercise 5.1 *In the text (page 84) we claimed that the DCG found in See file `dCG4Gaps2.pl`. gives two distinct analyses of 'The witch gave the house-elf to Harry'.*

1. *Design a Prolog query to verify this observation.*
2. *Explain why there are two analyses.*

Exercise 5.2 *Extend the gap-threading DCG (See file `dCG4Gaps2.pl`.) so that it handles the English relative pronouns 'that' and 'which', adjectives, and subject-verb agreement.*

Exercise 5.3 *[This is a possible mid-term project.]*

Write a DCG that handles the basic facts of German relative clauses. Your DCG should correctly handle basic facts about agreement, case, and German word order in relative clauses.

6

Parsing: Bottom-Up and Top-Down

6.1 Bottom-Up Parsing and Recognition

The basic idea of *bottom-up parsing and recognition* is to begin with the concrete data provided by the input string — that is, the words we have to parse/recognize — and try to build bigger and bigger pieces of structure using this information. Eventually we hope to put all these pieces of structure together in a way that shows that we have found a sentence.

Putting it another way, bottom-up parsing is about moving from concrete low-level information, to more abstract high-level information. And this is reflected in a very obvious point about any bottom-up algorithm: *In bottom-up parsing, we use our CFG rules right to left.*

What does this mean? Consider the CFG rule $C \longrightarrow P1\ P2\ P3$. Working bottom-up means that we will try to find a P1, a P2, and a P3 in the input that are right next to each other. If we find them, we will use this information to conclude that we have found a C. That is, in bottom-up parsing, the flow of information is from the *right* hand side of the rules to the *left* hand side of the rules.

Let's look at an example of bottom-up parsing/recognition. Suppose we're working with the grammar of English that was given earlier (page 73), and that we want to see whether the following string of symbols is a sentence:

```
vincent shot marsellus.
```

Working bottom-up, we might do the following. First we go through the string, systematically looking for strings of length 1 that we can rewrite by using our CFG rules in a right to left direction. Now, we have the rule $PN \longrightarrow vincent$, so using this in a right to left direction gives us:

```
pn shot marsellus.
```

But wait: we also have the rule $NP \longrightarrow PN$, so using this right to left we build:

```
np shot marsellus
```

It's time to move on. We're still looking for strings of length 1 that we can rewrite using our CFG rules right to left — but we can't do anything with `np`. But we *can* do something with the second symbol, `shot`. We have the rule $TV \longrightarrow shot$, and using this right to left yields:

```
np tv marsellus
```

Can we rewrite `tv` using a CFG rule right to left? No — so we move on and see what we can do with the last symbol, `marsellus`. We have the rule $PN \longrightarrow marsellus$, and this lets us build:

```
np tv pn
```

Now, we can apply the rule $NP \longrightarrow PN$ once more and get:

```
np tv np
```

Are there any more strings of length 1 we can rewrite using our context free rules right to left? No — we've done them all. So now we start again at the beginning looking for strings of length 2 that we can rewrite using our CFG rules right to left. And there is one: we have the rule $VP \longrightarrow TV\ NP$, and this lets us build:

```
np vp
```

Are there any other strings of length 2 we can rewrite using our CFG rules right to left? Yes — we can now use $S \longrightarrow NP\ VP$:

```
s
```

And this means we are finished. Working bottom-up we have succeeded in rewriting our original string of symbols into the symbol `s` — so we have successfully recognized 'Vincent shot marsellus' as a sentence.

Well, that was an example. A couple of points are worth emphasizing. This is just one of many possible ways of performing a bottom-up analysis. All bottom-up algorithms use CFG rules right to left — but there many different ways this can be done. To give a rather pointless example: we could have designed our algorithm so that it started reading the input in the middle of the string, and then zig-zagged its way to the front and back. And there are many much more serious variations — such as the choice between depth first and breadth first search — which we shall discuss in the second part of this chapter.

In fact, the algorithm that we used above is crude and inefficient. But it does have one advantage — it is easy to understand, and as we shall soon see, easy to put into Prolog.

6.2 Bottom-Up Recognition in Prolog

The main goal of this section is to write a simple bottom-up recognizer in Prolog. But before we do this, we have to decide how to represent CFGs in Prolog. The representation that we are going to introduce distinguished between phrase structure rules and lexical rules by representing them in different ways. As we mentioned above, it is often useful to be able to make this distinction when dealing with natural languages. For representing phrase structure rules, we shall use a notation that is quite close to the one used in DCGs. In fact, there are only two differences. First, whereas DCGs use the symbol

```
-->
```

for the rewrite arrow, we shall use the symbol

```
--->
```

Second, in DCGs we would write $S \longrightarrow NP\,VP$ as:

```
s --> np,vp.
```

However we shall use instead the following notation:

```
s ---> [np,vp].
```

Here's an example. The phrase structure rules of our English grammar (page 73) become in this notation (from file *See file* `ourEng.pl`.):

```
s     ---> [np, vp].

np    ---> [pn].

np    ---> [pn,rel].

np    ---> [det, nbar].

nbar  ---> [n].

nbar  ---> [n, rel].

rel   ---> [wh, vp].

vp    ---> [iv].

vp    ---> [tv, np].

vp    ---> [dv, np, pp].

vp    ---> [sv, s].

pp    ---> [p, np].
```

How does Prolog know about the symbol `--->`? Well, it needs to be told what it means, and we can do this using an operator definition as follows:

```
?- op(700,xfx,--->).
```

That is, we have declared `--->` to be a binary infix operator. The best place to put this definition is probably in the recognizer, not in each and every grammar. But note: this means you will have to consult the recognizer before you consult any of the the grammars, as otherwise Prolog won't know what `--->` means.

Now, we can represent phrase structure rules. Lexical rules we shall represent using the predicate `lex/2`. For example, *PN* ⟶ *vincent* will be represented as `lex(vincent,pn)`. Here are the lexical rules of the little English grammar that we have seen above in the new notation.

```
lex(vincent,pn).
lex(mia,pn).
lex(marsellus,pn).
lex(jules,pn).
lex(a,det).
lex(the,det).
lex(her,det).
lex(his,det).
lex(gun,n).
lex(robber,n).
lex(man,n).
lex(woman,n).
lex(who,wh).
lex(that,wh).
lex(to,p).
lex(died,iv).
lex(fell,iv).
lex(loved,tv).
lex(shot,tv).
lex(knew,tv).
lex(gave,dv).
lex(handed,dv).
lex(knew,sv).
lex(believed,sv).
```

Incidentally — we shall use this notation for grammars throughout the course. All our parser/recognizers will make us of grammars in this format.

It's now time to define our very first recognizer — a simple (though inefficient) recognizer which carries out the algorithm sketched above. Here it is. The predicate `recognize_bottomup/1` takes as input a list of symbols (for example, `[vincent,shot,marsellus]`) and tries to build the list `[s]` (that is, a sentence). Here is its definition (from file See file `bottomup_recognizer.pl`.):

```
recognize_bottomup([s]).
 %%% the recursive case: choose a part of the incoming
%%% string which matches with the right hand side of a
%%% rule and replace it with the category symbol
%%% on the left hand side of that rule. Recursivel
%%% recognize the result.
```

```
recognize_bottomup(String) :-
        /*** uncomment the following line to see which states
             the recognizer goes through ***/
        % nl,write('STATE: '),write(String),
        split(String,Front,Middle,Back),
        ( Cat ---> Middle
         ;
          (Middle = [Word], lex(Word,Cat))
        ),
        /*** uncomment to see which rules are applied ***/
        % tab(5),write('RULE: '),write((Cat ---> Middle)),nl,
        split(NewString,Front,[Cat],Back),
        recognize_bottomup(NewString).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% split(+list, ?list, ?list, ?list)
%%% splits a list into three segments
```

How does this work? Well, the first clause, `recognize_bottomup([s])`, tells us that we have succeeded if we find the list `[s]`. Incidentally, if you glance down at the following clause, you will see that `recognize_bottomup/1` is recursive. This first clause is the base clause of the recursion.

So what about the second clause? First we have:

```
split(String,Front,Middle,Back)
```

The predicate `split/4` splits a list into three parts. It is defined as follows:

```
split(ABC, A, B, C) :-
        append(A, BC, ABC),
        append(B, C, BC).
```

`split/4` uses the standard `append/3` predicate to split up the incoming list by calling it with uninstantiated varibles in the first two arguments. `append/3` is called twice: The first time the incoming list is split in two parts, and the second time one of the parts is again split into two parts, resulting in three parts altogether. Unfortunately, using `append/3` in this way is very inefficient.

So — `split/4` splits the string into three parts: `Front`, `Middle`, and `Back`. Next comes a disjunction:

```
Cat ---> Middle
;
(Middle = [Word], lex(Word,Cat))
```

It succeeds if we have either a phrase structure rule with `Middle` as its right hand side, or if `Middle` is actually a word that we can map to a category by a lexical rule.

Now for the key step. Suppose we have such a rule. Then

```
split(NewString,Front,[Cat],Back)
```

builds a new string out of our input string by replacing the list `Middle` with the list `[Cat]`. That is, from

```
Front   Middle   Rest
```

we get the new string

```
Front   Cat   Rest
```

Note how we used the predicate `split/4` here in reverse to "unsplit", that is to join the lists.

In short: we have used our rule right to left to build a new string. The rest is simple. We recursively call

```
recognize_bottomup(NewString)
```

on the new string we have built. If we have a sentence on our hands, this recursion will eventually produce `[s]`, and we will succeed using the first clause. Note that every call to `recognize_bottomup/1` makes use of `append/3` to decompose the input list. So, via backtracking, we will eventually find all possible ways of decomposing the input list — thus if the input really is a sentence, we will eventually succeed in showing this.

6.3 An Example Run

Let's look at an example, to see if we've got it right. If you ask Prolog

`recognize_bottomup([vincent,shot,marsellus])`

it will answer `yes`, as it should. Try some other examples and check whether Prolog answers the way it should. A trace will give you more information about how Prolog is arriving at these answers. Here is an abbreviated trace for the query `recognize_bottomup([vincent,shot,` You can see how Prolog splits up the string that is to be recognized into three parts, which rules it applies for replacing the middle part with a category symbol, and you can see the recursive calls of `recognize_bottomup` that it makes.

```
?- recognize_bottomup([vincent,shot,marsellus]).
Call: (7) recognize_bottomup([vincent, shot, marsellus]) ?
Call: (8) split([vincent, shot, marsellus], _G647, _G648, _G649) ?
Exit: (8) split([vincent, shot, marsellus], [], [vincent], [shot, marsellus])
Call: (8) lex(vincent, _G653) ?
Exit: (8) lex(vincent, pn) ?
 Call: (8) recognize_bottomup([pn, shot, marsellus]) ?
 Call: (9) split([pn, shot, marsellus], _G656, _G657, _G658) ?
 Exit: (9) split([pn, shot, marsellus], [], [pn], [shot, marsellus]) ?
 Call: (9) _G658--->[pn] ?
 Exit: (9) np--->[pn] ?
  Call: (9) recognize_bottomup([np, shot, marsellus]) ?
  Call: (10) split([np, shot, marsellus], _G662, _G663, _G664) ?
  Exit: (10) split([np, shot, marsellus], [np], [shot], [marsellus]) ?
  Call: (10) lex(shot, _G671) ?
  Exit: (10) lex(shot, tv) ?
   Call: (10) recognize_bottomup([np, tv, marsellus]) ?
   Call: (11) split([np, tv, marsellus], _G677, _G678, _G679) ?
   Exit: (11) split([np, tv, marsellus], [np, tv], [marsellus], []) ?
   Call: (11) lex(marsellus, _G689) ?
   Exit: (11) lex(marsellus, pn) ?
    Call: (11) recognize_bottomup([np, tv, pn]) ?
    Call: (12) split([np, tv, pn], _G698, _G699, _G700) ?
    Exit: (12) split([np, tv, pn], [np, tv], [pn], []) ?
    Call: (12) _G706--->[pn] ?
    Exit: (12) np--->[pn] ?
     Call: (12) recognize_bottomup([np, tv, np]) ?
     Call: (13) split([np, tv, np], _G716, _G717, _G718) ?
     Exit: (13) split([np, tv, np], [np], [tv, np], []) ?
     Call: (13) _G724--->[tv, np] ?
     Exit: (13) vp--->[tv, np] ?
      Call: (13) recognize_bottomup([np, vp]) ?
      Call: (14) split([np, vp], _G731, _G732, _G733) ?
      Exit: (14) split([np, vp], [], [np, vp], []) ?
      Call: (14) _G736--->[np, vp] ?
      Exit: (14) s--->[np, vp] ?
       Call: (14) recognize_bottomup([s]) ?
       Exit: (14) recognize_bottomup([s]) ?
      Exit: (13) recognize_bottomup([np, vp]) ?
     Exit: (12) recognize_bottomup([np, tv, np]) ?
    Exit: (11) recognize_bottomup([np, tv, pn]) ?
   Exit: (10) recognize_bottomup([np, tv, marsellus]) ?
  Exit: (9) recognize_bottomup([np, shot, marsellus]) ?
 Exit: (8) recognize_bottomup([pn, shot, marsellus]) ?
Exit: (7) recognize_bottomup([vincent, shot, marsellus]) ?

Yes
```

This trace only shows the essence of how the recognizer arrives at its answer. We cut out all the rest. Try it yourself and you will see that the recognizer spends a LOT of time trying out different ways of splitting up the string.

6.4 How Well Have we Done?

6.4.1 Implementation

By this stage of your Prolog career, it should be quite clear to you why this recognizer is badly implemented. We made heavy use of `append/3` to subdivide lists into the required pattern. This is very inefficient. If you examine a trace, you will see the the program spends most of its time trying out different ways of putting lists together. The time it devotes to the key recognition steps is comparatively small.

It's worth emphasizing that this implementation inefficiency has nothing to do with the basic idea of bottom-up recognition/parsing. For a start, there are many nice Prolog implementations of fairly simple bottom-up parsers — in particular, what are known as shift-reduce parsers — that are *much* better than this. Moreover, soon we will be discussing naive top-down parsing/recognition. If this is implemented using `append/3`, the result is just as inefficient as what we have just seen. But we are going to take care to avoid this implementation inefficiency there. We'll be using difference lists instead, and as we'll see, the result is a *lot* faster.

6.4.2 Algorithmic Problems

There is, however, a deeper problem. What we have just discussed is a *naive* bottom-up recognizer — but its naivete has nothing to do with its implementation.It has an inefficiency you will find in many different kinds of parsers/recognizers namely *The algorithm needlessly repeats work.*

Consider the sentence 'The robber knew Vincent shot Marsellus' As we have already mentioned, this sentence is locally ambiguous. In particular, the first part of the string, 'The robber knew Vincent' is a sentence. Now, our naive bottom-up algorithm will find that this string is a sentence — and then discover that there are more words in the input. Hence, in this case, `s` is not the right analysis. Thus our parser will backtrack and it will *undo and forget all the useful work it has done*. For example, while showing that 'The robber knew Vincent' is a sentence, the parser has to show that 'The robber' is an `np`. Great! This is dead right! And we need to know this to get the correct analysis! But then, when it backtracks, it undoes all this work. So when it finally hits on the right way of analyzing 'The robber knew Vincent shot Marsellus' , it will once again demonstrate that 'The robber' is an NP. If lots of backtracking is going on, it may end up showing this simple fact over and over and over again.

It is for this reason that we call the algorithm described above 'naive'. The poor implementation is easily corrected — our top-down parser won't have it — but this deeper problem is harder to solve.

6.4.3 [Sidetrack] Using a Chart

But the problem *can* be solved. We can perform *non*-naive parsing by using *charts*. In their simplest form, charts are essentially a record of what pieces of information we found and where (for example, that the initial string 'The robber' is an `np`). Sophisticated parsing algorithms use the chart as a look-up table: they check whether they have already produced an analysis of a chunk of input. This saves them having to repeat needless work.

And the use of charts really does pay off: naive algorithms are *exponential* in the size of their input — which, roughly speaking, means they may take an impossibly long time to run *no matter how good the implementation is*. The use of charts, however, turns context free parsing into a *polynomial* problem. That is, roughly speaking, it turns the problem of finding a parse into a problem that can be solved in reasonable time, for any CFG grammar.

We will study the idea of *chart parsing* later in the course: it is an idea of fundamental importance. But note well: chart parsing is not an idea we need to study *instead of* the various naive parsing strategies — its something we need to study *in addition to* them. The various naive strategies (bottom-up, top-down, left corner,...) are importantly different, and we need to understand them. What is nice about chart parsing is that it is a general idea that can remove the naivete from *all* these approaches.

6.5 Top Down Parsing and Recognition

6.5.1 The General Idea

Remember that in bottom-up parsing/recognition we start at the most concrete level (the level of words) and try to show that the input string has the abstract structure we are interested in (this usually means showing that it is a sentence). So we use our CFG rules right-to-left.

In *top-down parsing/recognition* we do the reverse: We start at the most abstract level (the level of sentences) and work down to the most concrete level (the level of words). So, given an input string, we start out by assuming that it is a sentence, and then try to prove that it really is one by using the rules *left-to-right*. That works as follows: If we want to prove that the input is of category S and we have the rule $S \rightarrow NP\,VP$, then we will try next to prove that the input string consists of a noun phrase followed by a verb phrase. If we furthermore have the rule $NP \rightarrow Det\ N$, we try to prove that the input string consists of a determiner followed by a noun and a verb phrase. That is, we use the rules in a left-to-right fashion to expand the categories that we want to recognize until we have reached categories that match the preterminal symbols corresponding to the words of the input sentence.

Of course there are lots of choices still to be made. Do we scan the input string from right-to-left, from left-to-right, or zig-zagging out from the middle? In what order should we scan the rules? More interestingly, do we use *depth-first* or *breadth-first* search?

In what follows we'll assume that we scan the input left-to-right (that is, the way we read) and the rules from top to bottom (that is, the way Prolog reads). But we'll look at both depth first and breadth-first search.

6.5.2 With Depth-First Search

Depth first search means that whenever there is more than one rule that could be applied at one point, we first explore one possibility (and all its consequences). Only if we fail, we consider the alternative(s) following the same strategy. So, we stick to a decision as long as possible.

Let's look at an example. Here's part of the grammar `ourEng.pl` again:

```
s      ---> [np, vp].

np     ---> [pn].

vp     ---> [iv].
vp    ---> [dv, np, pp].

lex(vincent,pn).

lex(mia,pn).

lex(loved,tv).

lex(knew,tv).

lex(gave,dv).
```

The sentence “Mia loved vincent” is admitted by this grammar. Let’s see how a top-down parser using depth first search would go about showing this. The following table shows the steps a top-down depth first parser would make. The second row gives the categories the parser tries to recognize in each step and the third row the string that has to be covered by these categories.

	State		Comments
1.	`s`	*mia loved vincent*	`s --> [np,vp]`
2.	`np vp`	*mia loved vincent*	`np --> [pn]`
3.	`pn vp`	*mia loved vincent*	`lex(mia,pn)` We’ve got a match
4.	`vp`	*loved vincent*	`vp --> [iv]` We’re doing depth first search. So we ignore the other vp rule for the moment.
5.	`iv`	*loved vincent*	No applicable rule. Backtrack to the state in which we last applied a rule. That’s state 4.
4’.	`vp`	*loved vincent*	`vp --> [tv]`
5’.	`tv np`	*loved vincent*	`lex(loved,tv)` Great, we’ve got match!
6’.	`np`	*vincent*	`np --> [pn]`
7’.	`pn`	*vincent*	`lex(vincent,pn)` Another match. We’re done.

It should be clear why this approach is called top-down: we clearly work from the abstract to the concrete, and we make use of the CFG rules left-to-right.

And why was this an example of depth first search? Because when we were faced with a choice, we selected one alternative, and worked out its consequences. If the choice turned out to be wrong, we backtracked. For example, above we were faced with a choice of which way to try and build a VP — using an intransitive verb or a transitive verb. We first tried to do so using an intransitive verb (at state 4) but this didn’t work out (state 5) so we backtracked and tried a transitive analysis (state 4’). This eventually worked out.

6.5.3 With Breadth-First Search

Let's look at the same example with breadth-first search. The big difference between breadth-first and depth-first search is that in breadth-first search we pursue all possible choices "in parallel", instead of just exploring one. So, instead of commiting to one decision, we so to speak jump between all alternatives.

It is useful to imagine that we are working with a big bag containing all the possibilities we should look at — so in what follows we have used set-theoretic braces to indicate this bag. When we start parsing, the bag contains just one item.

	State	Comments
1.	{⟨ `s`, *mia loved vincent*⟩}	`s --> [np, vp]`
2.	{⟨ `np vp`, *mia loved vincent*⟩}	`np --> [pn]`
3.	{⟨ `pn vp`, *mia loved vincent*⟩}	Match!
4.	{⟨ `vp`, *loved vincent*⟩}	`vp --> [iv], vp --> [tv, np]`
5.	{⟨ `iv`, *loved vincent*⟩, ⟨ `tv np`, *loved vincent*⟩}	No applicable rule for iv analysis. lex(loved,tv)
6.	{⟨ `np`, *vincent*⟩}	`np --> [pn]`
7.	{⟨ `pn`, *vincent*⟩}	We're done!

The crucial difference occurs at state 5. There we try both ways of building VPs at once. At the next step, the intransitive analysis is discarded, but the transitive analysis remains in the bag, and eventually succeeds.

The advantage of breadth-first search is that it prevents us from zeroing in on one choice that may turn out to be completely wrong; this often happens with depth-first search, which causes a lot of backtracking. Its disadvantage is that we need to keep track of all the choices — and if the bag gets big (and it may get *very* big) we pay a computational price.

So which is better? There is no general answer. With some grammars breadth-first search, with others depth-first.

?- Question!

Can you explain whe these to serach strategies are called *depth-first* and *breadth-first, respectively?*

6.6 Top Down Recognition in Prolog

6.6.1 The Code

It is easy to implement a top-down *depth-first* recognizer in Prolog — for this is the strategy Prolog itself uses in its search. Actually, it's not hard to implement a top-down *breadth-first* recognizer in Prolog either, though we are not going to discuss how to do that. As we said earlier, this implementation will be far better than that used in the naive bottom-up recognizer. This is not because because top-down algorithms are better than bottom-up ones, but simply because we are not going to use `append/3`. Instead we'll use difference lists.

Here's the main predicate, `recognize_topdown/3`. Note the operator declaration (we want to use our `--->` notation we introduced last week).

```
:- op(700,xfx,--->).


recognize_topdown(Category,[Word|Reststring],Reststring) :-
        /*** Uncomment the following lines to see the steps the top
             down parser takes ***/
        %%% nl, write('String to recognize: '), write([Word|Reststring]),
        %%% nl, write('Category to recognize: '), write(Category),
        lex(Word,Category).


recognize_topdown(Category,String,Reststring) :-
        Category ---> RHS,
        /*** Uncomment the following lines to see which steps the
             recognizer takes. ***/
        %%% nl, write('Rule: '), write((Category ---> RHS)),
        matches(RHS,String,Reststring).
```

Here `Category` is the category we want to recognize (`s`, `np`, `vp`, and so on). The second and third argument are a difference list representation of the string we are working with (you might find it helpful to think of the second argument as a pointer to the front of the list and the third argument `Reststring` as a pointer to the rest of the list).

The first clause deals with the case that `Category` is a preterminal that matches the category of the next word on the input string. That is: we've got a match and can remove `Word` from the string that is to be recognized.

The second clause deals with phrase structure rules. Note that we are using the CFG rules right-to-left: `Category` will be instantiated with something, so we look for rules with `Category` as a left-hand-side, and then we try to match the right-hand-side of these rules (that is, `RHS`) with the string.

Now for `matches/3`, the predicate which does all the work:

```
matches([],String,String).


matches([Category|Categories],String,RestString) :-
        recognize_topdown(Category,String,String1),
        matches(Categories,String1,RestString).
```

The first clause handles an empty list of symbols to be recognized. The string is returned unchanged. The second clause lets us match a non-empty list against the difference list. This works as follows. We want to see if `String` begins with strings belonging to the categories

```
[Category|Categories]
```

leaving behind `RestString`. So we see if `String` starts with a substring of category `Category` (the first item on the list). Then we recursively call `matches` to see whether what's left over (`String1`) starts with substrings belonging to the categories `Categories` leaving behind `RestString`. This is classic difference list code.

Finally, we can wrap this up in a driver predicate:

```
recognize_topdown(String) :-
        recognize_topdown(s,String,[]).
```

Now we're ready to play. We shall make use of the `ourEng.pl` grammar that we worked with before.

6.6.2 A Quick Evaluation

We used this same grammar with our bottom-up recognizer — and we saw that it was very easy to grind `bottomup_recognizer.pl` into the dust (see Section 6.4). Try this example sentence again:

'Jules believed the robber who shot Marsellus fell. '

The bottom-up recognizer takes a long time on this examples. But the top-down program handles it without problems:

`recognize_bottomup([jules,believed,the,robber,who,shot,marsellus,fell]).`

`recognize_topdown([jules,believed,the,robber,who,shot,marsellus,fell]).`

The following sentence is not admitted by the grammar, because the last word is spelled wrong (felll instead of fell).

'Jules believed the robber who shot Marsellus felll.'

Unfortunately it takes our bottom-up recognizer a long time to find that out, and hence to reject the sentence. The top-down program is far better:

`recognize_bottomup([jules,believed,the,robber,who,shot,marsellus,felll]).`

`recognize_topdown([jules,believed,the,robber,who,shot,marsellus,felll]).`

6.7 Top Down Parsing in Prolog

It is easy to turn this recognizer into a parser — and (unlike with `bottomup_recognizer.pl`) it's actually worth doing this, because it is efficient on small grammars. As is so often the case in Prolog, moving from a recognizer to a parser is simply a matter of adding additional arguments to record the structure we find along the way.

Here's the code. The ideas involved should be familiar by now. Read what is going on in the fourth argument position declaratively:

```
:- op(700,xfx,--->).
```

```
parse_topdown(Category,[Word|Reststring],Reststring,[Category,Word]) :-
        /*** Uncomment the following lines to see the steps the top
             down parser takes ***/
        %%% nl, write('String to recognize: '), write([Word|Reststring]),
        %%% nl, write('Category to recognize: '), write(Category),
        lex(Word,Category).

parse_topdown(Category,String,Reststring,[Category|Subtrees]) :-
        Category ---> RHS,
        /*** Uncomment the following lines to see the steps the top
             down parser takes ***/
        %%% nl, write('Rule: '), write((Category ---> RHS)),
        matches(RHS,String,Reststring,Subtrees).

matches([],String,String,[]).

matches([Category|Categories],String,RestString,[Subtree|Subtrees]) :-
        parse_topdown(Category,String,String1,Subtree),
        matches(Categories,String1,RestString,Subtrees).
```

And here's the new driver that we need:

```
parse_topdown(String,Parse) :-
        parse_topdown(s,String,[],Parse).
```

Time to play. Here's a simple example:

```
parse_topdown([vincent,fell],Parse).
```

And another one:

```
parse_topdown([vincent,shot,marsellus],Parse).
```

And here's a much harder one:

```
parse_topdown([jules,believed,the,robber,who,shot,the,robber,who,shot,the,robber,wh
```

As this last example shows, we really need a pretty-print output!

6.8 The Code

See file `bottomup_recognizer.pl`.	The naive bottom-up recognizer
See file `bottomup_recognizer_tests.pl`.	Input for testing
See file `topdown_recognizer.pl`.	The (less naive :-) top down recognizer
See file `topdown_recognizer_tests.pl`.	...
See file `topdown_parser.pl`.	...
See file `topdown_parser_tests.pl`.	...
See file `ourEng.pl`.	The English grammar we discussed in the lecture
See file `aNbN.pl`.	A grammar which generates $a^n b^n \backslash\{\}$
See file `epsilon.pl`.	A grammar with an empty production; `s --> []`
See file `leftrec.pl`.	A left-recursive grammar for $a^n b^n \backslash\{\}$

6.9 Practical Session

6.9.1 Bottom Up

Play with the bottom-up recognizer (defined in See file `bottomup_recognizer.pl`.). Make sure you thoroughly understand how it works. In particular, make sure you understand

- the way append/3 is used to divide the input list into three sublists;
- why it is so slow!

We have provided two grammars for you to test it with, namely:

See file `ourEng.pl`.	The English grammar we discussed in the lecture.
See file `aNbN.pl`.	Which generates $a^n b^n \backslash \{\}$.

We have also provided a file

See file `bottomup_recognizer_tests.pl`.

which contains examples for you to cut-and-paste to help you test.

6.9.2 Top Down

- Make sure you thoroughly understand how the top-down recognizer (*See file* `topdown_recognizer.pl`.) works. Test it with `ourEng.pl`. You will find examples to cut-and-paste in *See file* `topdown_recognizer_tests.pl`.. Compare its performance with the naive bottom-up recognizer from the last section.
- Make sure you thoroughly understand how (*See file* `topdown_parser.pl`.) works. Test it with the `ourEng.pl` You will find examples to cut-and-paste in *See file* `topdown_parser_tests.pl`..
- Extend `ourEng.pl` so that noun phrases may contain adjectives. E.g. ‘The dangerous robber died’ should be accepted by the grammar.

6.10 Exercises

6.10.1 Bottom-Up

Exercises for bottom-up parsing.

Exercise 6.1 *Design a simple ambiguous context free grammar. Show that your grammar is ambiguous by giving an example of a string that has two distinct parse trees. Draw the two trees.*

Exercise 6.2 *The following grammar (See file* `epsilon.pl`*.) contains a rule producing an empty terminal symbol:* $S \rightarrow \varepsilon$ *written as* `s -->[]` *in our Prolog notation.*

```
s ---> [].
s ---> [left,s,right].

lex(a,left).
lex(b,right).
```

Does this grammar admit the string a b*? What happens if you try to use* `bottomup_recognizer.pl` *with this grammar to recognize the string* a b*? Why?*

Exercise 6.3 *Our naive bottom-up recognizer only recognizes strings of category* `s`*. Change it, so that it recognizes strings of any category, as long as the whole string is spanned by that category, and returns the category that it found. For example:*

```
?- recognize_bottomup_any([the,man],Cat).
Cat = np
yes
?- recognize_bottomup_any([the,man,dances],Cat).
Cat = s
yes
?- recognize_bottomup_any([the,man,dances,a],Cat).
no
```

6.10.2 Top-Down

Exercises for top-down parsing.

Exercise 6.4 *Using the See file `ourEng.pl`. grammar, give a detailed top-down* depth-first *analysis of* Marsellus knew Vincent loved Mia*. That is, start with:*

		State
1.	`s`	marsellus knew vincent loved mia

and then show all the steps of your work, including the backtracking.

Exercise 6.5 *Using the See file `ourEng.pl`. grammar, give a detailed top-down* breadth-first *analysis of* Marsellus knew Vincent loved Mia*. That is, start with:*

	State
1.	{⟨ `s`, marsellus knew vincent loved mia⟩}

and then show all the steps of your work.

Exercise 6.6 *Can See file `topdown_recognizer.pl`. or See file `topdown_parser.pl`. handle the following grammar (See file `leftrec.pl`.)?*

```
s ---> [left,right].
left ---> [left,x].
left ---> [x].
right ---> [right,y].
right ---> [y].

lex(a,x).
lex(b,y).
```

If you are not sure try it. Why does it not work? How can it be fixed?

6.10.3 Midterm Project

This could be your midterm project

Exercise 6.7 *As we said in Section 6.7, we need a pretty printer that translates list representations of parses (like* `[s, [np, [pn, vincent]], [vp, [tv, shot], [np, [pn, marsellu` *to trees. As an example, see a pretty printer*[1] *for so called semantic tableaux that was written in Prolog. Just select some example from the boxes and then click on "send". Here, the output is a html-table. Write some pretty printer for the parses we have seen before (it might also produce latex or plain ascii or whatever format you like).*

---

[1] `http://www.coli.uni-sb.de/~stwa/Tableaux/tableaux.cgi`

7

Left-Corner Parsing

7.1 Introduction Left-Corner Parsing

7.1.1 Going Wrong with Top-down Parsing

Assume that we have the following grammar fragment

$S \longrightarrow NP\ VP$
$NP \longrightarrow Det\ N$
$NP \longrightarrow PN$
$VP \longrightarrow IV$
$Det \longrightarrow the$
$N \longrightarrow robber$
$PN \longrightarrow Vincent$
$IV \longrightarrow died$

and that we want to use it to top-down recognize the string ‘vincent died’. Proceeding in a top-down manner, we would first expand *S* to *NP VP*. Next we would check what we can do with the *NP* and find the rule *NP* ⟶ *Det N*. We would therefore expand *NP* to *Det N*. Then we have to find a phrase structure rule to expand *Det* or to find a lexical rule to relate *vincent* to the category *Det*. Neither is possible, so we would backtrack checking whether there are any alternative decisions somewhere.

Ignoring the Given Data

So, when recognizing in a top-down manner, we totally ignore what the actual input string looks like. We start from some non-terminal symbol and then use rules to rewrite that symbol. Only when we can’t apply any rules anymore, we check whether what we have produced matches with the input string.

Here is part of the trace that we will get when trying to recognize *vincent died* with the top-down recognizer of the previous section. You can see how Prolog first tries to use the first *NP* rule *NP* ⟶ *Det N* to expand the noun phrase. And only when Prolog realizes that *Det* leads into a dead-end does it try the next *NP* rule *NP* ⟶ *Det N*.

```
Call: (7) recognize_topdown(s, [vincent, died], []) ?
Call: (8) matches([np, vp], [vincent, died], []) ?
Call: (9) recognize_topdown(np, [vincent, died], _G579) ?
```

```
Call: (11) recognize_topdown(det, [vincent, died], _G585) ?
Fail: (11) recognize_topdown(det, [vincent, died], _G585) ?
Call: (11) recognize_topdown(pn, [vincent, died], _G582) ?
Exit: (11) recognize_topdown(pn, [vincent, died], [died]) ?
Exit: (9) recognize_topdown(np, [vincent, died], [died]) ?
Call: (10) recognize_topdown(vp, [died], _G582) ?
.
.
.
```

7.1.2 Going Wrong with Bottom-up Parsing

As we have seen in the previous example, top-down parsing starts with some goal category that it wants to recognize and ignores what the input looks like. In bottom-up parsing, we essentially take the opposite approach: We start from the input string and try to combine words to constituents and constituents to bigger constituents using the grammar rules from right to left. In doing so, any possible "child" consituents that can be built are built; no matter whether they eventually fit into a suitable "mother" constituent (an *S* in the end).

Ignoring the Overall Goal

No *top-down* information of the kind ‘we are at the moment trying to built a sentence’ or ‘we are at the moment trying to built a noun phrase’ is taken into account. Let’s have a look at an example.

Say, we have the following grammar fragment:

S ⟶ *NP VP*
NP ⟶ *Det N*
VP ⟶ *IV*
VP ⟶ *TV NP*
TV ⟶ *plant*
IV ⟶ *died*
Det ⟶ *the*
N ⟶ *plant*

Note, how `plant` is ambiguous in this grammar: it can be used as a common noun or as a transitive verb. If we now try to bottom-up recognize ‘the plant died’, we would first find that `the` is a determiner, so that we could rewrite our string to `Det plant died`. Then we would find that `plant` can be a transitive verb giving us `Det TV died`. `Det` and `TV` cannot be combined by any rule. So, `died` would be rewritten next, yielding `Det TV IV` and then `Det TV VP`. Here, it would finally become clear that we took a wrong decision somewhere: nothing can be done anymore and we have to backtrack. Doing so, we would find that `plant` can also be a noun, so that `Det plant died` could also be rewritten as `Det N died`, which will eventually lead us to success.

7.1.3 Combining Top-down and Bottom-up Information

As the previous two examples have shown, using a pure top-down approach, we are missing some important information provided by the words of the input string which would help us to guid our decisions. However, similarly, using a pure bottom-up approach, we can sometimes end up in dead ends that could have been avoided had we used some bits of top-down information about the category that we are trying to build.

The key idea of left-corner parsing is to combine top-down processing with bottom-up processing in order to avoid going wrong in the ways that we are prone to go wrong with pure top-down and pure bottom-up techniques. Before we look at how this is done, you have to know what is the left corner of a rule.

The Left Corner

The *left corner* of a rule is the first symbol on the right hand side. For example, *NP* is the left corner of the rule $S \longrightarrow NP\ VP$, and *IV* is the left corner of the rule $VP \longrightarrow IV$. Similarly, we can say that *vincent* is the left corner of the lexical rule $PN \longrightarrow vincent$.

A left-corner parser alternates steps of bottom-up processing with top-down predictions. The *bottom-up* processing steps work as follows. Assuming that the parser has just recognized a noun phrase, it will in the next step look for a rule that has an *NP* as its left corner. Let's say it finds $S \longrightarrow NP\ VP$. To be able to use this rule, it has to recognize a *VP* as the next thing in the input string. This imposes the *top-down* constraint that what follows in the input string has to be a verb phrase. The left-corner parser will continue alternating bottom-up steps as described above and top-down steps until it has managed to recognize this verb phrase, thereby completing the sentence.

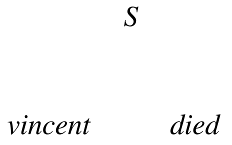
A left-corner parser starts with a top-down prediction fixing the category that is to be recognized, like for example *S*. Next, it takes a bottom-up step and then alternates bottom-up and top-down steps until it has reached an *S*.

To illustrate how *left-corner parsing* works, let's go through an example. Assume that we again have the following grammar:

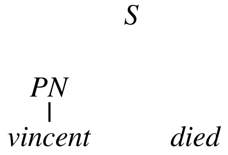
$S \longrightarrow NP\ VP$
$NP \longrightarrow Det\ N$
$NP \longrightarrow PN$
$VP \longrightarrow IV$
$Det \longrightarrow the$
$N \longrightarrow robber$
$PN \longrightarrow Vincent$
$IV \longrightarrow died$

Now, let's look at how a left-corner recognizer would proceed to recognize `vincent died`.

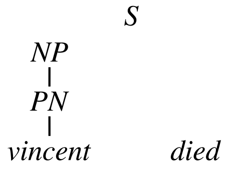
1.) Input: `vincent died`. Recognize an *S*. (Top-down prediction.)



2.) The category of the first word of the input is *PN*. (Bottom-up step using a lexical rule.)

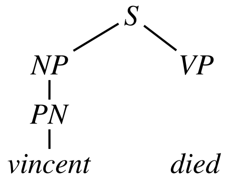


3.) Select a rule that has *PN* at its left corner: *NP* ⟶ *PN*. (Bottom-up step using a phrase structure rule.)



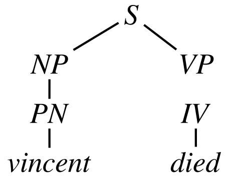
4.) Select a rule that has *NP* at its left corner: *S* ⟶ *NP VP*. (Bottom-up step.)

5.) Match! The left hand side of the rule matches with *S*, the category we are trying to recognize.



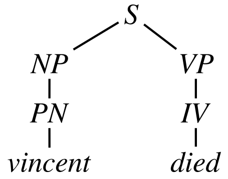
6.) Input: `died`. Recognize a *VP*. (Top-down prediction.)

7.) The category of the first word of the input is *IV*. (Bottom-up step.)



8.) Select a rule that has *IV* at its left corner: *VP* ⟶ *IV*. (Bottom-up step.)

9.) Match! The left hand side of the rule matches with *VP*, the category we are trying to recognize.



Make sure that you see how the steps of bottom-up rule application alternate with top-down predictions in this example. Also note that this is the example that we used earlier on for illustrating how top-down parsers can go wrong and that, in contrast to the top-down parser, the left-corner parser doesn't have to backtrack with this example.

7.2 A Left-Corner Recognizer in Prolog

Now, we will put the strategy that we have just described into Prolog. We will show how to implement a recognizer (see See file `leftcorner_recognizer.pl`.). One of the exercises will ask you to change this recognizer into a parser. As in the implementation of the top-down recognizer, we will use difference lists to keep track of how much of the input we have worked through.

The main predicate is `leftcorner_recognize/3`. It takes the following arguments: 1.) the category that is to be recognized, 2.) the input string, 3.) what's left of the input string after a starting sequence that belongs to the category specified in the first argument has been cut off. `leftcorner_recognize/3` first looks up the category of the first word of the input, `WCat`. It then calls `complete/4` which tries to close the hole between `WCat` and `Cat`. `leftcorner_recognize/3` evaluates to true, if it is possible to build a parse tree that has `Cat` at its root and is spanning a prefix of the input, such that `StringOut` is left.

```
leftcorner_recognize(Cat,[Word|StringIn],StringOut) :-
        lex(Word,WCat),
        complete(Cat,WCat,StringIn,StringOut).
```

`complete/4` has four arguments: 1.) `Cat`, the category that we are trying to recognize, 2.) `WCat`, the category that we already have recognized (it has to be incorporated into the left part of the parse tree), 3.) the input string that has not been recognized, yet, and 4.) what is left of the input string, once we have managed to recognized `Cat`.

In case the first and second argument are the same category (i.e., the category that we are trying to recognize is the same as the category that we have already found), we have completed that part of the parse tree.

```
complete(Cat,Cat,String,String).
```

If that is not the case, we need a recursive clause. This clause looks for a rule that has the second argument (the category that we have already recognized) as its left corner. It then calls `matches/3`, which will try to recognize strings of the other categories on the right hand side of the rule. If that's successfull, the recursive call of `complete/4` checks whether we already managed to recognize `Cat` or whether another left-corner bottom-up step has to be made to fill the hole that's left between `Cat` and `LHS`.

```
complete(Cat,SubCat,StringIn,StringOut) :-
        LHS ---> [SubCat|Cats],
        matches(Cats,StringIn,String1),
        complete(Cat,LHS,String1,StringOut).
```

`matches/3` finally checks whether the input begins with a certain sequence of categories and returns what's left of the input. The way it works and the way it is implemented is very similar to the `matches` predicate that we saw with the top-down recognizer.

```
matches([],String,String).
matches([Cat|Cats],StringIn,StringOut) :-
        leftcorner_recognize(Cat,StringIn,String1),
        matches(Cats,String1,StringOut).
```

And, lastly, here is a driver predicate:

```
leftcorner_recognize(String) :-
        leftcorner_recognize(s,String,[]).
```

7.3 Using Left-corner Tables

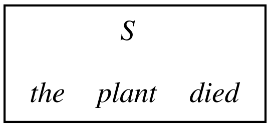
This left-corner recognizer handles the example that was problematic for the pure top down approach much more efficiently. It finds out what is the category of *vincent* and then doesn't even try to use the rule *NP* ⟶ *Det N* to analyze this part of the input. Remember that the top-down recognizer did exactly that.

But how about the example that was problematic for the bottom-up approach? Here, we don't see any improvement, yet. Just like the bottom up recognizer, the left-corner recognizer will first try to analyze *plant* as a transitive verb. Let's see step by step what the left-corner recognizer defined above does to process *the plant died* given the grammar

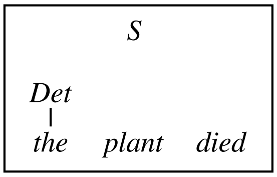
S ⟶ *NP VP*
NP ⟶ *Det N*
VP ⟶ *IV*
VP ⟶ *TV NP*
TV ⟶ *plant*
IV ⟶ *died*
Det ⟶ *the*
N ⟶ *plant*

We will only show how the parse tree developes.

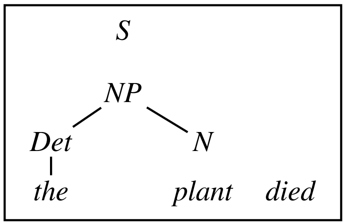
1.)



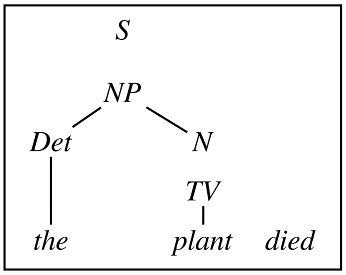
2.)



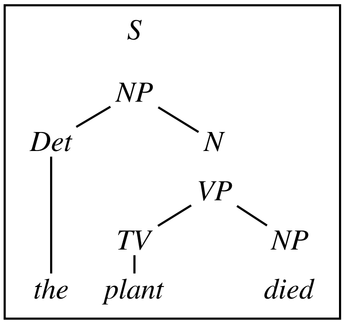
3.)



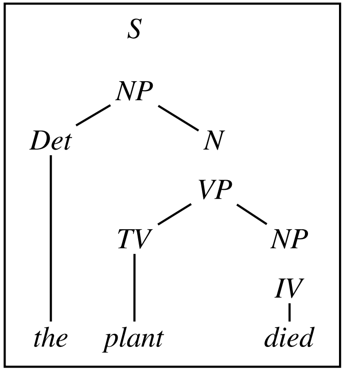
4.)



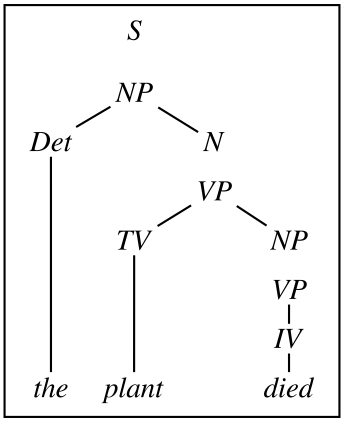
5.)



6.)



7.)



No way to continue! Backtracking!

So, just like the bottom-up recognizer, the left-corner recognizer chooses the wrong category for *plant* and needs a long time to realize its mistake. However, the left-corner recognizer provides the information that the constituent we are trying to build at that point is a noun. And nouns can never start with a transitive verb according to the grammar we were using. If the recognizer would use this information, it would notice immediately that the lexical rule relating *plant* to the category *transitive verb* cannot lead to a parse. The solution is to record this information in a table. This *left-corner table* stores which constituents can be at the left-corner of which other constituents. For the little grammar of the problematic example the left-corner table would look as follows:

S	*NP*, *Det*, *S*
NP	*Det*, *NP*
VP	*IV*, *TV*, *VP*
Det	*Det*
N	*N*
IV	*IV*
TV	*TV*

Note, how every category can be at the left corner of itself. In Prolog, we will simply store this table as facts in the knowledge base (`lc(A,B)` reads `A` is a possible left corner of `B`):

```
lc(np,s).
lc(det,np).
lc(det,s).
lc(iv,vp).
lc(tv,vp).

lc(X,X).
```

Now, we can check every time that we choose a rule whether it makes sense to use this rule given the top-down information about the category that we are trying to recognize.

So, every time we decided on a category for a word, we first look into the left-corner table to check whether the category of the word can appear at the left corner of the constituent we are working on:

```
leftcorner_recognize(Cat,[Word|StringIn],StringOut) :-
        lex(Word,WCat),
        lc(WCat,Cat),
        complete(Cat,WCat,StringIn,StringOut).
```

Similarly, we check that the left-hand sides of the rules we are using can appear as the left corner of the category we are building:

```
complete(Cat,SubCat,StringIn,StringOut) :-
        LHS ---> [SubCat|Cats],
        lc(LHS,Cat),
        matches(Cats,StringIn,String1),
        complete(Cat,LHS,String1,StringOut).
```

The other clause of `complete` is unchanged (see See file `leftcorner_recognizer_table.pl`.).

Now, go back to the example and check what happens. In step four, where the previous version of the algorithm made the mistake, this new version would check whether `lc(TV,N)` is true. Since this is not the case, it would immediately backtrack and find the second possibility for *plant* in the lexicon.

7.4 The Code

See file `leftcorner_recognizer.pl`.The left corner recognizer	
See file `leftcorner_recognizer_table.pl`.	The left corner recognizer using
See file `ourEng.pl`.	Our English grammar fragment
See file `leftrec.pl`.	A left recursive context free gra
See file `epsilon.pl`.	A grammar with an empty prod

7.5 Practical Session

1. Download See file `leftcorner_recognizer.pl`., the left-corner recognizer, and See file `ourEng.pl`., the English grammar fragment, and make sure that you understand how the left-corner algorithm works.
2. Write the left-corner table for See file `ourEng.pl`..
3. Now, use See file `ourEng.pl`. and the left-corner table that you have written to test See file `leftcorner_recognizer_table.pl`., the left-corner recognizer using a left-corner table. Make sure that you understand how the left-table is used.

7.6 Exercises

Exercise 7.1 *Download See file* `leftrec.pl`.*, the left-recursive grammar that we already saw in the previous chapter. Can the left-corner recognizer cope with it? Why does it not have the same problems with it as the top-down algorithm?*

Exercise 7.2 *Now, download See file* `epsilon.pl`*., the grammar with the empty production that we have seen before, as well. Write a left-corner table for it. (Note:* ε *is a kind of terminal symbol.).*

Exercise 7.3 *[This is tricky] Turn the left-corner recognizer into a parser by adding an extra argument to collect the structure.*

8

Passive Chart Parsing

8.1 Motivation

Suppose we're working with a grammar containing the following rules:

```
vp ---> [v,np].
vp ---> [v,np,pp].
vp ---> [v,np,vp].
```

That is, in this grammar there are three different ways of building VPs, and there is no distinction between the different types of verb (unlike as in our See file `ourEng.pl`. grammar).

Suppose we try to parse a sentence containing the substring 'sees vincent give the money to the drug dealer'. And suppose that we are working with a bottom-up recognizer such as See file `bottomup_recognizer.pl`. from Chapter 5.

Attempt 1a

We first analyze `sees` as being of category `v` and `vincent` as being of category `np`. This allows us to next use the rule `vp --> [v,np]` (see the little tree on the left hand side of the diagram below). Now we must go on and analyze the rest of the input string, 'give the money to the drug dealer'. We can do this: we successfully recognize it as a VP — but now we have a problem. We've got the following structure:

```
     vp               vp
    / \               |
   /   \              |
  /     \             |
 v       np           |
 |       |            |
 |       |            |
 |       |   ==================================
sees vincent give the money to the drug dealer
```

Two VPs standing next to each other isn't much use — we're going to have to backtrack and try something else. We do so, and — without thinking about it — we do something very silly: we throw away the fact that 'give the money to the drug dealer' is a VP.

Attempt 1b

This time we don't combine `sees` and `vincent` using the rule `vp --> [v,np]`, but first deal with the rest of the string. All goes well: we build the vp for 'give the money to the drug dealer' and then use the rule `vp --> [v,np,vp]` to build a VP for the whole string.

```
        vp
      / | \
     /  |  \
    /   |   \
   /    |    \
  /     |     \
 v      np    vp
 |      |      |
 |      |      |
 |      |      |
             =================================  SUCCEED
sees vincent give the money to the drug dealer
```

But note: to get this correct parse, we have to show that 'give the money to the drug dealer' is a VP. But we showed that in attempt 1a — and then threw this fact away when we backtracked! We're repeating work. This does not sound like a good idea.

And it is not: it really is very inefficient. Moreover it is an inefficiency of the *algorithm*, not the implementation. Naive algorithms really are naive — it it is silly to repeat work. And sometimes we will pay a price for being naive: no matter how cleverly we implement naive algorithms, on some grammars, and on some input strings, naive algorithms will give disastrous performance.

It is important to realize that this kind of inefficiency has nothing to do with the top-down versus bottom-up distinction. It doesn't matter which of these strategies we use, or even if we use a sophisticated mixture of these strategies, such as the left-corner parsing strategy. The basic point is the same: if we throw away the results of our work when we backtrack, we may have to redo this work later. And given that realistic grammars for natural language are not only highly ambiguous, but contain lots of local ambiguities as well, we may end up repeating the same work a huge number of times.

In this lecture we start our discussion of *chart parsing* . Chart parsing is based on an incredibly simple idea: 'Don't throw away information. Keep a record — a chart — of all the structure you have found.' Chart parsing is a very general idea, and it can be used with just about any kind of parsing (or recognition) strategy you care to think of (top down, bottom up, left-corner, depth first, breadth first, ...).

Today we are going to examine the simplest form of chart parsing, namely *passive chart parsing* . Roughly speaking, this means the chart is simply a record of all constituents that have been recognized. As we shall see in later lectures, we can use the chart in more interesting ways, and this leads to what is known as *active* chart parsing.

But today we'll stick with passive chart parsing. We'll discuss and then implement a simple bottom-up recognizer using a passive chart.

8.2 A Bottom-Up Recognizer Using a Passive Chart

8.2.1 An Example

In this section, we are going to see what a chart is and how it can be used in a bottom-up recognizer. That is, for simplicity we are going to do recognition, which means we're not actually going to build the parse trees. But it will be clear that all the information needed to carry out full parsing is there on the chart — the recognition algorithm takes us most of the way to a full parser.

But first things first — what is a chart anyway? Simply a record of what information we have found where. For example, suppose we were trying to parse the sentence 'Jules handed the gun to Marsellus'. Then a (very simple and incomplete) chart for this string is:

```
 -- pn --
|        |
|        |
0 Jules  1  handed  2  the  3  gun  4  to  5  Marsellus  6
```

This chart is made up of *nodes* and *edges* (or *arcs*). The nodes are the numbers 0,1,2,3,4,5, and 6. The nodes mark positions in the input — clearly every word is picked out by a pair of numbers. For example, the word `to` is the word between positions 4 and 5. Arcs join nodes, and their function is to tell us what structure we have recognized where. In the above chart there is only one edge, namely the one that goes from 0 to 1. We say that this edge *spans* 0,1. Note that this edge is labelled `pn`. Thus this edge tells us that between 0 and 1 we have found a proper name.

Passive chart parsing essentially proceeds by progressively adding more arcs to the chart as we discover more and more about the input string. The algorithm we are going to look at is a bottom-up algorithm. Hence, it starts from the concrete information given in the input string, and works its way upwards to increasingly more abstract levels (and in particular, to the sentence level) by making use of CFG rules right-to-left. In terms of arcs in the chart that means that we are going to use CFG rules right-to-left to combine arcs that are already present in the chart to form larger arcs. For example: suppose we have an arc going from node 2 to node 3 which is labelled `det` and an arc from 3 to 4 labelled `n` in our chart. Then the rule `np --> [det,n]` allows us to add an arc going from 2 to 4 which is labelled `np`.

Example is the best way to learn, so let's now look at a concrete bottom-up chart parsing algorithm. Suppose we are working with the See file `ourEng.pl`. grammar, and we want to analyze the sentence 'Vincent shot Marsellus'. How do we do so using the bottom-up algorithm? As follows. First, we start with the empty chart:

```
0 vincent  1  shot  2  marsellus 3
```

Then we read the first word (word 0,1) and build all arcs that our grammar and lexicon allow us to build for this word. First, we can use the lexical rule `lex(vincent,det)` to build an arc labelled `pn` going from position 0 to position 1. We add this arc to the chart and check whether we can maybe use it to build further arcs. And in fact, we have the rule `np --> [pn]`. Reading it from right to left, we can use the newly added arc to build a second arc from 0 to 1 which is labelled `np`. Now, there is no more that we can do: there are no other lexical entries for `vincent`, no more rules that have `[pn]` as their right hand side and no rules that have `[np]` as their right hand side. We therefore get the following chart:

```
  -- np --
|          |
|          |
  -- pn --
|          |
|          |
0 vincent  1  shot   2  marsellus 3
```

When nothing can be done for the first word anymore, we read the second word and all the arcs that we can build for that word and from combinations of the new arcs with arcs that are already in the chart. That is, we add all new arcs that can be build for the substring between 0 and 2. Given the grammar See file `ourEng.pl`. there is only one thing that we can do: add a `tv`-arc from position 1 to position 2. We get:

```
  -- np --
|          |
|          |
  -- pn --    -- tv --
|          |          |
|          |          |
0 vincent  1   shot   2  marsellus 3
```

Nothing more can be done for the substring between node 0 and node 2. So, we read the next word (word 2,3) and add all new arcs for the span 0,3.

The lexical rule `lex(marsellus,pn)` let's us add the `pn`-arc from 2 to 3. The we can use the rule `np --> [pn]` to build the `np`-arc between 2 and 3. This arc can then be combined with the `tv`-arc from 1 to 2 using the rule `vp --> [tv,np]`. Finally, the `vp`-arc which is created this way can be combined with the `np`-arc from 0 to 1 to form a sentence spanning 0 to 3 (`s --> [np,vp]`).

```
 ------------- s ------------------
|                                  |
|                                  |
|            --------- vp ---------
|          |                       |
|          |                       |
  -- np --              --- np ---
|          |          |            |
|          |          |            |
  -- pn --    -- tv --    --- pn ---
|          |          |            |
|          |          |            |
0 vincent  1   shot   2  marsellus 3
```

At this stage the algorithm halts. Why? Well — the chart contains all the arcs that can possibly be built for the string from 0 to 3 and there are no more words to be read. Pretty clearly, we have succeeded in recognizing that the input was a sentence. After all, the very last edge we added spans the entire input (that is, positions 0 to 3) and tells us that it has category `s`.

8.2.2 Being Complete

One central point we have not yet addressed explictly is: How can we be sure that we really find *all* possible edges spanning (parts of) our input. Well, the principle is simple. What we are doing is this:

1. We traverse our input string from left to right, and
2. We do not move on before we have done all that can be done at some point.

As a consequence, our chart is always complete to the left: All that could have been done there had been done. So, when we move on to some node x, we make sure that all possible edges between 0 and x are being added. Then we know, that the chart is complete up to node x when we move to x+1. As a consequence, we never have to backtrack to some earlier nodes to see whether there is anything left that has to be done. This strategy leads us systematically through the input without forgetting to add any edge.

8.3 Some Prolog Revision

8.3.1 Database manipulation

There is one key key decision that we have to make before starting to implement the recognizer and that is how we are going to represent the chart, and how we are going to work with it.

Now, there are various options here. For example, we could represent the chart using some sort of structured list. But there is another, more attractive, way to go: represent the chart using the Prolog database, and manipulate the chart (that is, add new edges) using the database manipulation operations. As we are going to be making heavy use of Prolog's database manipulations, let's quickly run through them and remember how they worked.

There are five database manipulation commands that we will using:

```
assert
retract
asserta
assertz
retractall
```

How are these used? Suppose we start with an empty database. So if we give the command:

```
?- listing.
yes
```

we get a yes — and the listing (of course) is empty.

Suppose we now give this command:

```
?- assert(happy(mia)).
yes
```

It succeeds (the `assert/1` commands *always* succeed). But what is important is not that it succeeds, but the side effect it has on the database, for if we now give the command:

```
?- listing.

happy(mia).
yes
```

we get again yes — but now there *is* something in the database, namely the fact we asserted.

Suppose we then made four more assert commands:

```
?- assert(happy(vincent)).
yes

?- assert(happy(marsellus)).
yes

?- assert(happy(butch)).
yes

?- assert(happy(vincent)).
yes
```

and then ask for a listing. Then we get:

```
?- listing.

happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).
yes
```

There is an inverse predicate to `assert/1`, namely `retract/1`. For example, if we go straight on and give the command:

```
?- retract(happy(marsellus)).
yes
```

and then list the database we get:

```
?- listing.

happy(mia).
happy(vincent).
happy(butch).
happy(vincent).
yes
```

Suppose we go on further, and say

```
?- retract(happy(vincent)).
yes
```

and then ask for a listing:

```
?- listing.

happy(mia).
happy(butch).
happy(vincent).
yes
```

Note that the *first* occurrence of happy(vincent) was removed. Prolog removes the first thing in the database that matches the argument of `retract`.

```
?- retract(happy(_)).
yes
?- listing.

happy(butch).
happy(vincent).
yes
```

The predicate `retractall/1` removes everything that matches its argument:

```
?- retractall(happy(_)).
yes
?- listing.
yes
```

If we want more control over where the asserted material is placed, there are two variants of assert, namely:

- `assertz`, which places stuff at the *end* of the database, and

- `asserta`, which places stuff at the *beginning* of the database.

Prolog puts some restrictions on what can be asserted and retracted. More precisely, other predicates are only allowed to remove or retract clauses of predicates that have been declared *dynamic* . So, if you want to write a predicate which asserts or retracts clauses of the form `happy(mia)`, you also have to put the following statement into your file:

```
:- dynamic happy/1.
```

Database manipulation is useful when we want to store the results of queries, so that if we need to ask the same question in future, we don't need to redo the work — we just look up the asserted fact. This technique is called 'memoization', or 'caching'. And of course, this 'memoization' idea is the basic concept underlying chart-parsing: we want to store information about the syntactic structure we find, instead of having to re-compute it all the time.

8.3.2 Failure Driven Loops

The idea of *failure driven loop* s is to *force Prolog to backtrack* until there are no more possibilities left. How can we force Prolog to backtrack? Well, Prolog automatically tries to backtrack when it fails. So, adding a goal which always fails (such as the built in predicate `fail/0` which has no effect but to make Prolog fail) at the end of a clause, will force Prolog to backtrack over all the other goals in that clause until it has tried all the possibilities. Here is an example:

```
write_everybody_happy :- happy(X),
                         write(X),nl,
                         fail.
write_everybody_happy :- true.
```

If you consult this little program and the ask Prolog the query `write_everybody_happy`, Prolog will look up in its database all objects that make `happy(X)` true and will print them out on the screen. So, if the database looked as follows

```
happy(harry).
happy(ron).
happy(hermione).
happy(hagrid).
```

Prolog would write on the screen

```
harry
ron
hermione
hagrid

yes
```

8.4 Passive Chart Recognition in Prolog

8.4.1 Representing the Chart

We'll now implement a bottom-up chart recognizer in Prolog. To do so, we first have to decide how to represent the chart. As already mentioned in the last section, we want to use the Prolog database for that. More explicitly, we will use the predicates `scan/3` and `arc/3`. The predicate `scan` encodes the position of the words. For example, we take `scan(2,3,loves)` to mean that the word *loves* is between positions 2 and 3 of the input sentence. The initial empty chart for the input sentence *vincent loves mia*, which doesn't contain any arcs, yet, thus looks like this in Prolog:

```
scan(0,1,vincent).
scan(1,2,loves).
scan(2,3,mia).
```

We will represent arcs using the predicate `arc/3` in the obvious way: `arc(0,2,np)` means that there is an `np`-arc between 0 and 2. So, the chart

```
 ------------- s ---------------
|                                |
|                                |
|            --------- vp ------
|          |                     |
|          |                     |
  -- np --              -- np --
|          |          |          |
|          |          |          |
  -- pn --   -- tv --   -- pn --
|          |          |          |
|          |          |          |
0 vincent  1   loves  2   mia    3
```

is represented in Prolog as

```
scan(0,1,vincent).
scan(1,2,loves).
scan(2,3,mia).
arc(0,1,pn).
arc(0,1,np).
arc(1,2,tv).
arc(2,3,pn).
arc(2,3,np).
arc(1,3,vp).
arc(0,3,s).
```

8.4.2 The Algorithm

Now, when wanting to recognize a sentence, the first thing we have to do is to initialize the chart, i.e., to write the appropriate `scan`-facts for the words into the Prolog database. This is done by the following predicate. It is a recursive predicate that works its way through the list representing the input sentence in the standard fashion of Prolog list processing predicates. For each word in the input it asserts a fact with the functor scan recording the position of that word.

```
initialize_chart([], _).

initialize_chart([Word|Input], From) :-
        To is From + 1,
        %%% assertz -> put new word at the end of the db
        assertz(scan(From, To, Word)),
        %%% write(scan(From, To, Word)), nl,
        initialize_chart(Input, To).
```

Then, we have to read the first word and add all the arcs that we can build from this first word. When nothing new can be added, we read the next word and add all arcs that we can build for the substring between 0 and 2. When no more can be done for that span, we read the next word and again add all new arcs that can be build for this new word. We continue like this until we reach the end of the sentence.

The main predicate of this part of the program is `process_bottomup/0`.

```
process_chart_bottomup :-
        doall(
             (scan(From, To, Word),
             %%% write('read word: '), write(Word),nl,
             lex(Word, Cat),
             add_arc(arc(From, To, Cat)))
            ).
```

It reads a word from the database (`scan(From, To, Word)`) and looks up its category in the lexicon (`lex(Word, Cat)`). The real work is done by calling `add_arc/1`. This predicate adds the new arc and all arcs that can be built from it to the chart.

Then, `doall/1` forces backtracking: `process_bottomup/0` backtracks to use other lexical entries for the word under consideration if there are any and to read the next word if there are none. `doall/1` implements a failure driven loop. It forces Prolog to backtrack over its argument. `doall/1` is implemented as follows:

```
doall(Goal) :- Goal, fail.

doall(_) :- true.
```

`add_arc/1` takes an arc as argument. If that arc is not yet in the chart, `add_arc` adds it. It then calls `new_arcs/1`, which constructs and adds all the arcs that can be build from combining the newly added arc with what is already in the chart.

```
add_arc(Arc) :-
        \+ Arc,
        assertz(Arc),
        %%% write(Arc),nl,
        new_arcs(Arc).
```

`new_arcs/1` also uses the failure driven loop predicate `doall/1` — we want to find *all* the new arcs that can be built. We do this by looking for suitable edges to our left in the chart. As we have explained in Section 8.2.2, the chart is complete to the left: All potential arcs to combine with can only be on our left.

For example if our newly added category is an `np`, we try to find rules having an NP as rightmost part of the right hand side (remember we are using a bottom-up strategy), e.g. `vp --> [tv,np]`. Then we try to find the left part of the right hand side (`tv` in this example) in the database (and hence in the chart on our left).

```
new_arcs(arc(J, K, Cat)) :-
        doall(
              (LHS ---> RHS,
              append(Before, [Cat], RHS),
              path(Before, I, J),
              add_arc(arc(I, K, LHS)))
             ).
```

`new_arcs` takes an arc `arc(J,K,Cat)` as argument. To find new arcs, we take a rule from the grammar and check a) whether `Cat` is the last category on the right hand side of that rule and b) whether there is a sequence of arcs in the chart that spans `I,J` and is labelled with the categories that come on the right hand side of the rule before `Cat`. If that is the case, a recursive call of `add_arc` adds the new arc /the one spanning `I,J` and labelled with the left hand side of the rule) to the chart and checks whether it can be used to build any further arcs.

Let's go back to our example from Section 8.2.1. Suppose we have just added the `vp` and called `new_arcs(arc(1, 3, vp))`.

```
             --------- vp ---------
           |                       |
           |                       |
  -- np --              --- np ---
|          |          |            |
|          |          |            |
  -- pn --   -- tv --   --- pn ---
|          |          |            |
|          |          |            |
0 vincent  1   shot   2  marsellus 3
```

We then find the rule `s --> [np,vp]`, so `Before` is instantiated with `[np]`. Next, `path([np],I,1)` checks whether there is such a path. Indeed there is: `I` is instantiated with 0. So we add the arc `arc(0,3,s)` to the chart (and recursively check whether there is anything else that has to be done).

It only remains to define `path/3`. It's first argument is a list of categories, and the second and third arguments are nodes. The predicate recursively checks if an arc, or a sequence of arcs, links the two input nodes:

```
path([], I, I).

path([Cat|Cats], I, K) :-
        arc(I, J, Cat),
        J =< K,
        path(Cats, J, K).
```

Now, we have defined the predicates that build the chart. What is missing is a predicate the calls them and then checks whether the final chart contains an arc that spans the whole input sentence and is labelled with `s`. Here it is:

```
chart_recognize_bottomup(Input) :-
        cleanup,
        initialize_chart(Input, 0),
        process_chart_bottomup,
        length(Input, N),
        arc(0, N, s).
```

The first thing this predicate does is to clean the chart. This is *very* important. Our recognizer works by asserting stuff into the database. This means we need to get rid of all the stuff that was asserted to the database before we try to recognize a new sentence — for otherwise all the old stuff will be hanging around, and may interfere with the analysis of the new sentence. In short, we need to retract all the information asserted the last time we used the recognizer before we use it again. And this is exactly what `cleanup/0` does for us:

```
cleanup :-
        retractall(scan(_,_,_)),
        retractall(arc(_,_,_)).
```

After cleaning the database up, `chart_recognize_bottomup/1` initializes the chart with the input sequence, and processes the chart. Finally, the only thing to be done is check whether there is an arc labled `s` that is spanning the whole input.

The whole program can be found in See file `passive_chart_bottomup.pl`..

8.4.3 An Example

Let's look at an example. Let's see what happens when we give this recognizer the example we looked at before, that is, "Vincent shot Marsellus". Here is an abbreviated trace that shows you how the words are read and which arcs are added to the chart subsequently.

```
[trace] 12 ?- chart_recognize_bottomup([vincent,shot,marsellus]).
   Exit: (11) scan(0, 1, vincent) ?
   Exit: (11) assert(arc(0, 1, pn)) ?
   Exit: (15) np--->[pn] ?
   Exit: (15) assert(arc(0, 1, np)) ?
   Exit: (11) scan(1, 2, shot) ?
   Exit: (11) assert(arc(1, 2, tv)) ?
   Exit: (11) scan(2, 3, marsellus) ?
   Exit: (11) assert(arc(2, 3, pn)) ?
   Exit: (15) np--->[pn] ?
   Exit: (15) assert(arc(2, 3, np)) ?
   Exit: (19) vp--->[tv, np] ?
   Exit: (19) assert(arc(1, 3, vp)) ?
   Exit: (23) s--->[np, vp] ?
   Exit: (23) assert(arc(0, 3, s)) ?

Yes
```

And this is what the final chart looks like:

```
13 ?- listing(scan).

scan(0, 1, vincent).
scan(1, 2, shot).
scan(2, 3, marsellus).

Yes
14 ?- listing(arc).

arc(0, 1, pn).
arc(0, 1, np).
arc(1, 2, tv).
arc(2, 3, pn).
arc(2, 3, np).
arc(1, 3, vp).
arc(0, 3, s).

Yes
```

8.5 The Code

See file `passive_chart_bottomup.pl`.	The bottom-up chart recognizer.
See file `ourEng.pl`.	Our well-known English grammar fragment.

8.6 Exercise

Exercise 8.1 *Today's exercise is to write a pretty-printer for charts. Remember that the chart is still in the database after the parse:* `chart_recognize_bottomup([vincent,shot,marsellu`

Write a failure-driven loop (page 126) that reads out the chart from the database and prints it out (as in our examples in the text, or upside down - as you like it).

[Hints:]

Exercise 8.2 *[Mid Term Project]*

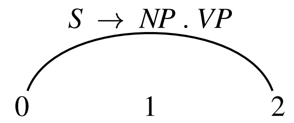
In !!!UNEXPECTED PTR TO EX_CHARTPARSING.EX.1!!!, the chart was printed after parsing. Alternatively, you can print out the chart incrementally during the parsing process. Modify the respective predicates in See file `passive_chart_bottomup.pl`*.. If the visualization is too fast, you can slow it down using the predicate* `sleep/1`*:* `write(a),flush_output,sleep(2),nl,write(b).` *The statement* `flush_output` *is important for incremental output. If you leave it away, Prolog might buffer the output some and print it all at once:* `write(a),sleep(2),nl,write(b).`

Active Chart Parsing

9.1 Active Edges

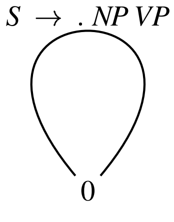
In this lecture we move from passive chart parsing to active chart parsing. Active chart parsing is based on a very simple idea. A passive chart, like the one we worked with in the previous chapter, is used to keep a record of *complete* constituents that we have found. (For example, on a passive chart we can record the information that the string between nodes 2 and 4 is an NP.) In an active chart we additionally record *hypotheses* — that is, we record what it is we are actually looking for, and how much of it we have found so far. Such information is recorded in *active edge* s (or *active arc* s).

Here's an example of an active edge:



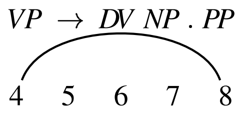
This is an arc between nodes 0 and 2. The important thing to note is the arc label: *S → NP . VP* This says: '*I am trying to build an s, consisting of an np followed by a vp. So far I have found an np arc, and I'm still looking for the vp.* ' So the insignificant looking '.' in the middle of the rule is very important: it marks the boundary between what we have found so far, and what we are still looking for.

Here's another example:



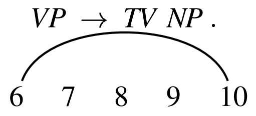
This says: '*I am trying to build an s, consisting of an np followed by a vp, starting at node 0. So far I have neither found the np nor the vp that I need.*'

Here's another example:



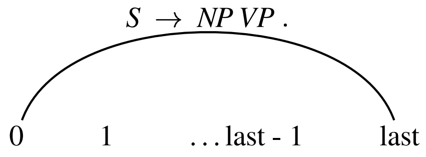
This says: ‘*I am trying to build a vp, consisting of a dv followed by an np followed by a pp, starting at node 4. So far I have found a dv arc and an np arc, and I’m still looking for the pp.*’ The `np`-arc ends in node 8, so the `pp`-arc will have to start in node 8.

Here’s another example:



This says: ‘*I am trying to build a vp consisting of a tv followed by an np starting at node 4 — and I’ve done it!*’ Note that the ‘.’ is at the *end* of the arc label. This means that everything we need to find has been found. Such an arc is called a *passive edge* , or a *passive arc* . It records *complete* information. In effect, these are the arcs we are used to from the previous chapter, though we didn’t use the ‘.’ notation there.

One final example:

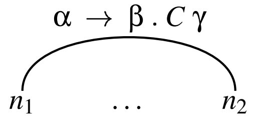


This says: ‘*I am trying to build an s consisting of an np and a vp starting at node 0 — and I’ve done it! I’ve found both the np and the vp, and moreover, I’ve done it in a way that the arc goes all the way from the first to the last node. This means I’ve recognized the sentence!*.’

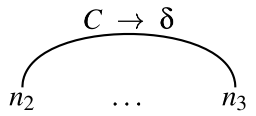
9.2 The Fundamental Rule

Active arcs are called ‘active’ for a very good reason: They ‘actively’ hint at what can be done with them. In particular, we can see at once how to combine them with passive edges to make new edges. The new edges we make may be either passive or active.

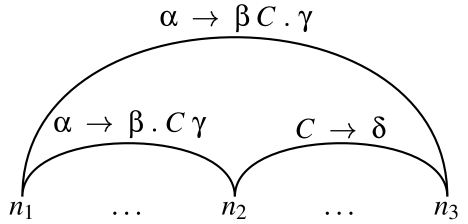
The *fundamental rule* for combining a passive edge and an active edge works as follows: Suppose the active edge goes from node n_1 to node n_2 and has category C immediately to the right of the dot.



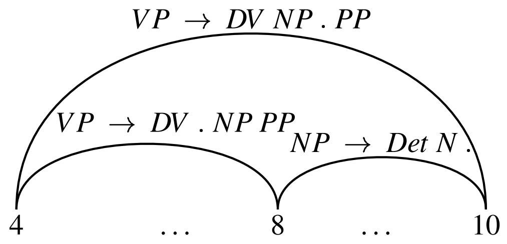
Further, suppose that the passive edge goes from node n_2 to node n_3 (hence, it starts where the active edge ends) and has category C on its left hand side.



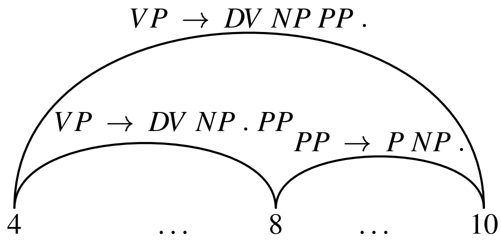
The fundamental rule now allows us to combine these two edges to build a new one that starts in node n_1 and ends in n_3. For the label of that new edge we move the dot in the active edge one category forward.



Here are two examples to illustrate the fundamental rule.



Here we have an active edge (the one on the left) which is looking for a np. And immediately to its right we have an np (note that the edge to the right is passive: that is we really know that there is a np between nodes 8 and 10). Using the fundamental rule, we can combine these edges to build a new edge going from node 4 to node 10. Note, that this new edge is active.



Here we have an active edge (the one on the left) which is looking for a pp. And immediately to the right we have a pp (note that the edge to the right is passive: that is we really know that there is a pp between nodes 8 and 10). Using the fundamental rule, we can combine these edges. Note, that in this case the new edge we have built is passive.

9.3 Using an Agenda

Active chart parsers usually make use of an *agenda* . An agenda is a datastructure that keeps track of the things that we still have to do. When new edges are created, we have to remember that we have to look at them to see whether they can be combined with other edges in any way. To not forget this we store them in the agenda. (They are not added directly to the chart as with the passive chart parser.) We will then take one edge at a time from the agenda, add it to the chart, and then use it to build new edges.

You can think of the agenda as a *list* of edges. When we take edges from the agenda, we will always take them from the beginning of the list. So, by modifying where we insert new edges into this list, we change the order in which edges are processed.

We are going to treat the agenda as a *stack*. That is, we add new edges to the *front* of the agenda. This leads to a depth-first search strategy. Another possibility would be to add new edges to the *end* of the agenda. The corresponding datastructure is called *queue* and would lead to breadth-first search. Finally, you could also order the elements in the agenda due to some other criterion, such as, e.g., how probable it is that the edges will lead to a parse.

9.4 A General Algorithm for Active Chart Parsing

A nice thing about using an agenda is that it is straightforward to specify a very general algorithm for active chart parsing. We'll now present this algorithm. With only minor changes — or to be more accurate, simply by being a bit more precise about how we are going to carry out the individual steps given below — it is very easy to convert this general algorithm into (say) a top-down or bottom-up active charting parsing algorithm.

Here's the general algorithm:

1. Make initial chart and agenda.
2. Repeat until agenda is empty:
    (a) Take first arc from agenda.
    (b) Add arc to chart. (Only do this if edge is not already on the chart!)

(c) Use the fundametal rule to combine this arc with arcs from the chart. Any edges obtained in this way should be added to the agenda.

(d) Make hypotheses (i.e., active edges) about new constituents based on the arc and the rules of the grammar. Add these new arcs to the agenda.

End repeat

3. See if the chart contains a *passive* edge from the first node to the last node that has the label s. If 'yes', succeed. If 'no', fail.

This algorithm is very general; especially step (2d). By being more precise about when and how we predict new hypotheses it is easy to try out different parsing strategies. We'll soon see how 2c is carried out in the bottom-up case, and later we'll learn how it is carried out in the top-down case.

Actually, it's not only the way 2c is handled that leads to different concrete algorithms. We also have to initialize the agenda slightly differently in the bottom-up and top-down cases. That is, there are different ways of carrying out step 1.

9.5 Bottom-up Active Chart Parsing

9.5.1 An Example

Suppose we want to analyze the sentence "Mia danced" using the following grammar.

$S \rightarrow NP\ VP$
$S \rightarrow NP\ VP\ PP$
$NP \rightarrow PN$
$VP \rightarrow IV$
$PP \rightarrow P\ NP$
$PN \rightarrow mia$
$IV \rightarrow danced$

First, we have to initialize the chart and the agenda (step 1 of the general algorithm (page 136)). The chart initialization works exactly as for the *passive* chart parser. So, the initial chart for the sentence *Mia danced* looks like this:

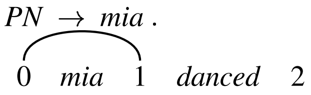
0 *mia* 1 *danced* 2

The initial agenda looks like this:

1. $\langle 0, 1, PN \rightarrow mia\ .\rangle$
2. $\langle 1, 2, IV \rightarrow danced\ .\rangle$

For all words of the input we have added passive edges saying of which category they are and what their position in the sentence is. The edge $\langle 0, 1, PN \rightarrow mia\ .\rangle$, for instance, is a passive edge (indicated by the dot at is end) that says that there is a *PN* between positions 0 and 1. This edge was added because we have the rule $PN \rightarrow mia$ in our grammar.

Now that the initialization process is over, we get to step 2, the 'repeat' part of our general algorithm (page 136). The first thing we do is take the first arc of the agenda (step 2a). In our example that's $\langle 0, 1, PN \rightarrow mia\ .\rangle$. We then check whether it is aready recorded in the chart (step 2b). That's not the case, so we add it:



Next, we try to apply the fundamental rule to the new arc (step 2c). The fundamental rule lets us combine an active arc with a passive arc that is immediately to the right of the active one. Since the arc we are working with is passive, we have to find an active arc to its left. Or more precisely, we have to find an active arc ending in position 0. There is no such arc, so we cannot do anything here. We go on to step 2d.

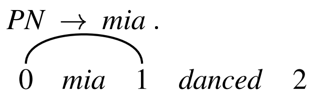
The general algorithm (page 136) tells us that we have to build new hypotheses in this step. Let's see what that means for the bottom-up case. In bottom-up parsing, we always build the parse tree from the bottom upwards, i.e., we combine already completed constituents to form bigger ones. Therefore, we only apply step 2d to passive arcs when working bottom-up; we only predict new constituents based on already completed ones.

In our example we are dealing with a passive arc at the moment (namely with $\langle 0, 1, PN \rightarrow mia\rangle$). Making hypotheses then works as follows. We look for grammar rules whose left corner (i.e. the the first symbol on their right hand side, see Section 7.1.3) is the same as the left hand side of the arc under consideration (which is a *PN* in our case). $NP \rightarrow PN$ is such a rule, for example. We then build active edges for all of these rules. These edges start in the same position as the arc we were looking at, end in exactly that same position, and have the dot right after the arrow. Like this: $\langle 0, 0, NP \rightarrow\ .\ PN\rangle$. This edge is the hypothesis that we might be able to build an *NP* starting in 0. To do so we still have to find a *PN*, also starting in 0. As we have already found a *PN* starting in 0, that's a sensible hypothesis to make. If there were more suitable rules, we would build an active edge for each of them.

9.5.2 An Example (continued)

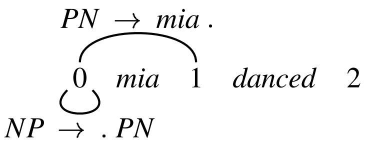
Summing up, the instantiation of step 2d for a bottom-up strategy is as follows: If the new arc A is *passive* and has category *C* as its left hand side, then look for grammar rules that have *C* as their left corner. Add active edges starting and ending in the starting point of A for all of these rules to the agenda. Their dots must be right behind the arrow.

Back to our example. We have added all new edges to the agenda and are now at the end of the first round through the repeat loop of step 2. The chart and agenda look as follows:



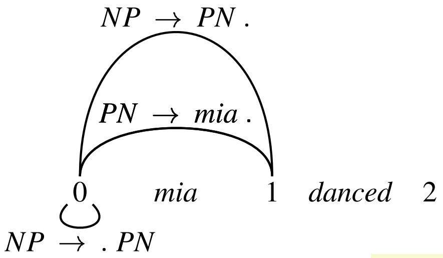
1. $\langle 0,0,NP \rightarrow .\ PN\rangle$
2. $\langle 1,2,IV \rightarrow danced\ .\rangle$

The agenda is not empty, yet, so let's start the 2nd round. The first part of this should be clear: we simply place the top of the agenda, that is, the active edge $\langle 0,0,NP \rightarrow .\ PN\rangle$, on the chart. To apply the fundamental rule (step 2c) we have to find a passive edge in the chart that starts in position 0 (the end point of the active edge). There is one; namely, $\langle 0,1,PN \rightarrow mia\rangle$. We can therefore build the edge $\langle 0,1,NP \rightarrow PN\ .\rangle$ (go back to Section 9.2 if you don't see how this works) and add it to the agenda. As step 2d only applies to passive edges, we skip it. The current state of chart and agenda are:



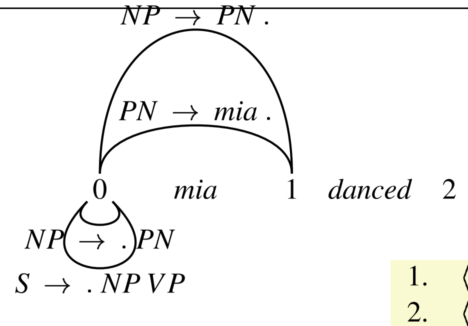
1. $\langle 0,1,NP \rightarrow PN\ .\rangle$
2. $\langle 1,2,IV \rightarrow danced\ .\rangle$

In the next round $\langle 0,1,NP \rightarrow PN\ .\rangle$ will be moved from the agenda to the chart. Step 2c doesn't produce any new edges, but in step 2d the active edges $\langle S \rightarrow NP\ VP\rangle$ and $\langle S \rightarrow NP\ VP\ PP\rangle$ will be added to the agenda. Chart and agenda then look as follows:



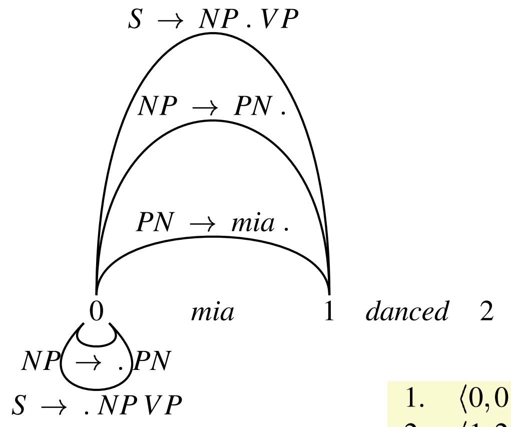
1. $\langle 0,0,S \rightarrow .\ NP\ VP\rangle$
2. $\langle 0,0,S \rightarrow .\ NP\ VP\ PP\rangle$
3. $\langle 1,2,IV \rightarrow danced\ .\rangle$

Next, we move $\langle 0,0,S \rightarrow .\ NP\ VP\rangle$ from the agenda to the chart. Applying the fundamental rule in step 2c gives us the new edge $\langle 0,1,S \rightarrow NP\ .\ VP\rangle$.



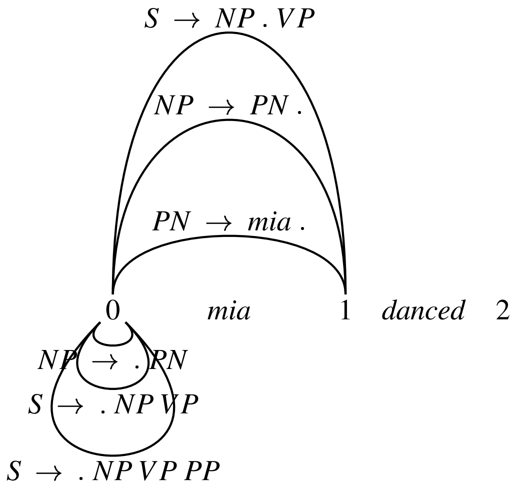
1. $\langle 0,1,S \rightarrow NP\ .\ VP\rangle$
2. $\langle 0,0,S \rightarrow\ .\ NP\ VP\ PP\rangle$
3. $\langle 1,2,IV \rightarrow danced\ .\rangle$

$\langle 0,1,S \rightarrow NP\ .\ VP\rangle$ is moved to the chart. No new edge can be built.



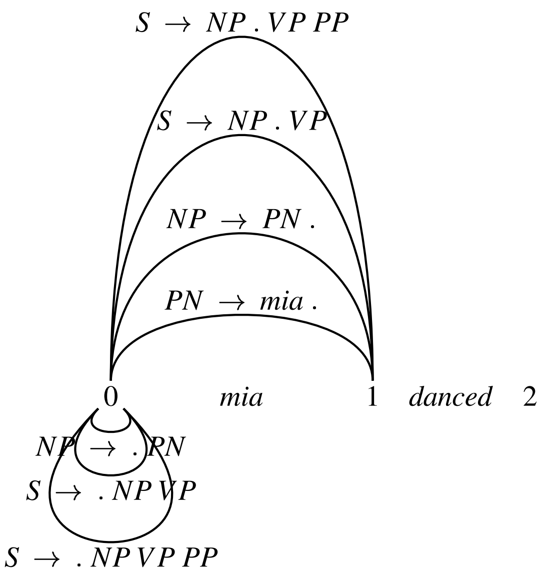
1. $\langle 0,0,S \rightarrow\ .\ NP\ VP\ PP\rangle$
2. $\langle 1,2,IV \rightarrow danced\ .\rangle$

$\langle 0,0,S \rightarrow\ .\ NP\ VP\ PP\rangle$ is moved to the chart. The fundamental rule creates $\langle 0,1,S \rightarrow NP\ .\ VP\ PP\rangle$.



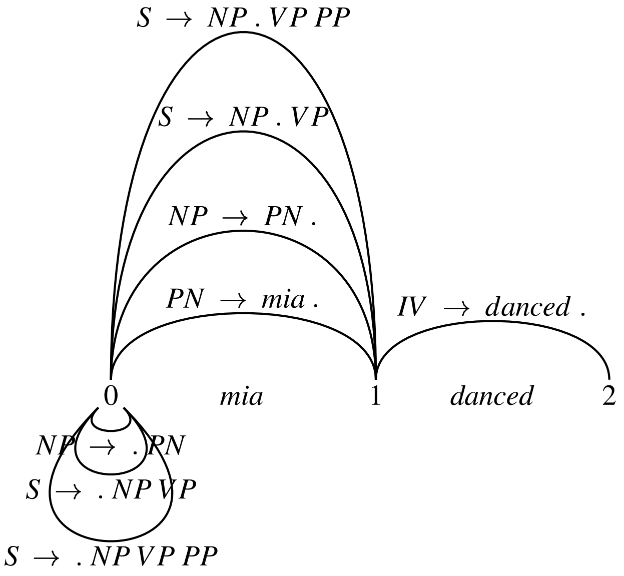
1. $\langle 0,1,S \rightarrow NP \,.\, VP\ PP\rangle$
2. $\langle 1,2,IV \rightarrow danced\ .\rangle$

$\langle 0,1,S \rightarrow NP \,.\, VP\ PP\rangle$ is moved to the chart. No new edge is created.


$S \rightarrow NP \,.\, VP\ PP$
$S \rightarrow NP \,.\, VP$
$NP \rightarrow PN\ .$
$PN \rightarrow mia\ .$
0
mia
1
danced
2
$NP \rightarrow .\ PN$
$S \rightarrow .\ NP\ VP$
$S \rightarrow .\ NP\ VP\ PP$

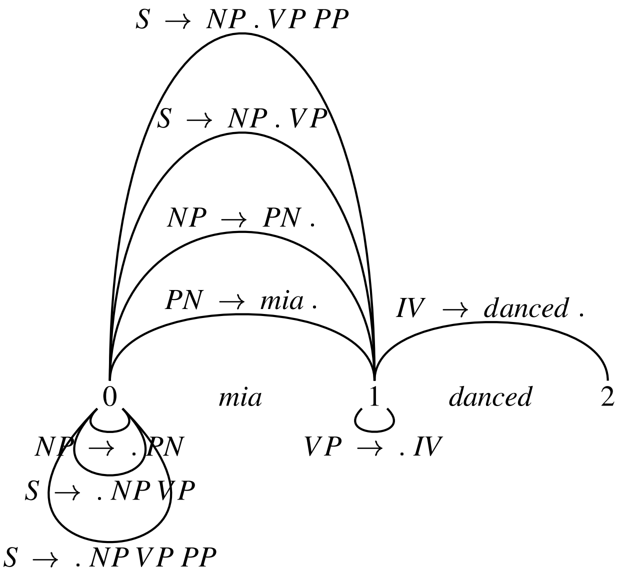
1. $\langle 1,2,IV \rightarrow danced\ .\rangle$

$\langle 1,2,IV \rightarrow danced\ .\rangle$ is moved to the chart. Step 2d predicts $VP \rightarrow .\ IV$.


$S \rightarrow NP \,.\, VP\ PP$
$S \rightarrow NP \,.\, VP$
$NP \rightarrow PN\ .$
$PN \rightarrow mia\ .$
$IV \rightarrow danced\ .$
0
mia
1
danced
2
$NP \rightarrow .\ PN$
$S \rightarrow .\ NP\ VP$
$S \rightarrow .\ NP\ VP\ PP$

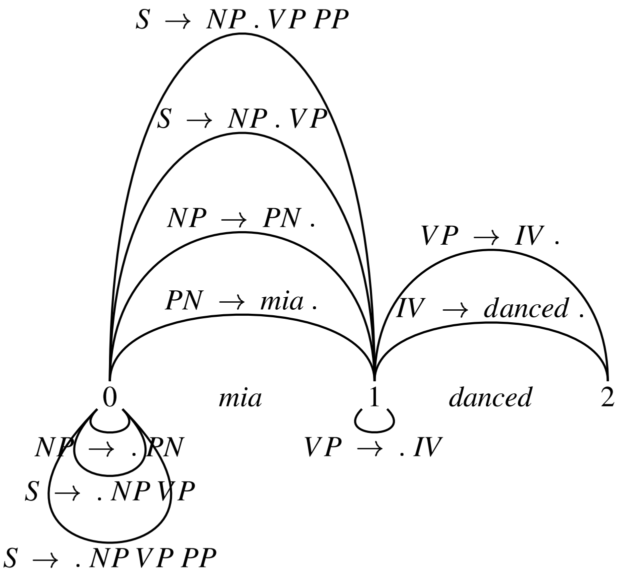
1. $\langle 1,2,VP \rightarrow .\ IV\rangle$

$\langle 1,2,VP \rightarrow .\ IV\rangle$ is moved to the chart. The fundamental rule produces $\langle 1,2,VP \rightarrow IV\ .\rangle$.



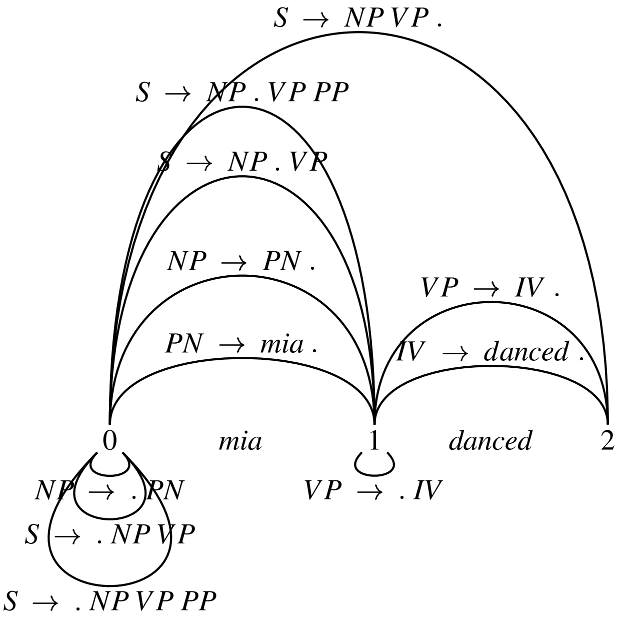
1. $\langle 1,2,VP \rightarrow IV\ .\rangle$

$\langle 1,2,VP \rightarrow IV\ .\rangle$ is moved to the chart. The fundamental rule produces $\langle 0,2,S \rightarrow NP\ VP\ .\rangle$ and $\langle 0,2,S \rightarrow NP\ VP\ .\ PP\rangle$. Step 2d, although applicable as $\langle 1,2,VP \rightarrow IV\ .\rangle$ is a passive edge, produces no new rules.



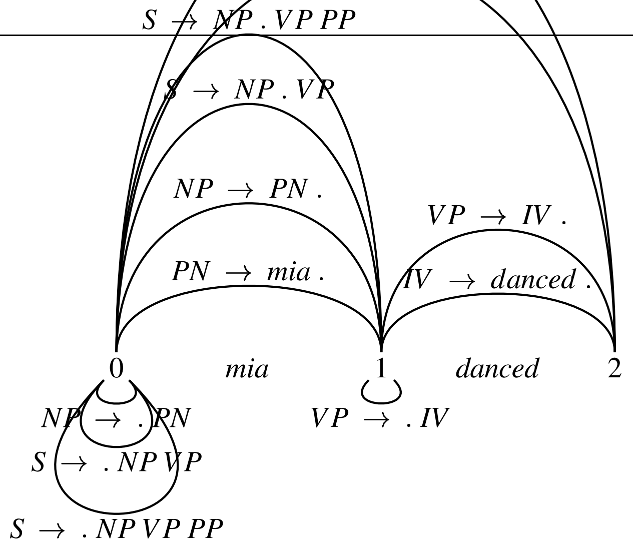
1. $\langle 0,2,S \rightarrow NP\ VP\ .\rangle$
2. $\langle 0,2,S \rightarrow NP\ VP\ .\ PP\rangle$

$\langle 0,2,S \rightarrow NP\ VP\ .\rangle$ is moved to the chart. Neither step 2c nor step 2d produce any new rules.



1. $\langle 0,2,S \rightarrow NP\ VP\ .\ PP\rangle$

$\langle 0,2,S \rightarrow NP\ VP\ .\ PP\rangle$ is moved to the chart. Again, no new edges can be built.



The agenda is now empty, so we exit the repeat. This takes us to step 3. And yes, we do have a passive s node from the first to the last node, so we have succeeded in recognizing the sentence.

9.5.3 Putting it into Prolog

It is not particularly difficult to write a Prolog program that carries out bottom-up active chart parsing. Indeed, the code is probably slightly simpler than the code for the passive chart parser that we studied in the last chapter. But before we get into it, let's briefly review the built in predicate `findall/3` as we are going to make extensive use of it.

9.5.3.1 findall/3

`findall/3` is a predicate for finding all solutions to a goal or a sequence of goals. Queries of the form

```
findall(+Object, +Goal, -List)
```

compute a list of all instantiations of `Object` that satisfy `Goal`. The query

```
?- findall(X, happy(X), L).
```

for example, would give you a list of all instantiations of `X` such that `happy(X)` is satisfied. `Object` doesn't necessarily have to be a variable. For example,

```
?- findall(happy_duck(X), (happy(X), duck(X)), L).
```

would give you a list with all instantiations of `happy_duck(X)` such that `happy(X), duck(X)` is satisfied. So, assuming that your database looks as follows

```
duck(tick).
duck(trick).
duck(track).
duck(dagobert).

happy(tick).
happy(trick).
happy(track).
```

Prolog would answer

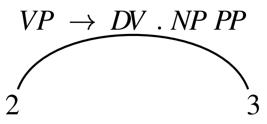
```
L = [happy_duck(tick), happy_duck(trick), happy_duck(track)]
```

9.5.3.2 Representing the Arcs

Before we look at the implementation of the parser, we have to decide how to represent arcs. We are going to use the same strategy as in the last chapter. For that purpose we use a predicate which we call `arc`. In addition to the positions that the arc spans, this predicate has to encode a dotted rule. We will therefore use `arc` with five arguments in the following way:

```
arc(Start, End, LHS, FoundSoFar, ToFind)
```

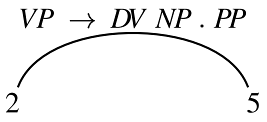
The three last arguments represent the information contained in the '.' notation of arc labels. `LHS` is the left hand side of the respective rule, `FoundSoFar` is the list of those symbols that are to the left of the dot, and `ToFind` are those that are to the right of the dot. For example, the arc



is represented as

```
arc(2, 3, vp, [dv], [np, pp])
```

One important thing to notice is that we represent the categories in `FoundSoFar` in reversed order. So



would be

```
arc(2, 5, vp, [np, dv], [pp]).
```

9.5.3.3 Bottom Up Active Chart Recognition

For representing the chart, we will again use the Prolog database. The agenda will be represented as a list. New edges are added by just appending them to the front (or the end).

The main predicate is `active_chart_recognize/1` (see See file `active_chart_bottomup.pl`.). It takes a list of words (the input sentence) as argument and succeeds if it can:

1. Initialize the chart and agenda. (step 1 of the general algorithm (page 136))
2. Build the chart while processing the agenda until it's empty. (step 2)
3. End up with a chart containing a passive `s` arc that leads from the first node to the last node. (step 3)

Here is the code:

```
active_chart_recognize(Input) :-
        cleanup,
        %%% step 1
        initialize_chart_bottomup(Input, 0),
        initialize_agenda_bottomup(Agenda),
        %%% step 2
        process_agenda(Agenda),
        %%% step 3
        length(Input, N),
        arc(0, N, s, _, []).
```

Now, let's look at the initialization predicates. We said that the initial *chart* is exactly as for the passive chart parser (see Section 8.4.1). So, `initialize_chart_bottomup/2` looks exactly like the `initialze_chart/2` predicate of the last chapter:

```
initialize_chart_bottomup([], _).


initialize_chart_bottomup([Word|Input], From) :-
        To is From + 1,
        assert(scan(From, To, Word)),
        initialize_chart_bottomup(Input, To).
```

The initial *agenda* should contain passive arcs recording the position and category of the words of the input sentence. We retrieve *one* word and its category using the following sequence of goals

```
scan(From, To, Word),
lex(Word, Cat).
```

To get *all* categories for *all* of the words, we simply use `findall`. The passive arc is constructed directly in the first argument of `findall`.

```
initialize_agenda_bottomup(Agenda) :-
        findall(arc(From, To, Cat, [Word], []),
                (
                 scan(From, To, Word),
                 lex(Word, Cat)
                ),
                Agenda
               ).
```

This is what we need in order to carry out steps 1 and 3 of the general algorithm (page 136) in a bottom-up fashion. Now, let's look at step 2, the loop which does most of the work.

9.5.3.4 Bottom Up Active Chart Recognition (continued)

`process_agenda/1` is a recursive predicate. Its argument is the list representing the agenda. It takes the first arc off that list and processes it. This may add new arguments to the agenda. In the end it recursively calls itself with the new agenda as argument. The recursion stops when the agenda is empty.

Processing of an arc works as follows. We make sure that the arc is not in the chart yet, and add it. (That's step 2b of the general algorithm.) The predicate `make_new_arcs_bottomup/2` then carries out steps 2c and 2d which may create new arcs. These are appended to the front of the agenda. If the arc is already in the chart, we throw it away and look at the rest of the agenda.

```
process_agenda([]).

process_agenda([Arc|Agenda]) :-
        \+ Arc,!,
        assert(Arc),
        %%% nl,write('New arc on chart: '), write(Arc), nl,
        make_new_arcs_bottomup(Arc, NewArcs),
        append(NewArcs, Agenda, NewAgenda),
        %%% write('New arcs on agenda: '), write(NewArcs), nl,
        process_agenda(NewAgenda).

process_agenda([_|Agenda]) :-
        process_agenda(Agenda).
```

There are two steps in the general algorithm (page 136) which generate new rules: steps 2c and 2d. In the bottom-up case, step 2c is applied to all edges, while step 2d is applied only to passive edges. The predicate `make_new_arcs_bottomup/2` therefore has two clauses which carry out only step 2c (`apply_fundamental_rule/2`) or step 2c and step 2d (`predict_new_arcs_bottomup/2`) depending on whether the arc that's coming in is active or passive.

```
make_new_arcs_bottomup(Arc, NewArcs) :-
        Arc = arc(_,_,_,_,[_|_]),
        apply_fundamental_rule(Arc, NewArcs).
```

```
make_new_arcs_bottomup(Arc, NewArcs) :-
        Arc = arc(_,_,_,_,[]),
        apply_fundamental_rule(Arc, NewArcs1),
        predict_new_arcs_bottomup(Arc, NewArcs2),
        append(NewArcs1, NewArcs2, NewArcs).
```

`apply_fundamental_rule/2` tries to apply the fundamental rule to the arc given in the first argument. There are two clauses: one for those cases where we are dealing with a passive arc and one for those cases where we are dealing with an active arc. In the first case, we have to look for an active arc which satisfies the following two conditions:

- It must end in the starting position of the passive arc.
- The next category that the active arc has to find must be what the passive arc has on its left hand side.

We again use `findall/3` to collect all possible solutions.

```
apply_fundamental_rule(arc(I, J, Cat, Done, [SubCat|SubCats]), NewArcs) :-
        findall(arc(I, K, Cat, [SubCat|Done], SubCats),
                arc(J, K, SubCat, _, []),
                NewArcs
               ).
```

In case we are dealing with an active arc, we looking for a passive arc in the chart.

```
apply_fundamental_rule(arc(J, K, Cat, _, []), NewArcs) :-
        findall(arc(I, K, SuperCat, [Cat|Done], Cats),
                arc(I, J, SuperCat, Done, [Cat|Cats]),
                NewArcs
            ).
```

When processing the chart in a bottom-up fashion we only apply step 2d to passive rules. In that case, we look for grammar rules that have the left hand side of the arc as the first symbol in the right hand side. `findall/3` again gives us all possible solutions.

```
predict_new_arcs_bottomup(arc(J, _, Cat, _, []), NewArcs) :-
        findall(arc(J, J, SuperCat, [], [Cat|Cats]),
                SuperCat ---> [Cat|Cats],
                NewArcs
            ).
```

9.6 The Code

See file `active_chart_bottomup.pl`.	The active bottom-up chart recognizer.
See file `ourEng.pl`.	Our English grammar fragment.

9.7 Top-Down Active Chart Parsing

9.7.1 Top-down rule selection

Let's first look at how to change 2d. When working bottom-up, we were interested in using rules right-to-left. We would start with structure that we knew we had (for example, perhaps we have a *passive* edge that tells us that there is a PN between positions 0 and 1) and then find a rule that we could use right-to-left to make use of this fact (for example, reading the rule $NP \rightarrow PN$ right to left we could deduce that we had an NP between positions 0 and 1). Summing up: working bottom up, we read the rules right-to-left, and start with the information in *passive* edges.

However, when we work top-down, we do precisely the opposite: we read the rules left-to-right and start with the information in *active* edges. Let's see why.

Suppose we have an active edge that starts at position 0 and is trying to build an S out of an NP and a VP in that order, and suppose it has found neither the NP nor the VP. So the arc label would be $S \rightarrow\ .\ NP\ VP$. Further, suppose we *can't* apply the fundamental rule (that is, there is no passive edge that tells us that there is an NP starting at position 0). What should we do? Since we are hypothesizing that it is possible to build an S out of an NP and a VP in that order starting at position 0, we should try and find grammar rules that will let us do this. In particular, we should look for grammar rules that have NP on the left-hand-side, for these are the rules that tell us how to build NPs — if we want that sentence, we need an NP! We then use these rules to add a new active edge at position 0. For example, if we have the rule $NP \rightarrow Det\ N$, we should add an active edge at position 0 that is trying to build an NP out of a Det and an N in that order.

In short: working top-down, we read the rules left-to-right, and start with the information in *active* edges. So in the top-down case we should change step 2d to read: 'If the edge you added was *active*, try to select rules that can be used with the edge you have just added to make new active edges. Add these new edges to the agenda.' This is the main point we need to make about rule selection, but there is another, less important issue, due to which we will make slight changes to the way we represent our grammar.

9.7.2 Initializing Chart and Agenda

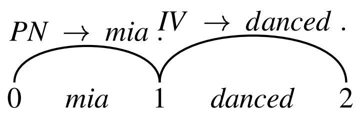
In the bottom-up algorithm, we form an initial agenda containing passive arcs that represent all possible categories of all the words in the input string. Now, this is clearly important information, and it's information we will continue to need when working top-down. But since in the top-down case new rules can only be predicted from active edges, we won't be able to use the passive arcs representing categories of words in the beginning. Only when we have worked our way down to the pre-terminal nodes will this information be important. We therefore write those arcs directly into the chart. What we do need though is at least one active edge on the agenda to start the parsing process.

So what active edge(s) should we start with? Recall that active edges can be thought of as *hypotheses* about structure — and there is one very obvious hypothesis to make about the structure of the input string, namely, that it is a sentence.

This suggests that, right at the start of the top-down algorithm, we should look at our grammar and find all the rules that let us make sentences. We should then use these rules to make active edges. These active edges should start at position 0. After all, we want to show that we have an S that starts at position 0 and spans the entire sentence.

For example, if we have the rule $S \rightarrow NP\ VP$ in our grammar, we should add at position 0 an active edge that wants to make an S out of an NP and a VP in that order, at position 0. And if we have coordination rules, for example $S \rightarrow S\ Coord\ S$, where `coord` can be ‘and’, ‘or’, ‘but’, and so on, then we should also add an active edge at position 0 that is trying to build an S out of an S followed by a coordination word followed by another S.

The initial state of chart and agenda for the sentence *Mia danced* and the grammar of the last chapter would look like this:



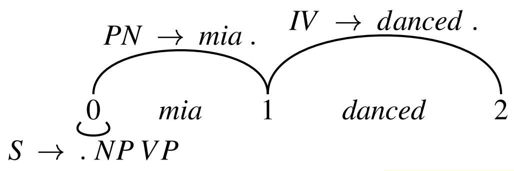
1. $\langle 0,0,S \rightarrow .\ NP\ VP\rangle$
2. $\langle 0,0,S \rightarrow .\ NP\ VP\ PP\rangle$

9.7.3 An Example

Let’s see an example of the top-down algorithm in action. As in the last chapter, we will parse the sentence *Mia danced* and we will also use the same grammar. Here it is again:

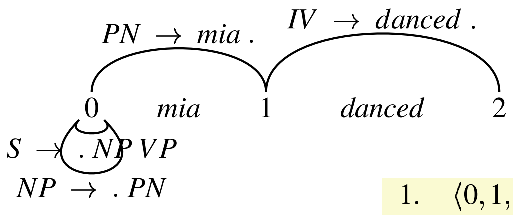
$S \rightarrow NP\ VP$
$S \rightarrow NP\ VP\ PP$
$NP \rightarrow PN$
$VP \rightarrow IV$
$PP \rightarrow P\ NP$
$PN \rightarrow mia$
$IV \rightarrow danced$

So, our initial chart and agenda will look like shown at the end of the last section. We enter the repeat loop and move the edge $\langle 0,0,S \rightarrow .\ NP\ VP\rangle$ to the chart. There is nothing in the chart that we could use to apply the fundamental rule, so we go on to step 2d. $\langle 0,0,S \rightarrow .\ NP\ VP\rangle$ is an active edge, and as we saw above, step 2d is carried out for active instead of passive edges in the top-down algorithm. We therefore have to look for a grammar rule that has *NP* at its left hand side. We find $NP \rightarrow PN$ and add it to the agenda.



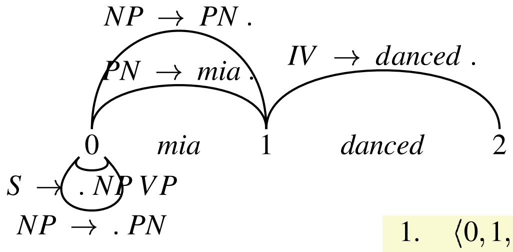
1. $\langle 0,0,NP \rightarrow .\ PN\rangle$
2. $\langle 0,0,S \rightarrow .\ NP\ VP\ PP\rangle$

$\langle 0,0,NP \rightarrow .\ PN\rangle$ is moved from the agenda to the chart. The fundamental rule creates $\langle 0,1,NP \rightarrow PN\ .\rangle$.



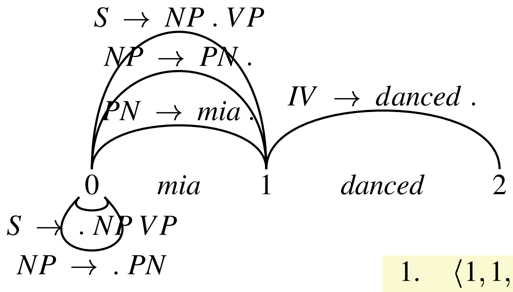
1. $\langle 0,1,NP \rightarrow PN\ .\rangle$
2. $\langle 0,0,S \rightarrow .\ NP\ VP\ PP\rangle$

$\langle 0,1,NP \rightarrow PN\ .\rangle$ is moved to the chart. Applying the fundamental rule creates $\langle 0,1,S \rightarrow PN\ .\ VP\rangle$.



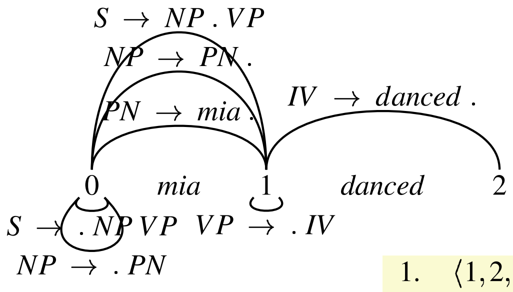
1. $\langle 0,1,S \rightarrow PN\ .\ VP\rangle$
2. $\langle 0,0,S \rightarrow .\ NP\ VP\ PP\rangle$

$\langle 0,1,S \rightarrow PN\ .\ VP\rangle$ goes to the chart. Step 2d adds a new hypothesis: $\langle 1,1,VP \rightarrow .\ IV\rangle$.



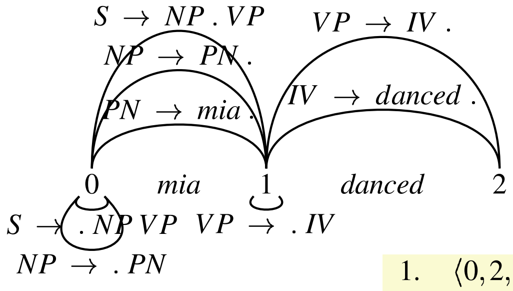
1. $\langle 1,1,VP \rightarrow .\ IV\rangle$
2. $\langle 0,0,S \rightarrow .\ NP\ VP\ PP\rangle$

$\langle 1,1,VP \rightarrow .\ IV\rangle$ is moved to the chart and combined with $IV \rightarrow danced\ .$ by the fundamental rule to form $\langle 1,2,VP \rightarrow IV\ .\rangle$.



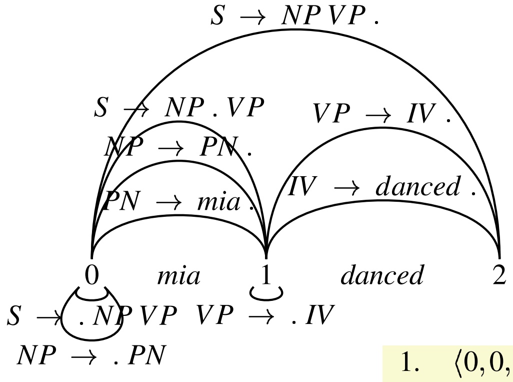
1. $\langle 1,2,VP \rightarrow IV\ .\rangle$
2. $\langle 0,0,S \rightarrow\ .\ NP\ VP\ PP\rangle$

$\langle 1,2,VP \rightarrow IV\ .\rangle$ is moved to the chart. The fundamental rule gives us $\langle 0,2,S \rightarrow NP\ VP\ .\rangle$.



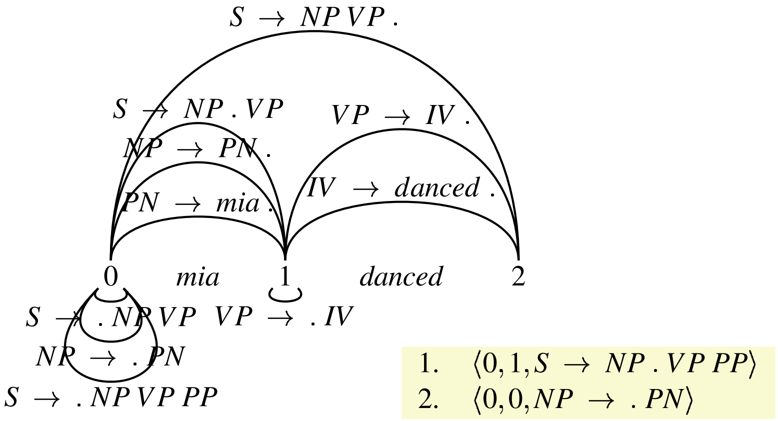
1. $\langle 0,2,S \rightarrow NP\ VP\ .\rangle$
2. $\langle 0,0,S \rightarrow\ .\ NP\ VP\ PP\rangle$

$\langle 0,2,S \rightarrow NP\ VP\ .\rangle$ is moved to the chart. Neither the fundamental rule nor step 2d produce any new edges.



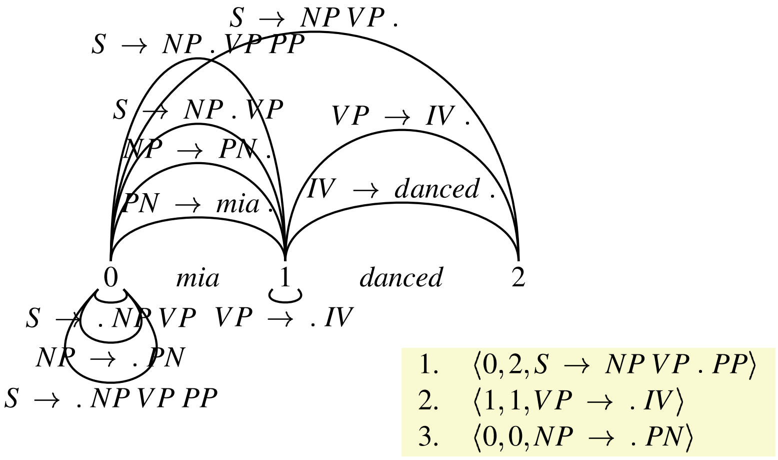
1. $\langle 0,0,S \rightarrow\ .\ NP\ VP\ PP\rangle$

$\langle 0,0,S \rightarrow .\ NP\ VP\ PP\rangle$ is moved to the chart. The fundamental rule produces $\langle 0,1,S \rightarrow NP\ .\ VP\ PP\rangle$ and step 2d predicts $\langle 0,0,NP \rightarrow .\ PN\rangle$.



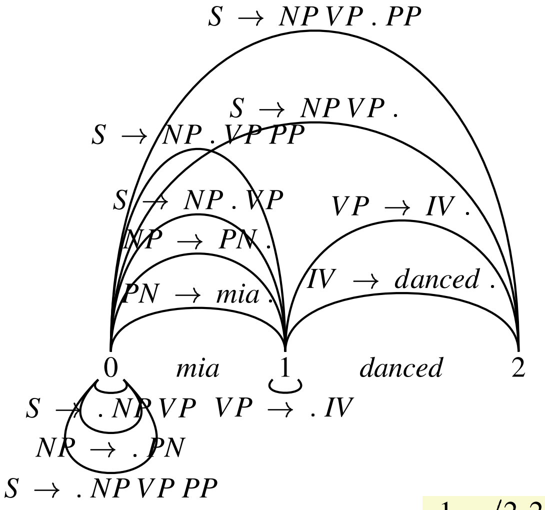
1. $\langle 0,1,S \rightarrow NP\ .\ VP\ PP\rangle$
2. $\langle 0,0,NP \rightarrow .\ PN\rangle$

$\langle 0,1,S \rightarrow NP\ .\ VP\ PP\rangle$ is moved to the chart. The fundamental rule creates $\langle 0,2,S \rightarrow NP\ VP\ .\ PP\rangle$ and step 2d predicts $\langle 1,1,VP \rightarrow .\ IV\rangle$



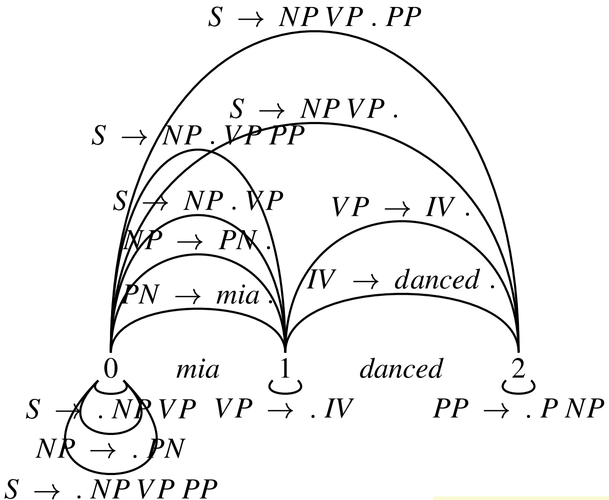
1. $\langle 0,2,S \rightarrow NP\ VP\ .\ PP\rangle$
2. $\langle 1,1,VP \rightarrow .\ IV\rangle$
3. $\langle 0,0,NP \rightarrow .\ PN\rangle$

$\langle 0,2,S \rightarrow NP\ VP\ .\ PP\rangle$ is moved to the chart. There are no appropriate edges in the chart to apply the fundamental rule. Step 2d, however, produces $\langle 2,2,PP \rightarrow .\ P\ NP\rangle$.



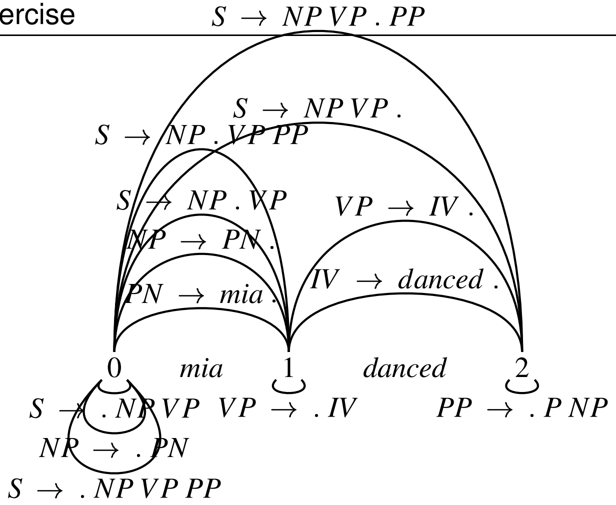
1. $\langle 2,2,PP \rightarrow\ .\ P\ NP\rangle$
2. $\langle 1,1,VP \rightarrow\ .\ IV\rangle$
3. $\langle 0,0,NP \rightarrow\ .\ PN\rangle$

$\langle 2,2,PP \rightarrow\ .\ P\ NP\rangle$ is moved to the chart, but no new edges can be created.



1. $\langle 1,1,VP \rightarrow\ .\ IV\rangle$
2. $\langle 0,0,NP \rightarrow\ .\ PN\rangle$

The two edges which are left on the agenda at this point are both already recorded in the chart. So, they will be popped off the agenda without adding any new edges to it.



You should experiment with other examples of this algorithm in action. You should also compare the top-down treatment of *Mia danced* with the bottom-up treatment that you saw in the last chapter.

9.8 Exercise

Exercise 9.1 *[This may be your end project]*

Implement a top-down chart recognizer in Prolog.

As we have said, you only have to adapt steps 1 (the initialization of chart and agenda) and 2d (how new hypotheses are predicted) of the general algorithm (page 136) to arrive at a top-down parser.

That means that you can reuse most of the code that we wrote for the bottom-up algorithm. In fact, you only have to change the predicates `initialize_chart_bottomup/2` *and* `initialize_agenda_bottomup/1` *that take care of the initialization, and the predicates* `make_new_arcs_bottomup/2` *and* `predict_new_arcs_bottomup/2` *that implement step 2d.*

initialize chart

Let's first look at the initialization. For top-down processing, we initially write not only the words and their position into the chart but also passive arcs giving us the categories of the words. So, `initialize_chart_topdown/2` *has to recurse through the input list, and*

1. *write the approriate* `scan` *entries to the chart,*
2. *retrieve all categories for each word from the lexicon, and*
3. *assert a passive arc to the chart for each of these categories.*

For step 2 and 3, you might use a failure driven loop (remember `doall/1` *from Section 8.4.2).*

initialize agenda

In the initial agenda we want to have all possible hypotheses of how a sentence can be built. Use `findall/3` *to to find all rules that have* `s` *on the left hand side of the arrow and add and active arc from position 0 to position 0.*

new arcs

Now, let's look at how new arcs are created. As in the bottom-up case, we apply step 2c to all arcs. But unlike there, we will apply step 2d only to active *arcs.* `make_new_arcs_topdown/2` *has to make sure that* `apply_fundamental_rule/2` *and* `predict_new_arcs_topdown/2` *are applied appropriately depending on whether an active or a passive arc is coming in.*

The fundamental rule remains the same. But `predict_new_arcs_topdown/2` *differs from* `predict_new_arcs_bottomup/2` *. The arcs that are coming in now are active:* $C \rightarrow CFound\ .\ CatToFind \ldots$. `predict_new_arcs_topdown/2` *has to look for all rules that have the category* `CatToFind` *on the left hand side and create new active edges for them. These new active edges don't cover anything of the string, yet, and start in the position where the active edge ends.*

That is basically it. Remember that a proper documentation is an essential part of your Endprojekt. Give a short specification of what *you implemented, i.e. the use and functionality of your program. Then describe in detail (!)* how *you implemented it. I.e. quote parts (or all) of the code and explain how it works. Third, supply examples calls that can be used for testing your program.*

Once again, if you have trouble or see that something doesn't work properly, don't mind. Document this problem and discuss your efforts to solve it. You need not be perfect, but you should show your understanding of the matter.

10

Semantic Construction

10.1 Introduction

Meaning Representations

Before looking at the details of semantics construction, there's one big question that we have to answer: Meaning as such is a very abstract concept. It's not at all easy to imagine what it is to 'get to the meaning' of a sentence, let alone to 'construct' it from that sentence. To study meaning, and especially to deal with it on a computer, we need a handle on it, something more concrete: We shall work with *meaning representation* s - strings of a formal language that has technical advantages over natural language. In this chapter, we will represent meanings using formulas of *first-order logic*.

For instance, we will say that the meaning of the sentence 'Every man walks' is represented by the first order formula $\forall x(\textsc{man}(x) \rightarrow \textsc{walk}(x))$, and that the formula $\textsc{love}(\textsc{john}, \textsc{mary})$ represents the meaning of the natural language sentence 'John loves Mary'.

So basically, this chapter will be concerned with finding a systematic way of translating natural language sentences into formulas of first order logic (and writing a program that automates this task). Here's what we will do:

1. We will start with a very short repetition of some central concepts of first order logic. For a broader overview, turn to the first chapter of the course Computational Semantics[1].
2. Then, we will show how to represent first order formulas - thus, our target representations - in Prolog.
3. Next, we will discuss theoretically some of the basic problems that arise in semantic construction, introduce λ-calculus, and show why it is our tool of choice for solving these problems.
4. Finally we will turn to our implementation: We give Prolog code for the basic functionalities of λ-calculus, and then show how to couple λ-based semantic construction with our first, DCG-based, parsing-predicates.

---

[1] `http://www.coli.uni-sb.de/~stwa/Milca/html/`

10.2 First-Order Logic: Basic Concepts

10.2.1 Vocabularies

Intuitively, a vocabulary tells us the language the 'first-order conversation' is going to be conducted in. It tells us *in what terms* we will be able to talk about things. Here is our first vocabulary:

$$(\{\textsc{mary}, \textsc{john}, \textsc{anna}, \textsc{peter}\}, \{(\textsc{love}, 2), (\textsc{therapist}, 1), (\textsc{moron}, 1)\})$$

Generally, a vocabulary is a pair of sets:

1. The first set tells us what symbols we can use to name certain entities of special interest. In the case of the vocabulary we have just established, we are informed that we will be using four symbols for this purpose (we call them *constant symbol* s or simply *name* s), namely MARY, JOHN, ANNA, and PETER.
2. The second set tells us with what symbols we can speak about certain properties and relations (we call these symbols *relation symbol* s or *predicate symbol* s). With our example vocabulary, we have one predicate symbol LOVE of arity 2 (that is, a 2-place predicate symbol) for talking about one two-place relation, and two predicate symbols of arity 1 (THERAPIST and MORON) for talking about (at most) two properties.

As such, the vocabulary we've just seen doesn't yet tell us a lot about the kinds of situations we can describe. We only know that some entitities, at most two properties, and one two-place relation will play a special role in them. But since we're interested in natural language, we will use our symbols 'suggestively'. For instance, we will only use the symbol LOVE for talking about a (one-sided) relation called loving, and the two symbols THERAPIST and MORON will serve us exclusively for talking about therapists and morons. With this additional convention, the vocabulary really shows us what kind of situations the conversation is going to be about (formally, it gives us all the information needed to define the class of models of interest - but we said that we won't go into this topic here). Syntactically, it helps us define the relevant first-order language (that means the kinds of formulas we can use). So let's next have a look at how a first order language is generated from a vocabulary.

10.2.2 First-Order Languages

A *first-order language* defines how we can use a vocabulary to form complex, sentence-like entities. Starting from a vocabulary, we then build the first-order language over that vocabulary out of the following ingredients:

The Ingredients

1. All of the symbols in the vocabulary. We call these symbols the *non-logical* symbols of the language.
2. A countably infinite collection of variables x, y, z, w and so on.

3. The Boolean connectives $\neg$ (negation), $\rightarrow$ (implication), $\vee$ (disjunction), and $\wedge$ (conjunction).
4. The quantifiers $\forall$ (the universal quantifier) and $\exists$ (the existential quantifier).
5. The round brackets) and (. (These are essentially punctuation marks; they are used to group symbols.)

Items 2-5 are called logical symbols. They are common to all first-order languages. Hence the only aspect that distinguishes first-order languages from one another is the choice of non-logical symbols (that is, of vocabulary).

10.2.3 Building Formulas

Terms

Let's suppose we've composed a certain vocabulary. How do we mix these ingredients together? That is, what is the *syntax* of first-order languages? First of all, we define a first-order *term* τ to be any constant or any variable. Roughly speaking, terms are the noun phrases of first-order languages: constants can be thought of as first-order counterparts of proper names, and variables as first-order counterparts of pronouns.

Atomic Formulas

We can then combine our 'noun phrases' with our 'predicates' (meaning, the various relation symbols in the vocabulary) to form what we call *atomic formula* s:

If R is a relation symbol of arity n , and $\tau_1,\ldots,\tau_n$ are terms, then $R(\tau_1,\ldots,\tau_n)$ is an atomic formula.

Intuitively, an atomic formula is the first-order counterpart of a natural language sentence consisting of a single clause (that is, a simple sentence). So what does a formula like $R(\tau_1,\cdots,\tau_n)$ actually mean? As a rough translation, we could say that the entities that are named by the terms $\tau_1,\cdots,\tau_n$ stand in a relationship that is named by the symbol R. An example will clarify this point:

$$\textsc{love}(\textsc{peter},\textsc{anna})$$

What's meant by this formula is that the entity named PETER stands in the relation denoted by LOVE to the entity named ANNA - or more simply, that Peter loves Anna.

Complex Formulas

Now that we know how to build atomic formulas, we can define more complex ones as well. The following inductive definition tells us exactly what kinds of *well-formed formula* s (or *wffs*, or simply *formulas*) we can form.

1. All atomic formulas are wffs.
2. If ϕ and ψ are wffs then so are $\neg\phi$, $(\phi \rightarrow \psi)$, $(\phi \vee \psi)$, and $(\phi \wedge \psi)$.
3. If ϕ is a wff, and x is a variable, then both $\exists x\phi$ and $\forall x\phi$ are wffs. (We call ϕ the *matrix* or *scope* of such wffs.)

4. Nothing else is a wff.

Roughly speaking, formulas built using $\wedge$, $\rightarrow$, $\vee$ and $\neg$ correspond to the natural language expressions '... and ...', 'if ... then ...', '... or ...', and 'it is not the case that ...', respectively. First-order formulas of the form $\exists x\phi$ and $\forall x\phi$ correspond to natural language expressions of the form 'some...' or 'all...'.

10.2.4 Bound and Free Variables

Free and Bound Variables

Let's now turn to a rather important topic: the distinction between *free variable* s and *bound variable* s.

Have a lok at the following formula:

$$\neg(\textsc{therapist}(x) \vee \forall x(\textsc{moron}(x) \wedge \forall y\textsc{person}(y)))$$

The first occurrence of x is *free*, whereas the second and third occurrences of x are *bound*, namely by the first occurrence of the quantifier $\forall$. The first and second occurrences of the variable y are also bound, namely by the second occurrence of the quantifier $\forall$.

Informally, the concept of a *bound variable* can be explained as follows: Recall that quantifications are generally of the form:

$$\forall x\phi$$

or

$$\exists x\phi$$

where x may be any variable. Generally, all occurences of this variable within the quantification are bound. But we have to distinguish two cases. Look at the following formula to see why:

$$\exists x(\textsc{man}(x) \wedge (\forall x\textsc{walks}(x)) \wedge \textsc{happy}(x))$$

1. x may occur within another, embedded, quantification $\forall x\psi$ or $\exists x\psi$, such as the x in $\textsc{walks}(x)$ in our example. Then we say that it is bound by the quantifier of this embedded quantification (and so on, if there's another embedded quantification over x within ψ).
2. Otherwise, we say that it is bound by the top-level quantifier (like all other occurences of x in our example).

Here's a full formal simultaneous definition of *free* and *bound*:

1. Any occurrence of any variable is free in any atomic formula.
2. No occurrence of any variable is bound in any atomic formula.

3. If an occurrence of any variable is free in ϕ or in ψ, then that same occurrence is free in $\neg\phi$, $(\phi \rightarrow \psi)$, $(\phi \vee \psi)$, and $(\phi \wedge \psi)$.
4. If an occurrence of any variable is bound in ϕ or in ψ, then that same occurrence is bound in $\neg\phi$, $(\phi \rightarrow \psi)$, $(\phi \vee \psi)$, $(\phi \wedge \psi)$. Moreover, that same occurrence is bound in $\forall y\phi$ and $\exists y\phi$ as well, for any choice of variable y.
5. In any formula of the form $\forall y\phi$ or $\exists y\phi$ (where y can be any variable at all in this case) the occurrence of y that immediately follows the initial quantifier symbol is bound.
6. If an occurrence of a variable x is free in ϕ, then that same occurrence is free in $\forall y\phi$ and $\exists y\phi$, for any variable y distinct from x. On the other hand, all occurrences of x that are free in ϕ, are bound in $\forall x\phi$ and in $\exists x\phi$.

If a formula contains no occurrences of free variables we call it a *sentence* .

10.2.5 Notation

In what follows, we won't always be adhering to the official first-order syntax defined above. Instead, we'll generally try and use as few brackets as possible, as this tends to improve readability. For example, we would rather not write

Outer Brackets

$$(\textsc{therapist}(\textsc{john}) \wedge \textsc{moron}(\textsc{peter}))$$

which is the official syntax. Instead, we are (almost invariably) going to drop the outermost brackets and write

$$\textsc{therapist}(\textsc{john}) \wedge \textsc{moron}(\textsc{peter})$$

Precedence

To help further reduce the bracket count, we assume the following precedence conventions for the Boolean connectives: $\neg$ takes precedence over $\vee$ and $\wedge$, both of which take precedence over $\rightarrow$. What this means, for example, is that the formula

$$\forall x((\neg\textsc{therapist}(x)) \wedge \textsc{moron}(x) \rightarrow \textsc{moron}(x))$$

is shorthand for the following:

$$\forall x((\neg\textsc{therapist}(x) \wedge \textsc{moron}(x)) \rightarrow \textsc{moron}(x))$$

In addition, we'll use the square brackets] and [as well as the official round brackets, as this can make the intended grouping of symbols easier to grasp visually.

10.2.6 Representing formulas in Prolog

We would like to use first order logic as a semantic representation formalism, and we want to deal with it in Prolog. So the next thing we need is a way of writing down formulas of first-order logic in Prolog. In short, we will simply use Prolog terms for this purpose that resemble the formulas they stand for as closely as possible. This is what we deal with in this section.

Atomic Formulas

First, we must decide how to represent *constant symbols*, *predicate symbols*, and *variables*. We do so in the easiest way possible: a first-order constant c will be represented by the Prolog atom `c`, and a first-order predicate symbol P will be represented by the Prolog atom `p`. Variables will also be represented by Prolog atoms. Note that this choice of representation won't allow our programs to distinguish constants from variables. So it's our own responsibility to choose the atoms for constants distinct from those for variables when we write down formulas in Prolog.

Given these conventions, it is obvious how *atomic formulas* should be represented. For example, LOVE(JOHN,MARY) would be represented by the Prolog term `love(john,mary)`, and HATE(PETER,x) would be represented by `hate(peter,x)`.

Complex Formulas

Next for Boolean combinations of simple formulas. The symbols

```
&        v        >        ~
```

will be used to represent the connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$ respectively.

The following Prolog code ensures that these connectives have their usual precedences:

```
:- op(900,yfx,>).          % implication

:- op(850,yfx,v).          % disjunction

:- op(800,yfx,&).          % conjunction

:- op(750, fy,~).          % negation
```

Have a look at *Learn Prolog Now!*[2] if you are unsure about what this code means.

Here are some examples of complex first-order formulas represented in Prolog. To test your understanding of the above operator definitions: How would the formulas look in fully bracketed version?

- `love(john, mary) & love(mary, john) > hate(peter,john)`
- `love(john, mary) & ~ love(mary, john) > hate(john.peter)`
- `~ love(mary, john) v love(peter,mary) & love(john, mary) > hate(john.peter)`

---

[2] http://www.coli.uni-sb.de/~kris/prolog-course/html/node82.html#sec.l9.operators

Quantifiers

Finally, we must decide how to represent quantifiers. Take for example the first order formula $\textsc{man}(x)$. Its representation as a Prolog term is `man(x)`. Now $\forall x.\textsc{man}(x)$ will be represented as

```
forall(x,man(x))
```

and $\exists x.\textsc{man}(x)$ will be represented as

```
exists(x,man(x))
```

10.3 Building Meaning Representations

10.3.1 Being Systematic

Is there a *systematic* way of translating such simple sentences as ‘John loves Mary’ and ‘A woman walks’ into first-order logic?

The key to answering this question is to be more precise about what we mean by ‘systematic’. When examining the sentence ‘John loves Mary’, we see that its semantic content is (at least partially) captured by the first-order formula LOVE(JOHN,MARY). Now this formula consists of the symbols LOVE, JOHN and MARY. Thus, the most basic observation we can make about systematicity is the following: the proper name ‘John’ contributes the constant symbol JOHN to the representation, the transitive verb ‘loves’ contributes the relation symbol LOVE, and the proper name ‘Mary’ contributes the constant symbol MARY.

More generally, it’s the words of which a sentence consists that contribute the relation symbols and constants in its semantic representation. But (important as it may be) this observation doesn’t tell us everything we need to know about systematicity. It tells us where the building blocks of our meaning representations will come from - namely from words in the lexicon.

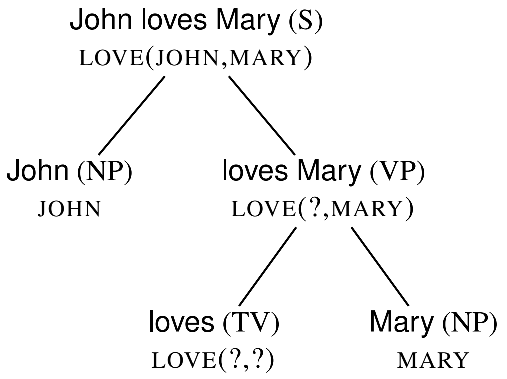
But it doesn’t tell us how to *combine* these building blocks. For example we have to form the first-order formula LOVE(JOHN,MARY) from the symbols LOVE, JOHN and MARY. But from the same symbols we can also form LOVE(MARY,JOHN). So why do we choose to put MARY in the second argument slot of LOVE rather than in the first one? Is there a principle behind this decision? For this task, we haven’t been specific yet about what we mean by working *in a systematic fashion*.

10.3.2 Being Systematic (II)

Syntactic Structure...

Our missing link here is the notion of *syntactic structure* . As we know well from the previous chapters, ‘John loves Mary’ isn’t just a string of words: it has a hierarchical structure. In particular, ‘John loves Mary’ is an S (sentence) that is composed of the subject NP (noun phrase) ‘John’ and the VP (verb phrase) ‘loves Mary’. This VP is in turn composed of the TV (transitive verb) ‘loves’ and the direct object NP ‘Mary’. Given this hierarchy, it is easy to tell a conclusive story about - and indeed, to draw a convincing picture of - why we should get the representation LOVE(JOHN,MARY) as a result, and nothing else:

See movie in HTML version.



...and its use for Semantics

When we combine a TV with an NP to form a VP, we have to put the semantic representation associated with the NP (in this case, MARY) in the *second* argument slot of the VP's semantic representation (in this case, LOVE(?,?)). And why does JOHN need to be inserted into the *first* argument slot? Simply because this is the slot reserved for the semantic representations of NPs that we combine with VPs to form an S.

In more general terms, given that we have some reasonable syntactic story about what the pieces of our sentences are, and which pieces combine with which other pieces, we can try to use this information to explain how the various semantic contributions have to be combined.

Summing up we are now in a position to give quite a good explication of 'systematicity': When we construct meaning representations systematically, we integrate information from *two different* sources:

1. The lexical items (i.e. the words) in a sentence give us the basic ingredients for our representation.
2. Syntactic structure tells us how the semantic contributions of the parts of a sentence are to be joined together.

10.3.3 Three Tasks

Let us have a look at the general picture that's emerging. How do we translate simple sentences such as 'John loves Mary' and 'A woman walks' into first-order logic? Although we still don't have a specific method at hand, we can formulate a plausible strategy for finding one. We need to fulfill three tasks:

Task 1	Specify a reasonable syntax for the natural language fragment of interest.
Task 2	Specify semantic representations for the lexical items.
Task 3	Specify the translation of complex expressions (i.e. phrases and sentences) *compositionally*. That is, we need to specify the translation of such expressions in terms of the translation of their parts, *parts* here referring to the substructure given to us by the syntax.

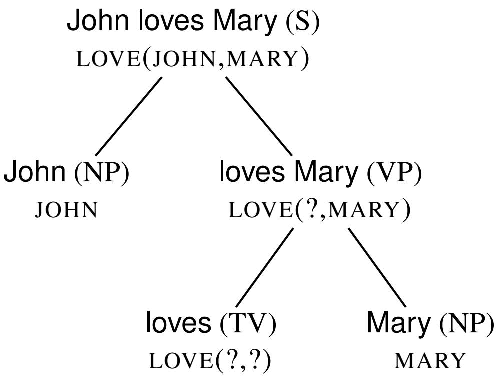
Of course all three tasks should be carried out in a way that naturally leads to computational implementation. Because this chapter is on *semantic* construction, tasks 2 and 3 are where our real interests lie, and most of our attention will be devoted to them. But we also need a way of handling task 1.

10.3.4 From Syntax to Semantics

Task 1 ✓

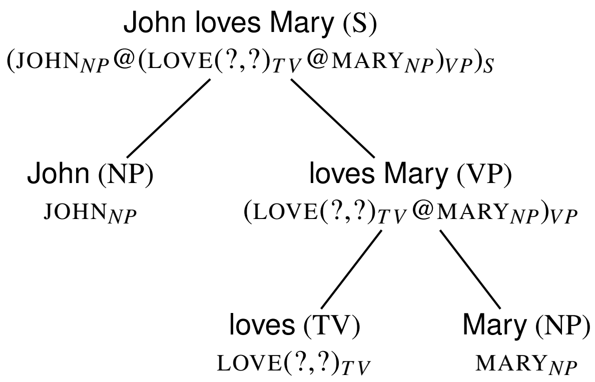
In order to approach Task 1, we will use a simple context free grammar like the ones we've seen in previous lectures. As ususal, the syntactic analysis of a sentence will be represented as a tree whose non-leaf nodes represent *complex syntactic categories* (such as S, NP and VP) and whose leaves represent *lexical items* (these are associated with *lexical categories* such as noun, transitive verb, determiner, proper name and intransitive verb). To enhance the readability of such trees, we will ommit the non-branching steps and take for instance Mary (NP) as a leave node.

Let's have a second look at our semantically annotated syntax-tree for the sentence 'John loves Mary' (from Section 10.3.2).



We said that the systematic contribution of syntactic structure to semantic construction consists in guiding the semantic contributions of words and phrases to the right places in the final semantic representation. Obviously, when we contructed the formula LOVE(JOHN,MARY) along the above syntax tree, we made tacit use of a lot of knowledge about how exactly syntactic information should be used. Can we make this knowledge more explicit?

Let's take a step back. What's the simplest way of taking over syntactic information into our semantic representation? Surely, the following is a very undemanding first step:



We've simply taken the semantic contributions of words and phrases, uniformly joined them with an @-symbol, and encoded the tree structure in the bracketing structure of the result. Yet the result is still quite far from what we actually want to have. It definitely isn't a first order formula. In fact we've only postponed the question of how to exploit the syntactic structure for guiding arguments to their places. What we've got is a nice linear 'proto'-semantic representation, in which we still have all syntactic information at hand. But this representation still needs a lot of post-processing.

What we could now try to do is start giving post-procesing rules for our 'proto'-semantic representation, rules like the following:'If you find a transitive verb representation between two @-symbols, always take the item to its left as first argument, and the item to its right as second argument.'

Formulating such rules would soon become very complicated, and surely our use of terms like 'item to the left' indicates that we've not yet reached the right level of abstraction in our formulation. In the next section, we're going to look at λ-calculus, a formalism that gives us full flexibility in speaking about missing pieces in formulas, where they're missing, and when and from where they should be supplied. It provides the right level of generality for capturing the systematics behind the influence that syntactic structure has on meaning construction. Post-processing rules like the one just seen won't be necessary, their role is taken over by the uniform and very simple operation of β-reduction.

10.4 The Lambda Calculus

10.4.1 Lambda-Abstraction

λ-expressions are formed out of ordinary first order formulas using the λ-operator. We can prefix the λ-operator, followed by a variable, to any first order formula or λ-expression. We call expressions with such prefixes *λ-abstraction* s (or, more simply, *abstractions*). We say that the variable following a λ-operator is *abstracted over* . And we say that the variable abstracted over is *(λ-)bound* by its respective λ-operator within an abstraction, just as we say that a quantified variable is bound by its quantifier inside a quantification.

Abstractions

The following two are examples of λ-abstractions:

1. $\lambda x.\textsc{woman}(x)$
2. $\lambda u.\lambda v.\textsc{love}(u,v)$

In the first example, we have abstracted over x. Thus the x in the argument slot of WOMAN is bound by the λ in the prefix. In the second example, we have abstracted twice: Once over v and once over u. So the u in the first argument slot of LOVE is bound by the first λ, and the v is bound by the second one.

Missing Information

We will think of occurrences of variables bound by λ as placeholders for missing information: They serve us to mark *explicitly* where we should substitute the various bits and pieces obtained in the course of semantic construction. Let us look at our first example λ-expression again. Here the prefix $\lambda x.$ states that there is information missing in the formula following it (a one-place predication), and it gives this 'information gap' the name x. The same way in our second example, the two prefixes $\lambda u.$ and $\lambda v.$ give us separate handles on *each of the two* information gaps in the following two-place predication.

10.4.2 Reducing Complex Expressions

So the use of λ-bound variables allows us to mark places where information is missing in a partial first order formula. But how do we fill in the missing information when it becomes available? The answer is simple: We *substitute* it for the λ-bound variable. We can read a λ-prefix as a request to perform substitution for its bound variable.

Controlled substitution

In $\lambda x.\textsc{woman}(x)$, the binding of the free variable x in WOMAN(x) explicitly indicates that WOMAN has an argument slot where we may perform substitutions.

We will use concatenation (marked by an @-symbol) to indicate when we have to perform substitutions, and what to substitute. By concatentating a λ-expression with another expression, we trigger the substitution of the latter for the λ-bound variable. Consider the following expression (we use the special symbol @ to indicate concatenation):

$$\lambda x.\textsc{woman}(x)@\textsc{mary}$$

Functional Application, β-Reduction

This compound expression consists of the abstraction $\lambda x.\textsc{woman}(x)$ written immediately to the left of the expression MARY, both joined together by @. Such a concatenation is called *functional application* ; the left-hand expression is called the *functor*, and the right-hand expression the *argument*. The concatenation is an instruction to discard the $\lambda x.$ prefix of the functor, and to replace every occurrence of x that was bound by this prefix with the argument. We call this substitution process β-*reduction* (other common names include β-conversion and λ-conversion). Performing the β-reduction demanded in the previous example yields:

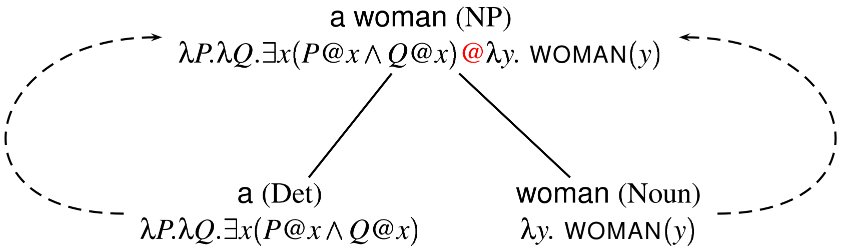
$$\text{WOMAN}(\text{MARY})$$

The purpose of λ-bound variables is thus to mark the slots where we want substitutions to be made, the purpose of λ-prefixes is to indicate at what point in the reduction process substitutions should be made, and the arguments of applications provide the material to be substituted. Abstraction, functional application, and β-reduction together will drive our first really systematic semantic construction mechanism. Next, let's see how it works in practice.

10.4.3 Using Lambdas

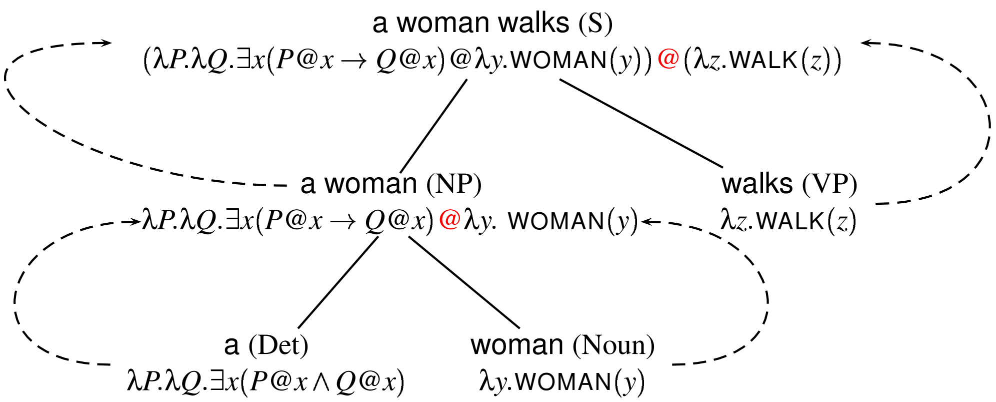
Let's return to the sentence 'A woman walks'. According to our grammar, a determiner and a common noun can combine to form a noun phrase. Our semantic analysis couldn't be simpler: we will associate the NP node with the functional application that has the determiner representation as functor and the noun representation as argument. Structurally, this is of course the same thing that we did in Section 10.3.4. Only this time the semantic contributions of constituents are generally λ-expressions, and we will simply read the @-symbols as application markers. In fact it will turn out that the combination of functional application and β-reduction is a method of such generality that we can even completely disregard the phrase-indices (such as NP and VP) that were part of our 'proto'-representations in Section 10.3.4.

Building a structured application...


a woman (NP)
$\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)@\lambda y.\ \text{WOMAN}(y)$
a (Det)
$\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)$
woman (Noun)
$\lambda y.\ \text{WOMAN}(y)$

As you can see from the picture, we use the λ-expression $\lambda P.\lambda Q.(\exists x(P@x \wedge Q@x))$ as our representation for the indefinite determiner 'a'. We'll take a closer look at this representation soon, after we've looked at how it does its job in the semantic construction process. But there's one thing that we have to remark already now. While the λ-bound variables in the examples we've seen so far were placeholders for missing *constant* symbols, P and Q in our determiner-representation stand for missing *predicates*. The version of λ-calculus introduced here does not distinguish variables that stand for different kinds of missing information. Nevertheless we will stick to a convention of using lower case letters for variables that stand for missing constant symbols, and capital letters otherwise.

But now let's carry on with the analysis of the sentence 'A woman walks'. We have to incorporate the intransitive verb 'walks'. We assign it the representation $\lambda.z\text{WALK}(z)$. The following tree shows the final representation we obtain for the complete sentence:



The S node is associated with $(\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)@\lambda y.\textsc{woman}(y))\,@\,(\lambda z.\textsc{walk}(z))$. We obtain this representation by a procedure analogous to that performed at the NP node. We associate the S node with the application that has the NP representation just obtained as functor, and the VP representation as argument.

...and reducing it.

Now instead of hand-tailoring lots of specially dedicated post-processing rules, we will simply β-reduce as often as possible the expression we find at the S node. We must follow its (bracketed) structure when we perform β-reduction. So we start with reducing the application $\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)\,@\,\lambda y.\textsc{woman}(y)$. We have to replace P by $\lambda y.\textsc{woman}(y)$, and drop the λP prefix. The whole representation then looks as follows:

$$\lambda Q.\exists x(\lambda y.\textsc{woman}(y)@x \wedge Q@x)\,@\,\lambda z.\textsc{walk}(z)$$

See movie in HTML version.

Let's go on. This time we have two applications that can be reduced. We decide to get rid of the λQ. Replacing Q by $\lambda z.\textsc{walk}(z)$ we get:

$$\exists x(\lambda y.\textsc{woman}(y)\,@\,x \wedge \lambda z.\textsc{walk}(z)\,@\,x)$$

Again we have the choice where to go on β-reducing, but this time it should be clear that our choice doesn't make any difference for the final result (in fact it never does. This property of λ-calculus is called *confluence*). Thus let's β-reduce twice. We have to replace both y and z by x. Doing so finally gives us the desired:

$$\exists x(\textsc{woman}(x) \wedge \textsc{walks}(x))$$

Determiner

Finally, let's have a closer look at the determiner-representation we've been using. Remember it was $\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)$. Why did we choose this expression? In a way, there isn't really an answer to this question, except simply: *Because it works.*

So now let's have a closer look at why it works. We know that a determiner must contribute a quantifier *and* the pattern of the quantification. Intuitively, indefinite determiners in natural language are used to indicate that there is something of a certain kind (epressed in the so-called *restriction* of the determiner), about which one is going to say that it also has some other property (expressed in the so-called *scope* of the determiner). In the sentence 'A woman walks', the 'a' indicates that there is something of a kind, which is then specified to be 'woman', that also has a certain property, which is then specified as 'walk'.

So for the case of an indefinite determiner, we know that the quantifier in its first-order formalization has to be existential, and that the main connective within the quantification is a conjunction symbol. This is the principle behind formalizing indefinite determiners in first-order logic.

Now clever use of λ-bound variables in our determiner representation allows us to leave unspecified all but just these two aspects. All that is already 'filled in' in the representation $\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)$ is the quantifier and a little bit about the internal structure of its scope, namely the main connective $\wedge$. The rest is 'left blank', and this is indicated using variables.

The second crucial aspect of a λ-expression is the order of prefixes. This is where the role of syntactic structure comes in: It should be obvious from the further construction process why we had to choose $\lambda P.\lambda Q$ and not $\lambda Q.\lambda P$ - the reason is simply that phrases and sentences containing determiners are generally built up syntactically as they are. So when deciding about the order of λ-prefixes of a meaning representation, one has to think of the right generalizations over the syntactic use of its natural language counterpart.

10.4.4 Advanced Topics: Proper Names and Transitive Verbs

It looks as if there are clouds on the horizon. We said before that the first-order counterparts of proper names are constant symbols, and that for example JOHN stands for 'John'. But while the semantic representation of a quantifying NP such as 'a woman' can be used as a functor, surely such a constant symbol will have to be used as an argument. Will this be a problem for our semantic construction mechanism?

Proper names

In fact, there's no problem at all - if we only look at things the right way. We *want* to use proper names as functors, the same way as quantified noun phrases. So maybe we just shouldn't translate them as constant symbols *directly*. Let's simply keep the intended use in mind when designing semantic representations for proper names. It's all a matter of abstracting cleverly. Indeed the λ-calculus offers a delightfully simple functorial representation for proper names, as the following examples show:

'Mary': $\lambda P.P@\textsc{mary}$

'John': $\lambda Q.Q@\textsc{john}$

Role-Reversing

From outside (i.e. if we only look at the λ-prefix) these representations are exactly like the ones for quantified noun phrases. They are abstractions, thus they can be used as functors in the same way. However looking at the inside, note what such functors do. They are essentially instructions to substitute their argument in P (or Q), which amounts to applying their own arguments to themselves! Because the λ-calculus offers us the means to specify such role-reversing functors, proper names can be used as functors just like quantified NPs.

Transitive verbs

As an example of these new representations in action, let us build a representation for ‘John loves Mary’. But before we can do so, we have to meet another challenge: ‘loves’ is a transitive verb, it takes an object and forms a VP; we will want to apply it to its object-NP. And the resulting VP should be usable just like a standard intransitive verb; we want to be able to apply the subject NP to it. This is what we know in advance.

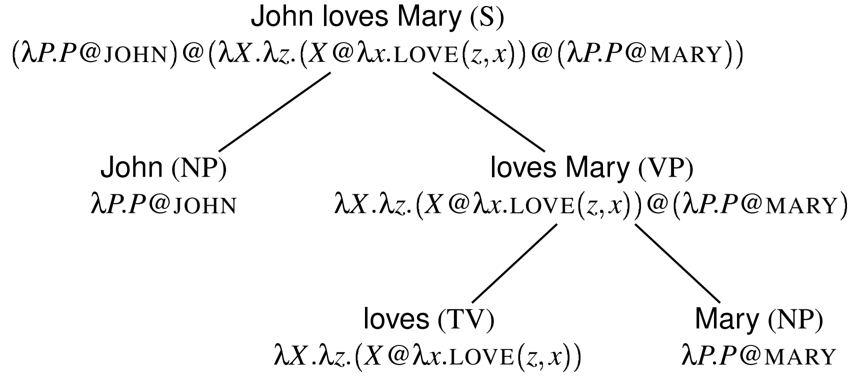
Given these requirements, a λ-expression like the simple $\lambda u.\lambda v.\textsc{love}(u,v)$ (which we’ve seen in Section 10.4.1) surely won’t do. After all, the object NP combining with a transitive verb is itself a functor. It would be inserted for u in this λ-expression, but u isn’t applied to anything anywhere. So the result could never be β-reduced to a well-formed first-order formula. How do we make our representation fit our needs this time? Let’s try something like our role-reversing trick again; we’ll assign ‘loves’ the following λ-expression:

$\lambda R.\lambda z.(R@\lambda x.\textsc{love}(z,x))$

An example

Thus prepared we’re now ready to have a look at the semantic construction for ‘John loves Mary’. We can build the following tree:

See movie in HTML version.


John loves Mary (S)
$(\lambda P.P@\textsc{john})@(\lambda X.\lambda z.(X@\lambda x.\textsc{love}(z,x))@(\lambda P.P@\textsc{mary}))$
John (NP)
$\lambda P.P@\textsc{john}$
loves Mary (VP)
$\lambda X.\lambda z.(X@\lambda x.\textsc{love}(z,x))@(\lambda P.P@\textsc{mary})$
loves (TV)
$\lambda X.\lambda z.(X@\lambda x.\textsc{love}(z,x))$
Mary (NP)
$\lambda P.P@\textsc{mary}$

How is this going to work? Let’s look at the application at the S-node, and think through step by step what happens when we β-convert (page 167) it: Inside our complex application, the representation for the object NP is substituted for X. It ends up being applied to something looking like an intransitive verb (namely to $\lambda x.\textsc{love}(z,x)$). This application is going to be no problem - it’s structurally the same we would get if our object NP was the subject of an intransitive verb. So everything is fine here.

Now the remaining prefix λz makes the complete VP-representation also function like that of an intransitive verb (from outside). And indeed the subject NP semantic representation finally takes the VP semantic representation as argument, as if it was the representation of an intransitive verb. So everything is fine here, too.

Trace the semantic construction!

Make sure you understand what is going on here by β-reducing the expression at the S-node yourself!

10.4.5 The Moral

Our examples have shown that λ-calculus is ideal for semantic construction in two respects:

1. The process of combining two representations was perfectly uniform. We simply said which of the representations is the functor and which the argument, whereupon combination could be carried out by applying functor to argument and β-converting. We didn't have to make any complicated considerations here.
2. The load of semantic analysis was carried by the lexicon: We used the λ-calculus to make missing information stipulations when we gave the meanings of the *words* in our sentences. For this task, we had to think accurately. But we could make our stipulations declaratively, without hacking them into the combination process.

Our observations are indeed perfectly general. Doing semantic construction with the help of λ-calculus, most of the work is done before the actual combination process.

What we have to do...

When giving a λ-abstraction for a lexical item, we have to make two kinds of decisions:

1. We have to locate gaps to be abstracted over in the partial formula for our lexical item. In other words, we have to decide *where to put* the λ-bound variables inside our abstraction. For example when giving the representation $\lambda P.P@\textsc{mary}$ for the proper name 'Mary' we decided to stipulate a missing functor. Thus we applied a λ-abstracted variable to MARY.
2. We have to decide *how to arrange* the λ-prefixes. This is how we control in which order the arguments have to be supplied so that they end up in the right places after β-reduction when our abstraction is applied. For example we chose the order $\lambda P.\lambda Q$ when we gave the representation $\lambda P.\lambda Q.\exists x(P@x \wedge Q@x)$ for the indefinite determiner 'a'. This means that we will first have to supply it with the argument for the restriction of the determiner, and then with the one for the scope.

...and how

Of course we are not totally free how to make these decisions. What constrains us is that we want to be able to combine the representations for the words in a sentence so that they can be *fully* β-*reduced* to a well-formed first order formula. And not just some formula, but the one that captures the meaning of the sentence.

So when we design a λ-abstraction for a lexical item, we have to anticipate its potential use in semantic construction. We have to keep in mind *which final semantic representations* we want to build for sentences containing our lexical item, and *how* we want to build them. In order to decide what to abstract over, we must think about *which pieces* of semantic material will possibly be supplied from elsewhere during semantic construction. And in order to arrange our λ-prefixes, we must think about *when and from where* they will be supplied.

Summing up

The bottom line of all this is that devising lexical representations will be the tricky part when we give the semantics for a fragment of natural language using λ-calculus. But with some clever thinking, we can solve a lot of seemingly profound problems in a very streamlined manner.

10.4.6 What's next

What's next?

For the remainder of this lecture, the following version of the three tasks listed earlier (page 164) will be put into practise:

Task 1 Specify a DCG for the fragment of natural language of interest.

Task 2 Specify semantic representations for the lexical items with the help of the λ-calculus.

Task 3 Specify the translation $\mathcal{R}'$ of a syntactic item $\mathcal{R}$ whose parts are $\mathcal{F}$ and $\mathcal{A}$ with the help of functional application. That is, specify which of the subparts is to be thought of as functor (here it's $\mathcal{F}$), which as argument (here it's $\mathcal{A}$) and then define $\mathcal{R}'$ to be $\mathcal{F}'@\mathcal{A}'$, where $\mathcal{F}'$ is the translation of $\mathcal{F}$ and $\mathcal{A}'$ is the translation of $\mathcal{A}$. Finally, apply β-conversion as a post-processing step.

10.4.7 [Sidetrack:] Accidental Bindings

But before we can put λ-calculus to use in an implementation, we still have to deal with one rather technical point: Sometimes we have to pay a little bit of attention which variable names we use. Suppose that the expression $\mathcal{F}$ in $\lambda V.\mathcal{F}$ is a complex expression containing many λ operators. Now, it could happen that when we apply a functor $\lambda V.\mathcal{F}$ to an argument $\mathcal{A}$, some occurrence of a variable that is is free in $\mathcal{A}$ becomes bound when we substitute it into $\mathcal{F}$.

For example when we construct the semantic representation for the verb phrase 'loves a woman', syntactic analysis of the phrase could lead to the representation:

$$\lambda P.\lambda y.(P@\lambda x.\textsc{love}(y,x))@(\lambda Q.\lambda R.(\exists y(Q@(y)\wedge R@y))@\lambda w.\textsc{woman}(w))$$

β-reducing three times yields:

$$\lambda y.(\lambda R.(\exists y(\textsc{woman}(y)\wedge R@y))@\lambda x.\textsc{love}(y,x))$$

Notice that the variable y occurs λ-bound as well as existentially bound in this expression. In $\textsc{love}(y,x)$ it is bound by λy, while in $\textsc{woman}(y)$ and R it is bound by $\exists y$.

So far, this has not been a problem. But look what happens when we β-convert once more:

$$\lambda y.(\exists y(\text{WOMAN}(y) \wedge \lambda x.\text{LOVE}(y,x)@y))$$

$\text{LOVE}(y,x)$ has been moved inside the scope of $\exists y$. In result, the occurrence of y has been 'caught' by the existential quantifier, and λy doesn't bind any occurence of a variable at all any more. Now we β-convert one last time and get:

$$\lambda y.(\exists y(\text{WOMAN}(y) \wedge \text{LOVE}(y,y)))$$

We've got an empty λ-abstraction, made out of a formula that means something like 'A woman loves herself'. *That's not what we want to have*. Such accidental bindings (as they are usually called) defeat the purpose of working with the λ-calculus. The whole point of developing the λ- calculus was to gain control over the process of performing substitutions. We don't want to lose control by foolishly allowing unintended interactions.

10.4.8 [Sidetrack:] Alpha-Conversion

But such interactions need never happen. Obviously, the fact that lies at the heart of our problem is that we used *two* variables named y in our representation. But λ-bound variables are merely placeholders for substitution slots. The exact names of these placeholders do not play a role for their function. So, relabeling bound variables yields λ-expressions which lead to exactly the same substitutions in terms of 'slots in the formulas' (much like relabeling bound variables in quantified formulas doesn't change their truth values).

Let us look at an example. The λ-expressions $\lambda x.\text{MAN}(x)$, $\lambda y.\text{MAN}(y)$, and $\lambda z.\text{MAN}(z)$ are equivalent, as are the expressions $\lambda Q.\exists x(\text{WOMAN}(x) \wedge Q@x)$ and $\lambda Y.\exists x(\text{WOMAN}(x) \wedge Y@x)$. All these expressions are functors which when applied to an argument, replace the bound variable by the argument. No matter which argument $\mathcal{A}$ we choose, the result of applying any of the first three expressions to $\mathcal{A}$ and then β-converting is $\text{MAN}(\mathcal{A})$, and the result of applying either of the last two expressions to $\mathcal{A}$ is $\exists x(\text{WOMAN}(x) \wedge \mathcal{A}@x)$.

α-Equivalence

Two λ-expressions are called α-*equivalent* if they only differ in the names of λ-bound variables. In what follows we often treat α-equivalent expressions as if they were identical. For example, we will sometimes say that the lexical entry for some word is a λ-expression $\mathcal{E}$, but when we actually work out some semantic construction, we might use an α-equivalent expression $\mathcal{E}'$ instead of $\mathcal{E}$ itself.

α-Conversion

The process of relabeling bound variables is called α-*conversion* . Since the result of α-converting an expression performs the same task as the initial expression, α-conversion is always permissible during semantic construction. But the reader needs to understand that it's not merely *permissible* to α-convert, it can be *vital* to do so if β-conversion is to work as intended.

Returning to our intial problem, if we can't use $\lambda V.\mathcal{F}$ as a functor, any α-equivalent formula will do instead. By suitably relabeling the bound variables in $\lambda V.\mathcal{F}$ we can always obtain an α-equivalent functor that doesn't bind any of the variables that occur free in $\mathcal{A}$, and accidental binding is prevented.

So, strictly speaking, it is not merely functional application coupled with β-conversion that drives the process of semantic construction in this course, but functional application and β-conversion coupled with (often tacit) use of α-conversion. Notice we only didn't encounter the problem of accidental binding earlier because we (tacitly) chose the names for the variables in the lexical representations cleverly. This means that we have been working with α-equivalent variants of lexical entries all along in our examples.

10.5 Implementing Lambda Calculus

10.5.1 Representations

First, we have to decide how to represent λ-expressions in Prolog. As in the case of representing first-order formulas, we will use Prolog terms that resemble the expressions they represent as closely as possible. For abstractions, something as simple as the following will do:

```
lambda(x,F)
```

Secondly, we have to decide how to represent application. Let's simply transplant our @-notation to Prolog by defining @ as an infix operator:

```
 :- op(950,yfx,@).          % application
```

That is, we shall introduce a new Prolog operator `@` to explicitly mark where functional application is to take place: the notation `F@A` will mean 'apply function `F` to argument `A`'. We will build up our representations using these explicit markings, and then carry out β-conversion when all the required information is to hand.

10.5.2 Extending the DCG

Let's see how to use this notation in DCGs. We'll use an extended version of our well-known `dCGExample.pl` from Chapter 5. To make things a bit more interesting, we've added an intransitive verb and a proper name as well as the necessary rules to use them in sentences. To use the resulting DCG for semantic construction, we have to specify the semantic representation for each phrasal and lexical item. We do this by giving additional arguments to the phrase markers of the DCG (a technique that we already know from Chapter 5).

The resulting grammar is found in See file `semanticDCG.pl`.. Let's have a look at the phrasal rules first:

```
s(NP@VP) --> np(NP),vp(VP).
```

```
np(DET@N) --> det(DET),n(N).
np(PN) --> pn(PN).

vp(TV@NP) --> tv(TV),np(NP).
vp(IV) --> iv(IV).
```

The unary phrasal rules just percolate up their semantic representation (here coded as Prolog variables `NP`, `VP` and so on), while the binary prasal rules use `@` to build a semantic representation out of their component representations. This is completely transparent: we simply apply function to argument to get the desired result.

10.5.3 The Lexicon

The real work is done at the lexical level. Nevertheless, the lexical entries for nouns and intransitive verbs practically write themselves:

```
n(lambda(X, witch(X))) --> [witch], {vars2atoms(X)}.
n(lambda(X, wizard(X))) --> [wizard], {vars2atoms(X)}.
n(lambda(X, broomstick(X))) --> [broomstick], {vars2atoms(X)}.

iv(lambda(X, fly(X))) --> [flies], {vars2atoms(X)}.
```

If you do not remember the somewhat difficult representation of transitive verbs, look at Section 10.4.4 again. Here's the lexical rule for our only transitive verb form, 'curses':

```
tv(lambda(X, lambda(Y, X@lambda(Z, curse(Y,Z))))) --> [curses], {vars2atoms(X),
```

Recall that the λ-expressions for the determiners 'every' and 'a' are $\lambda P.\lambda Q.\forall x.(P@x \rightarrow Q@x)$ and $\lambda P.\lambda Q.\exists x.(P@x \wedge Q@x)$. We express these in Prolog as follows:

```
det(lambda(P, lambda(Q, exists(x, ((P@x) & (Q@x)))))) --> [a], {vars2atoms(P),
det(lambda(P, lambda(Q, forall(x, ((P@x) > (Q@x)))))) --> [every], {vars2atoms
```

Finally, the 'role-reversing' (Section 10.4.4) representation for our only proper name:

```
pn(lambda(P, P@harry)) --> [harry], {vars2atoms(P)}.
```

Prolog Variables?

Note that we break our convention (page 162) of representing variables by constants in these lexical rules. All the λ-bound variables are written as Prolog variables instead of atoms. This is the reason why we have to include the calls to `vars2atoms/1` in curly brackets. As you probably remember from Chapter 5, curly brackets allow us to include further Prolog calls with DCG-rules in some of our phrasal rules. Whenever a lexical entry is retrieved, `vars2atoms/1` replaces all Prolog variables in it by new atoms. Distinct variables are replaced by distinct atoms. We won't go into how exactly this happens - if you're interested, have a look at the code of the predicate. After this call, the retrieved lexical entry is in accord with our representational conventions again.

This sounds complicated - so why do we do it? If you have read the sidetracks in the previous section (Section 10.4.7 and Section 10.4.8), you've heard about the possibility of accidental binding and the need for α-conversion during the semantic construction process. Now by using Prolog variables in lexical entries and replacing them by atoms on retrieval, we make sure that no two meaning representations taken from the lexicon ever contain the same λ-bound variables. In addition, the atoms substituted by `vars2atoms/1` are distinct from the ones that we use for quantified variables. Finally, no other rules in our grammar ever introduce any variables or double any semantic material. In result accidental bindings just cannot happen. So altogether, using Prolog variables in the lexicon is a bit of a hack, but that way we get away without implementing α-conversion.

10.5.4 A First Run

Now this makes semantic construction during parsing extremely easy. Here is an example query:

?- `s(Sem,[harry,flies],[]).`

```
Sem = Sem=lambda(v1, v1@harry)@lambda(v2, fly(v2))
```

Or generate the semantics for 'Harry curses a witch.': `s(Sem,[harry,curses,a,witch],[]).`

The variables `v1,v2` etc. in the output come from the calls to `vars2atoms` during lexical retrieval. The predicate generates variable names by concatenating the letter `v` to a new number each time it is called.

So now we can construct λ-terms for natural language sentences. But of course we need to do more work *after* parsing, for we certainly want to reduce these complicated λ-expressions into readable first-order formulas by carrying out β-conversion. Next, we will implement the predicate `betaConvert/2`, which will do the job.

10.5.5 Beta-Conversion

The first argument of `betaConvert/2` is the expression to be reduced and the second argument will be the result after reduction. Let's look at the two clauses of the predicate in detail. You find them in the file See file `betaConversion.pl`..

```
betaConvert(Functor@Arg,Result):-
        betaConvert(Functor,lambda(X,Formula)),
        !,
        substitute(Arg,X,Formula,BetaConverted),
        betaConvert(BetaConverted,Result).
```

The first clause of `betaConvert/2` is for the cases where 'real' β-conversion is done, i.e. where a λ is thrown away and all occurences of the respective variable are replaced by the given argument. In such cases

1. The input expression must be of the form Functor@Arg,
2. The functor must be (recursively!) reducible to the form `lambda(X,Formula)` (and is in fact reduced to that form before going on).

If these three conditions are met, the substitution is made and the result can be further β-converted recursively.

This clause of `betaConvert/2` makes use of a predicate `substitute/4` (originally implemented by Sterling and Shapiro) that we won't look at in any more detail. It is called like this:

```
substitute(Substitute,For,In, Result).
```

`Substitute` is substituted for `For` in `In`. The result is returned in `Result`.

10.5.6 Beta-Conversion Continued

Second, there is a clause of `betaConvert/2` that deals with those expressions that do not match the first clause. Note that the first clause contains a cut. So, the second clause will deal with all *and only* those expressions whose functor is *not* (reducible to) a λ-abstraction. The only well-formed expressions of that kind are formulas like `walk(john) & (lambda(X,talk(X))@john)` and atomic formulas with arguments that are possibly still reducible. Apart from that, this clause also applies to predicate symbols, constants and variables (remember that they are all represented as Prolog atoms). It simply returns them unchanged.

```
betaConvert(Formula,Result):-
        compose(Formula,Functor,Formulas),
        betaConvertList(Formulas,ResultFormulas),
        compose(Result,Functor,ResultFormulas).
```

The clause breaks down `Formula` using the predicate `compose/3`. This predicate decomposes complex Prolog terms into the functor and a list of its arguments (thus in our case, either the subformulas of a complex formula or the arguments of a predication). For atoms (thus in particular for our representations of predicate symbols, constants and variables), the atom is returned as `Functor` and the list of arguments is empty.

If the input is not an atom, the arguments or subformulas on the list are recursively reduced themselves. This is done with the help of:

```
betaConvertList([],[]).
betaConvertList([Formula|Others],[Result|ResultOthers]):-
        betaConvert(Formula,Result),
        betaConvertList(Others,ResultOthers).
```

After that, the functor and the reduced arguments/subformulas are put together again using `compose/3` the other way round. Finally, the fully reduced formula is returned as `Result`

If the input is an atom, the calls to `betaConvertList/2` and `compose/3` trivially succeed and the atom is returned as `Result`.

Here is an example query with β-conversion:

```
?- s(Sem,[harry,flies],[]), betaConvert(Sem,Reduced).

Sem = lambda(A,A@mary)@lambda(B,walk(B)), Reduced = walk(mary)
```

Try it for ‘Harry curses a witch.’: `s(Sem,[harry,curses,a,witch],[]), betaConvert(Sem,Res).`

?- Question!

Above, we said that complex formulas like `fly(harry) & (lambda(x,fly(x))@harry)` are split up into their subformulas (which are then in turn β-converted) by the last clause of `betaConvert/2`. Explain how this is achieved at the example of this particular formula.

10.5.7 Running the Program

We’ve already seen a first run of our semantically annotated DCG, and we’ve now implemented a module for β-conversion. So let’s plug them together in a driver predicate `go/0` to get our first real semantic construction system:

```
go :-
        readLine(Sentence),
        resetVars,
        s(Formula,Sentence,[]),
        nl, print(Formula),
        betaConvert(Formula,Converted),
        nl, print(Converted).
```

This predicate first converts the keyboard input into a list of Prolog atoms. Next, it does some cleaning up that is needed to manage the creation of variable names during lexicon retrieval (see Section 10.5.3). Then it uses the semantically annotated DCG from See file `semanticDCG.pl`. and tries to parse a sentence.

Next, it prints the unreduced λ-expression produced by the DCG. Finally, the λ-expression is β-converted by our predicate `betaConvert/2` and the resulting formula is printed out, too.

Note: In order to get a nice printout of semantic representations, the definitions for the operators used in them have to be consulted on top-level. So in order to run the program, do the following at a Prolog prompt:

```
1 ?- [semconOperators],[runningLambda].
% semconOperators compiled 0.00 sec, 400 bytes
%   semconOperators compiled into semconHelpers 0.00 sec, 216 bytes
%  semconHelpers compiled into semconHelpers 0.00 sec, 7,232 bytes
%   semconOperators compiled into betaConversion 0.00 sec, 216 bytes
%  betaConversion compiled into betaConversion 0.00 sec, 1,628 bytes
%   semconOperators compiled into runningLambda 0.00 sec, 216 bytes
%  semanticDCG compiled into runningLambda 0.00 sec, 4,092 bytes
%  semconOperators compiled into runningLambda 0.00 sec, 0 bytes
% runningLambda compiled into runningLambda 0.01 sec, 14,184 bytes
```

```
Yes
2 ?- go.

>
```

Code For This Chapter

Here's a listing of the files needed:

See file `semanticDCG.pl`.	The semantically annotated DCG.
See file `runningLambda.pl`.	The driver predicate.
See file `betaConversion.pl`.	β-conversion.
See file `semconOperators.pl`.	Definitions of operators used in semantic representations
See file `semconHelpers.pl`.	Auxiliary predicates.
See file `simplifiedEng.pl`.	Simplified version of `ourEng.pl` from Chapter 8. Used in the ex

10.6 Exercises

Exercise 10.1 *Look at the semantics construction in Section 10.4.4 again. Work through the functional applications and β-reductions required to build the VP and S representations. Make sure you understand the role-reversing idea used in the TV semantic representation.*

Exercise 10.2 *Find a suitable λ-expression for the determiner 'no' and add it to our implementation of λ-calculus. Test your solution with sentences like 'No man walks.'*

Extend `semanticDCG.pl` accordingly.

Exercise 10.3 *Starting off from our treatment of transitive verbs (Section 10.4.4), how would you handle ditransitive verbs such as 'offer'? Give a semantic representation for 'offer', as well as semantically annotated grammar rules to analyse sentences like 'A witch offers Harry a broomstick.'*

Exercise 10.4 *[Project]*

This exercise is about extending the chart parser from Chapter 8 such that it produces semantic representations. Here's a simplified version of the grammar `ourEng.pl` that we used there: See file `simplifiedEng.pl`.. This simplified version contains only the kind of constructions that we already know how to deal with semantically.

1. *Extend the phrasal rules of this grammar with arguments for semantic construction along the lines of See file `semanticDCG.pl`..*
2. *Add a semantic representation to each lexical entry as a third argument. Use Prolog variables for the λ-bound variables.*
3. *Now extend the chartparser from Chapter 8 to deal with the semantic information. To get rid of the Prolog variables in the lexical entries, just add a call to* `vars2atoms/1` *immediately after the lexicon is accessed in* `process_chart_bottomup/0`.

4. *Add relative clauses to the grammar. For the syntactic part, just have a look at See file* `ourEng.pl`*.. As regards the semantics: The predication from the relative clause should simply be conjoined to the predication from the noun that is modified. For instance, the representation assigned to 'man who walks' should look like this when fully* β*-reduced:*

   $$\lambda x.(\textsc{man}(x) \wedge \textsc{walk}(x))$$

   So what you have to do is think of a representation for the relative pronouns that leads to this result when combined with the representations for 'walks' (that is $\lambda x.\textsc{walk}(x)$*) and 'man' (that is* $\lambda x.\textsc{man}(x)$*) in the order determined by the syntax tree.*

Code Index

`a2b.pl`	View[3]_Download[4]	A transducer that translates `a`s int
`aNbN.pl`	View[5]_Download[6]	A grammar generating $a^n b^n \backslash\{\}$.
`active_chart_bottomup.pl`	View[7]_Download[8]	An active bottom-up chart recogn
`adoubler.pl`	View[9]_Download[10]	A transducer that doubles the num
`badDCG4Pronouns.pl`	View[11]_Download[12]	A DCG for simple English senten
`betaConversion.pl`	View[13]_Download[14]	β-conversion.
`bottomup_recognizer.pl`	View[15]_Download[16]	A very naive bottom-up recognize
`cat_parse.pl`	View[17]_Download[18]	FSA-based parser for FSA with c
`dCG4GapThreading.pl`	View[19]_Download[20]	A DCG for simple English relativ
`dCG4Gaps.pl`	View[21]_Download[22]	A DCG for simple English relativ
`dCG4Gaps2.pl`	View[23]_Download[24]	A not very promising extension o
`dCG4Pronouns.pl`	View[25]_Download[26]	A DCG for simple English senten
`dCGExample.pl`	View[27]_Download[28]	A small DCG for very simple Eng
`deterministic.pl`	View[29]_Download[30]	Deterministic and ε free 'laughing
`epsilon.pl`	View[31]_Download[32]	A grammar with an empty produc
`haha.pl`	View[33]_Download[34]	'laughing-machines' from Sectio
`hahuho.pl`	View[35]_Download[36]	Further 'laughing-machines' need
`harry.pl`	View[37]_Download[38]	FSAs for 'Harry Potter-phrases' f
`leftcorner_recognizer.pl`	View[39]_Download[40]	A left corner recognizer.
`leftcorner_recognizer_table.pl`	View[41]_Download[42]	The left corner recognizer using a
`leftrec.pl`	View[43]_Download[44]	A left recursive context free gram
`operations.pl`	View[45]_Download[46]	The complete program for operati
`ourEng.pl`	View[47]_Download[48]	Our English grammar fragment.
`parse.pl`	View[49]_Download[50]	An FSA-based parser.
`passive_chart_bottomup.pl`	View[51]_Download[52]	A bottom-up chart recognizer.
`readAtom.pl`	View[53]_Download[54]	Predicate `readAtom/1` for reading
`recognize.pl`	View[55]_Download[56]	A recognizer/generator for FSAs
`recognize1.pl`	View[57]_Download[58]	A version of the recognizer/genera
`runningLambda.pl`	View[59]_Download[60]	The driver predicate for our imple
`semanticDCG.pl`	View[61]_Download[62]	A semantically annotated DCG.
`semconHelpers.pl`	View[63]_Download[64]	Auxiliary predicates for semantics
`semconOperators.pl`	View[65]_Download[66]	Definitions of operators used in se
`simplifiedEng.pl`	View[67]_Download[68]	Simplified version of `ourEng.p`
`trans.pl`	View[69]_Download[70]	A driver for transducers.
`trans_lex.pl`	View[71]_Download[72]	A driver for transducers using lex
`wrongDCG4Pronouns.pl`	View[73]_Download[74]	A DCG for simple English senten

Bibliography

[1] Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language, A First Course in Computational Semantics*. Draft, 1999. Available at: `http://www.comsem.org/`.

[2] Michael A. Covington. *Natural Language Processing for Prolog Programmers*. Prentice Hall, 1993.

[3] Matthew Crocker. Mechanisms for sentence processing. In S. Garrod and M. Pickering, editors, *Language Processing*. Psychology Press, London, 1999.

[4] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[5] Dan Jurafsky and James Martin. *Speech and Language Processing*. Prentice Hall, New Jersey, 2000.

[6] Martin Kay and Ronald M. Kaplan. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994.

[7] Harry R. Lewis and Cristos H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[8] Ehud Reiter and Robert Dale. *Building Natural-Language Generation Systems*. Cambridge University Press, 2000.

Index

(λ-)bound, 166
α-conversion, 174
α-equivalent, 174
β-reduction, 167
λ-abstraction, 166

abstracted over, 166
active edge, 133
activearc, 133
affix, 21
agenda, 136
alphabet, 4
ambiguity, 73
atomic formula, 159

bottom-up parsing and recognition, 91
bound variable, 160
breadth-first, 99

chart parsing, 98, 120
concatenation, 30
confluence, 169
constant symbol, 158
context free, 72
context free grammar, 71
context free rule, 71

depth-first, 99
derivation, 72
derivation (morphological), 21
dialogue act, 67
dynamic, 126

exponential, 98
extraction, 79

failure driven loop, 126
feature, 76
final state, 2
finite state automaton, 3
finite state generator, 1
finite state machine, 3
first-order language, 158
formal language, 3
free variable, 160
FSA, 3
FSM, 3
functional application, 167
fundamental rule, 134

gap, 79

inflection, 21
input tape, 3

jump arc, 4

Kleene closure, 30

language accepted, 3
language generated, 3
language generated by a context free grammar, 72
left corner, 111
left-corner parsing, 111
left-corner table, 115
lexical rule, 73
local ambiguity, 75

matrix, 159
meaning representation, 157
morpheme, 21
movement, 79

name, 158
non-deterministic, 5
non-terminal symbol, 71

Occurs-Check, 88
output tape, 3

parse tree, 73
passive arc, 134
passive chartparsing, 120
passive edge, 134
phrase structure grammar, 74
phrase structure rule, 73
polynomial, 98
predicate symbol, 158
preterminal symbol, 74

recognizer, 3
regular expression, 31
regular language, 31
relation symbol, 158
restriction, 170
rewrite arrow, 71
rewrite interpretation, 72
rewrite rule, 71

scope, 159, 170
search, 5
sentence, 161
sentence symbol, 71
speech act, 67
spurious analysis, 84
start state, 1
start symbol, 71
stem, 21
subcategorize, 80
syntactic structure, 163

tape, 18
term, 159
terminal symbol, 71
top-down parsing/recognition, 99
transition, 1
tree admissibility rule, 73

well-formed formula, 159