LISA: A Linguistic Information System

Richard Bergmair

Keplerstrasse 3

A-4061 Pasching

rbergmair@acm.org

Aug-02 - May-03

Final Draft, printed September 14, 2004

Abstract

The prototype-part of this paper describes a system enabling computing machinery to understand a problem stated in everyday English and to solve it.

A complex problem is stated purely in natural language, as if it was intended to be read by a human. The system uses linguistic knowledge of English words and the English grammar as well as semantic knowledge of the problem domain it operates in to understand it.

By "understanding the problem", we mean any kind of processing that can be applied to the natural-language problem-representation, which equips the system with the knowledge needed to actually solve it. Such an "understander" must not use any problem-specific knowledge explicitly stated by the programmer in a machine-readable form, knowledge about the problem is supposed to be inferred solely from its natural-language representation.

This requires analyzing each word that appears in the input, a task known as "morphological analysis" in Natural Language Processing, which is related to that of "lexical analysis" or "lexing" in compiler-construction. It is carried out in our prototype by machine-code which is generated by a special compiler.

After words are morphologically analyzed, they are translated into a structured representation that deals with the English language's grammar. Building up such a representation is a parser's job. Our parser is a highly performance-optimized implementation of the algorithm suggested by Earley (1970). It uses dynamic programming to parse its input in polynomial time, regardless of ambi-

guity or left-recursion in its grammar. In addition to the parser our implementation features an event-based traverser for this data-structure, so a third-party programmer can easily reuse and extend the parser.

Our parser is also capable of creating a representation that can be loaded into a PROLOG-interpreter. Knowledge about the English language as well as knowledge about the problem domain is also provided to the interpreter, so it can execute a query that solves the problem, leaving semantic reasoning mainly up to PROLOG.

The theoretical part of this paper aims to introduce the reader to the field of Natural Language Processing (NLP for short). As it is targeted towards readers from a technical or Computer Science-background, surveys of many introductory-level concepts from the field of Linguistics are given. It also contains summaries of much of the scientific work that became the basis for our prototype, and many other NLP-systems. The most important concepts introduced to the reader are finite state automata and finite state transducers as well as their application to morphological analysis, context-free grammars, including their augmentations with feature-structures and semantic attachments and how to use them for parsing. Furthermore the most important symbolic approaches to semantics including lexical semantics and general-purpose problem-solving are introduced.

Preface

Looking at science-fiction literature it can clearly be seen, that one idea has been on people's minds, ever since computing machinery entered the scene: communicating with a machine in natural language. Because of the continuing evolvement of this field, much of that is now rather science than fiction. But still: There was no HAL-9000 computer available in 2001, as Stanley Kubrick imagined, and building one confronts Computer Sciences with a great challenge that has yet to be taken.

The first part of this paper aims to take the fictional idea of interfacing an information system as a social entity to a scientific level. It introduces many of the problems researchers in the field of Computational Linguistics have to deal with, and the techniques these problems are traditionally addressed by.

The second part describes the most interesting parts of a prototype actually solving a problem stated in English. The overall structure and the problem domain this prototype was built for is introduced, and the two parts of the program that were most challenging, from a programmer's point of view are documented with sourcecode in full detail.

Contents

Part I

NLP: A Summary of Traditional Approaches

Chapter 1

Introduction

> *HAL:* Hey, Dave, what are you doing?
>
> Bowman works swiftly.
>
> *HAL:* Hey, Dave. I've got ten years of service experience and an irreplaceable amount of time and effort has gone into making me what I am.
>
> Stanley Kubrick and Arthur C. Clarke
>
> 2001: A Space Odyssey

This chapter aims to give a rough introduction to the field of computational linguistics. Is it really possible to create an artificial agent, capable of such advanced language-capabilities as speaking English? What would it take to build a computer like HAL? These are the questions we will be addressing in the subsequent sections.

1.1 Overview

Let's first consider an example. Suppose we confront HAL with the following sentence:

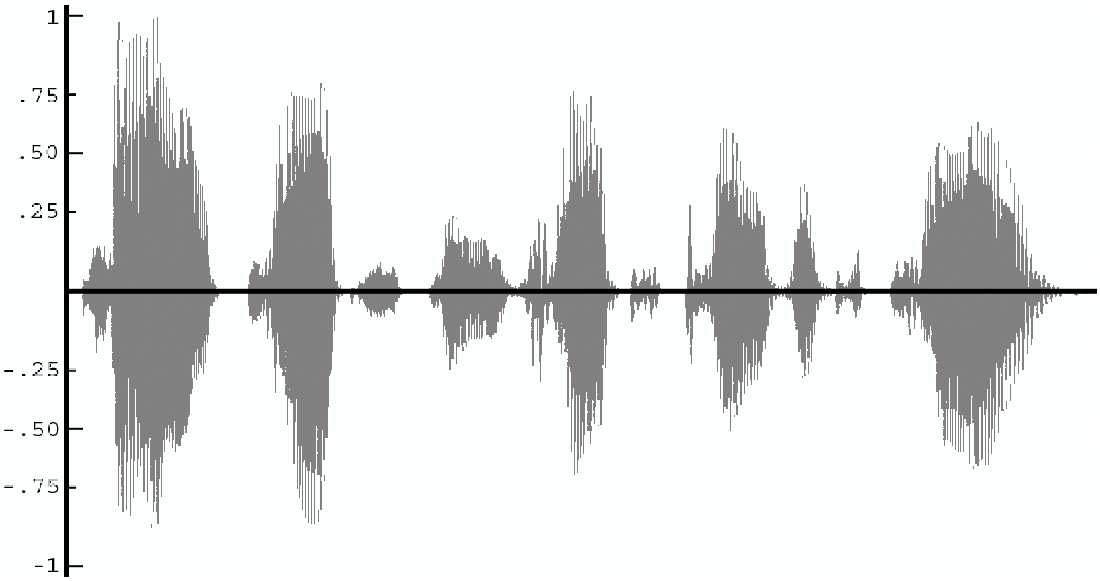


Figure 1.1: An oscillogram of the utterance *Joe taught steve to play the guitar*

(1.1) Joe taught Steve to play the guitar.

First of all it is important to understand the different representations the information from example 1.1 goes through, beginning with a sound-wave and ending in a semantic representation of the concepts behind the utterance.

Sound is what figure 1.1 shows. It is an oscillogram[1] of the utterance from example 1.1. It is a graphical representation of the input a sound-device gets when recording voice with a microphone.

Speech recognition is what HAL would have to be doing to convert this representation of a sound-wave into a "written" representation, or, putting it more accurately, a string of morphemes, which could be called the "nuclear" unit of speech and language, that can be matched against a dictionary in order to obtain a written representation. Today powerful speech-recognition-systems, which

[1]In our case: graph of pressure fluctuation versus time.

perform exactly that task, are available commercially at a broad range, handling many different languages, specialized vocabularies and difficult recording-situations. This is why we will not deal with speech-related issues in this paper, but rather start with a string-representation of written language.

An ASCII-coded string is therefore the first representation our system would be confronted with.

Morphological analysis is what the next step is sometimes referred to. Now that a string of words making up a sentence is available, each word being a string of characters, we can go about analyzing the words. In this step the system would have to find out that "taught" is a past-tense form of the verb "teach", etc.

An intermediate representation could be used to pass data from the morphological analysis to the syntactic one. Usually this isn't neccessary because, in practice, morphological and syntactic analysis are often handled in a highly integrated way.

Syntactic analysis is the process of putting the words into a more structured form, taking into account the grammar of the language.

A syntax-tree could be a way to represent output from the syntactic analysis. Such a tree could make statements like, "This sentence consists of a subject in nominative singular S, a predicate P and two accusative-objects O_1 and O_2, S being *Joe*, P being *taught*, O_1 being *Steve*, and O_2 consisting of the function word *to*, the verb V, the function word *the*, and a noun N, V being *play* and N being *guitar*".

It is important to keep in mind that this is only one possibility. The output of a syntactic analysis could instead make statements like, "This sentence is an

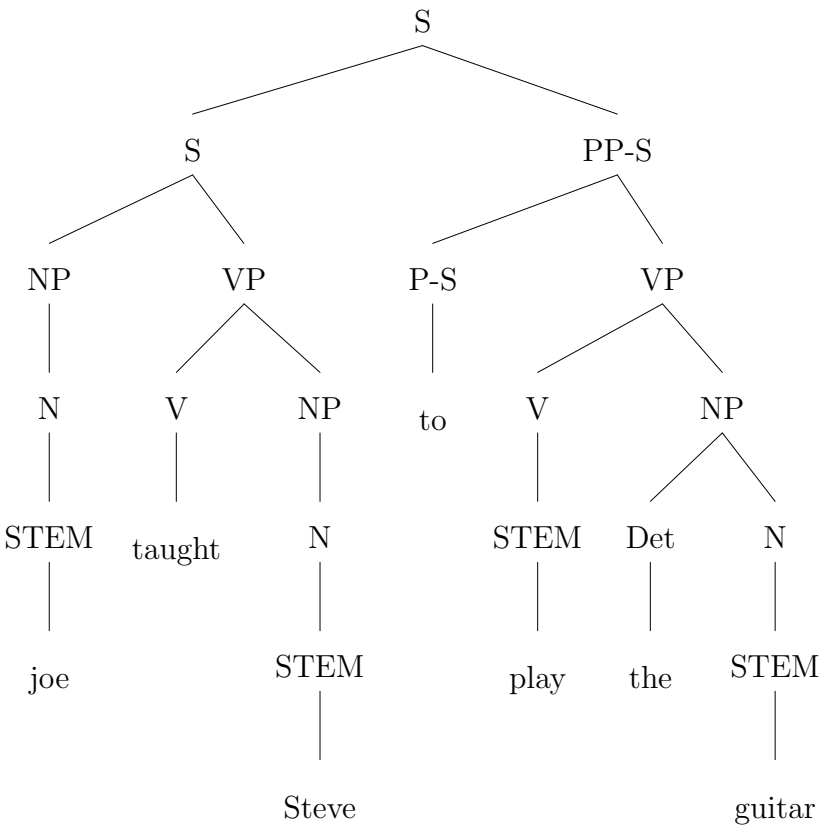


Figure 1.2: A syntax tree based on the ERG

active-voice sentence, the agent being *Joe*, the experiencer being *Steve*, the action being *teach*, etc.".

What exactly such an output looks like is highly dependent on the grammatical framework of choice. Figure 1.2 shows an output based on a syntactic analysis carried out using the ERG (English Resource Grammar: as distributed by Stanford's LinGO-initiative).

Semantic analysis is a more accurate term of what is commonly seen as the "understanding"-part of NLU (Natural Language Understanding). Whether a computer will ever be able to truly "understand" a meaningful sentence is subject to broad discussions in the field of AI-research and philosophy. For now, let's just settle with the rather pragmatic approach to semantics Winograd (1971, p281) used.

> A semantic theory must describe the relationship between the words and syntactic structures of natural language and the postulated formalism of concepts and operations on concepts.

Semantic representation If we chose, for example, First Order Predicate-Calculus (FOPC, for short) as a semantic representation, the output of the semantic analysis could be something like

$$\forall g \exists e, pIsa(e, Teaching) \wedge Teacher(e, Joe) \wedge Student(e, Steve) \wedge Subject(e, p) \wedge$$
$$Isa(p, PlayInstrument) \wedge Instrument(p, g) \wedge Isa(g, Guitar)$$

This notation could be read like "for every g there exists an e and a p, such that e is the event of teaching, the teacher participating in e being Joe, the student participating in e being $Steve$ the subject being taught in e being p, p being the event of playing an instrument, the instrument in p being g, and g being any $Guitar$".

Again FOPC is only one way of representing semantic data and the actual semantic representation is dependent on the semantic model of choice. Some semantic models don't even require a semantic representation at all.

1.2 Ambiguity

One of the most difficult tasks in discovering the meaning of a sentence is to choose between the meanings it could possibly have. Usually in a given situation a sentence can only be assigned one meaning that is plausible, but how is an artificial agent to decide upon the "plausibility" of an interpretation of a sentence?

Consider the following sentence from Schank (1971)

> We saw the Grand Canyon flying to Chicago.

There are many interpretations that could be assigned to this sentence. Here are some of them:

- While we were flapping our wings, flying to Chicago, we saw the Grand Canyon.
- We saw the Grand Canyon, which was travelling in an airplane to Chicago.
- When travelling to Chicago in an airplane, we saw the Grand Canyon.

That there are multiple interpretations for this single sentence is due to ambiguities on almost every level of language processing: sense-ambiguity, for example. The word *fly* can be used in the sense of *travel by airplane* as in *We flew to Chicago*, or in the sense of flying as in *Birds fly*.

Another example for ambiguity is structural ambiguity. In order to understand the above sentence, HAL will have to decide where the gerundive phrase *flying to Chicago* should be attached. It can either be part of a gerundive sentence, whose subject is *the Grand Canyon*, or it can be an adjunct modifying the

phrase headed by *saw*, leaving either *We* as the ones who perform the action of flying, or *the Grand Canyon*, that does the flying.

This task of choosing the right interpretation is called "disambiguation", and many of the problems researchers in the field of NLP are concerned with are instances of disambiguation-problems.

1.3 Knowledge

Disambiguation often requires the machine to have knowledge about the "world" it operates in. A machine operating in a so called "real-world-environment", like HAL, would therefore need substantial knowledge of the real world. HAL has to be aware of facts such as, that people have no wings, and can only fly by plane, or that the Grand Canyon cannot fly, neither by plane, nor by flapping its wings.

Giving HAL such knowledge is probably one of the most difficult tasks AI-research has to face. John McCarthy, one of the big names in AI, has done remarkable research in that area. The reader is referred to his book McCarthy (1990) and especially to some of his papers McCarthy (1958), McCarthy & Hayes (1969), McCarthy (1977, 1989) and the paper about his 1971 lecture, for which he was awarded the Turing Award McCarthy (1987).

Chapter 2

Morphology

> The major problem [in time travel] is quite simply one of grammar, and the main work to consult in this matter is Dr Dan Streetmentioner's *Time Traveller's Handbook of 1001 Tense Formations*. It will tell you for instance how to describe something that was about to happen to you in the past before you avoided it by time-jumping forward two days in order to avoid it. The event will be described differently according to whether you are talking about it from the standpoint of your own natural time, from a time in the further future, or a time in the further past and is further complicated by the possibility of conducting conversations whilst you are actually travelling from one time to another with the intention of becoming your own mother or father.
>
> Most readers get as far as the Future Semi-Conditionally Modified Subinverted Plagal Past Subjunctive Intentional before giving up: [...]
>
> Douglas Adams
> The Restaurant at the End of the Universe

2.1 Overview

(2.1) After playing for hours the guitarists recharged their tuner's batteries.

Example 2.1 aims to introduce the reader to the most important scenarios in morphological processing.

A computer-program attempting to understand this sentence would have to know each word in the sentence, but how is a computer supposed to understand the word, or rather, the substring *recharged* from the above string? It certainly wouldn't find it in any dictionary (in the sense of a string-array listing word-forms). It would be nonsensical to build up a dictionary containing all thinkable forms of a free morpheme like *charge*, since it would have to list

- (a) charge
- (more) charges
- (to) charge
- (He) charges
- (It is) charging
- (Yesterday I) charged
- (to) recharge
- (He) recharges
- (It is) recharging
- (Yesterday I) recharged
- (it is) charged
- (it is) uncharged

- (it is) unchargeable
- ...

Such a dictionary wouldn't only use up masses of memory, it would also be completely unmaintainable.

It would be desirable to have a dictionary list only-so called "root forms", like *charge*, and equip the system with rules like

1. It can be used as a verb
2. It can be used as a noun
3. It can be used as an adjective
4. The past-tense form of the verb can be derived by attaching the suffix *-ed*
5. The third-person-singular form of the verb can be derived by attaching the suffix *-es*
6. The progressive form of the verb can be derived by attaching the suffix *-ing*
7. The prefix *re-* can be attached to the verb
8. The plural form of the noun can be derived by attaching the suffix *-es*.
9. The prefix *un-* can be attached to the adjective
10. The suffix *-able* can be attached to the adjective

Rule 4 applies to almost every verb in the dictionary. (We will use the term "in the dictionary" to actually describe the concept of "either listed in the dictionary or derivable from a form in the dictionary".) Therefore the possibility of storing such rules centrally eliminates a considerable amount of redundancy from the system.

This doesn't only save physical storage-capacity, it also leaves the system of morphological rules and their applications as an independent module, which has many advantages. One of them is giving the system the ability to apply known rules to new words. If a human listener is confronted with new words he has never heard before, say *a great gardel* and *a big red tarivar* he can assume that *I gardelized my tarivar* could possibly mean "I turned my tarivar into a gardel", regardless of what the words are supposed to mean.

2.2 A Linguistic Perspective to Morphology

So far we have seen the need for a dictionary listing root-forms and morphological rules, and a system capable of applying the rules to words in the dictionary. In this section we will have a closer look at some examples of morphological rules in order to give the reader a rough idea of English morphology and the difficulties it confronts a non-human understander with. A more detailled description can be found in Weisler & Milekic (2000, pp79ff).

2.2.1 Derivational Morphology

Rules of derivational morphology are simply rules "deriving" more complex word-forms from simpler ones, often altering their meaning or syntactic category.

One of the most prominent rules of derivational morphology is probably the appending of the suffix *-ly* to derive an adverb from an adjective. This rule can be applied to almost any adjective, with three consequences:

- the substring *ly* is appended to the word-form
- the word-form is now an adverb, rather than an adjective
- the meaning of the word changes to "in an X manner"

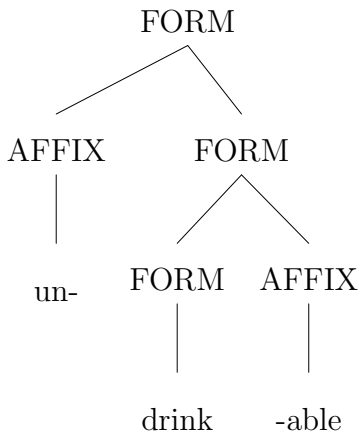


Figure 2.1: A tree showing the derivation of *undrinkable*

Another rule of derivational morphology uses the suffix *-er*, to derive *painter* from *(to) paint*. It alters the meaning to "someone who Xs".

In the case of *-ist*, it gets obvious, that not every rule can be applied to every word form. A *guitarist* is someone who plays the guitar, a *violinist* is someone who plays the violin, but no one has ever heard of a **drumist*.

Considering the suffix *-able*, turning a verb into an adjective, leaving the meaning as "it is possible to X it.", and the prefix *un-*, altering the meaning of an adjective to "not X" makes clear that it is also possible to subsequently apply rules to forms that are already morphologically derived from a root form. The form *undrinkable* can only be derived by first deriving *drinkable* from *drink*. The *un-*-rule can then be applied to the new form *drinkable* to produce *undrinkable*. Figure 2.1 shows a tree-representation of this concept.

Some more complications can be seen when, for example, considering the *de-* prefix, altering the meaning of a verb to "to reverse the action of Xing". Words like *deflate* suggest that not all root-forms actually exist, given a derived form that exists. **flate* is obviously not an English word, although the *de-* prefix seems to exactly behave like a morphological rule. That the morpheme **flate* might actually exist is further suggested by the evidence that forms like *inflate*

and *inflation* can be derived from it.

The word *deodorize* seems to be of similar nature, but this time the unbound morpheme *odor* does exist as a root form, while the derivation **odorize* does not exist as a word-form. Yet it is still possible to do further derivations, like applying the *de*--rule, leaving the form as *deodorize*, which is again an existent word-form.

delete and *depress* are similar cases. While *delete* does not seem to have anything to do with a morphological derivation, but only happens to start with *de* "by chance", the form *press* does exist, yet the application of the *de*--rule doesn't derive that meaning of *depress*.

2.2.2 Inflectional Morphology

While the manifestation of derivational morphology is usually limited to a change in its written form (usually an affix), a change of syntactic category and a change in meaning, inflectional morphology serves a rather different purpose. Rules of inflectional morphology produce an extremely regular semantic effect by modulating certain grammatical aspects of meaning, such as person, number, tense, case, etc. They are usually more regular than rules of derivational morphology and they never change syntactic category.

Nominative Sg.	filia (daughter)
Nominative Pl.	filiae (daughters)
Genetive Sg.	filiae (daughter's)
Genetive Pl.	filiarum (daughters')
Dative Sg.	filiae (I told my daughter something)
Dative Pl.	filiis (I told my daughters something)
Accusative Sg.	filiam (I love my daughter)
Accusative Pl.	filias (I love my daughters)
Vocative Sg.	filia ('Daughter, I love you!')
Vocative Pl.	filiae ('Daughters, I love you both!')

Ablative Sg.	filia (N/A)
Ablative Pl.	filiis (N/A)

Table 2.1: Inflectional table of the latin word "filia"

Generally nouns inflect for case and number, but case is usually neglected because it doesn't alter the appearence of a word-form in English. Table 2.1 is an inflectional table of the latin word *filia, -ae*. In Latin the word form depends on its case, that means it depends on how and where the form is used in a sentence. Therefore *daughter* as in *I love my daughter* is a different word-form as in *I've told my daughter a thousand times not to do that.*

While case is widely irrelevant for English morphology, number isn't. The plural form of *dog*, *dogs* can be derived by applying a rule of inflectional morphology, in this case the suffix *-s*. Again exceptions like *foot/feet*, *goose/geese* or *fish/fish* complicate things. Plural-nouns are another exception. These are nouns that are understood to only make sense in a plural form like *jeans*, and mass-nouns, that are understood to describe an uncountable amount of something, and are only valid in their singular forms, like *money.*

The inflectional system of verbs is a lot more complex, because verbs inflect for person, number, tense and in many languages for features like whether it is used in a conjunctive construction. The German language for example has an indicative and two different conjunctive forms of a verb and six tenses, leaving each word form with $3 * 2 * 6 * 3 = 108$ possible inflections. Fortunately they are highly redundant.

A typical example of verbal inflection is the rule appending *-ed* to a verb, in order to derive a past-tense word-form. Again exceptions are present like *teach/taught*, *catch/caught* or *take/took* and numerous other irregular forms.

2.3 A Naive Approach to Model Morphological Knowledge

The reader should by now have a picture of what morphology is all about and what kind of data is needed to do morphological analysis, but how is it modeled? How do we get a computer to recognize a word like *undrinkable*, given a dictionary-entry *drink*, a rule for *-able* and a rule for *un-*? How do we store such data physically?

Let's start with some morphological data for a parser in a simple problem-domain, say animals.

What we need is a dictionary listing root-forms:

- bird
- cat
- dog
- fish
- frog

In order to inflect the words for number we need rule R_1

1. Rule R_1 can be applied to any form in the dictionary but 'fish'
2. When applied, R_1
   (a) appends the string s to the root-form
   (b) changes the number to PLURAL.

And in order to handle the word *fish* we need another rule R_2

1. Rule R_2 can be applied only to the form *fish*.

2. When applied, R_2

   (a) changes the number to PLURAL.

Then we want to model some derivational morphology, let's call it "babytalk", producing forms like *doggie*, *fishie* or *froggie*.

1. Rule R_3 can be applied to the forms *bird* and *fish*

2. When applied, R_3

   (a) appends the string *-ie* to the root-form

   (b) sets the "babytalk-mark" to TRUE.

1. Rule R_4 can be applied to the forms *dog* and *frog*

2. When applied, R_4

   (a) appends the string *-gie* to the root-form

   (b) sets the "babytalk-mark" to TRUE.

1. Rule R_5 can be applied to the form *cat*

2. When applied, R_5

   (a) changes the form to *kittie*

   (b) sets the "babytalk-mark" to TRUE.

A model like this might already be directly implementable using a rule-based inference-system like PROLOG, but in order to achieve better performance, let's "precompute" some values, and put our morphological model in a procedural terminology, defining the functions as shown in table 2.2.

P_1 NUM=PLURAL

P_2 NUM=PLURAL

P_3	BABYTALK=TRUE
P_4	BABYTALK=TRUE
P_5	BABYTALK=TRUE

Table 2.2: Function-definitions of the procedures P_1 through P_5

In table 2.2 we simply turned the rules into procedures. Instead of rule R_1 which requires the grammatical numerus (NUM) to be PLURAL, we now have a procedure P_1, which carries out this action, namely assign a global-flag (which we'll call NUM) the value PLURAL.

Given these function-definitions we can show the return-values of the possible function-calls to be:

$$bird$$

$$cat$$

$$dog$$

$$fish$$

$$frog$$

$$birdie \leftarrow P_3(bird)$$

$$fishie \leftarrow P_3(fish)$$

$$doggie \leftarrow P_4(dog)$$

$$froggie \leftarrow P_4(frog)$$

$$kittie \leftarrow P_5(cat)$$

$$birds \leftarrow P_1(bird)$$

$$dogs \leftarrow P_1(dog)$$

$$frogs \leftarrow P_1(frog)$$

$$cats \leftarrow P_1(cat)$$

$$fish \leftarrow P_2(fish)$$

$$birdies \leftarrow P_1(birdie) \leftarrow P_1(P_3(bird))$$

$$fishies \leftarrow P_1(fishie) \leftarrow P_1(P_3(fish))$$

$$doggies \leftarrow P_1(doggie) \leftarrow P_1(P_4(dog))$$

$$froggies \leftarrow P_1(froggie) \leftarrow P_1(P_4(frog))$$

$$kitties \leftarrow P_1(kittie) \leftarrow P_1(P_5(cat))$$

While *bird* is itself a word-form, P_3 derives $birdie \leftarrow P_3(bird)$, and P_1 derives $birds \leftarrow R_1(bird)$. The new form *birdie* can again be used as an argument to P_1, this time deriving $birdies \leftarrow P_1(birdie)$, or, putting it differently, $birdies \leftarrow P_1(P_3(bird))$

If we group the above list by procedures, listing only arguments and return-values, we get table 2.3.

P_1	$NUM = PLURAL$	$birds \leftarrow bird$ $cats \leftarrow cat$ $dogs \leftarrow dog$ $frogs \leftarrow frog$ $birdies \leftarrow birdie$ $fishies \leftarrow fishie$ $doggies \leftarrow doggie$ $froggies \leftarrow froggie$ $kitties \leftarrow kittie$
P_2	$NUM = PLURAL$	$fish \leftarrow fish$
P_3	$BABYTALK = TRUE$	$birdie \leftarrow bird$ $fishie \leftarrow fish$
P_4	$BABYTALK = TRUE$	$doggie \leftarrow dog$ $froggie \leftarrow frog$
P_5	$BABYTALK = TRUE$	$kittie \leftarrow cat$

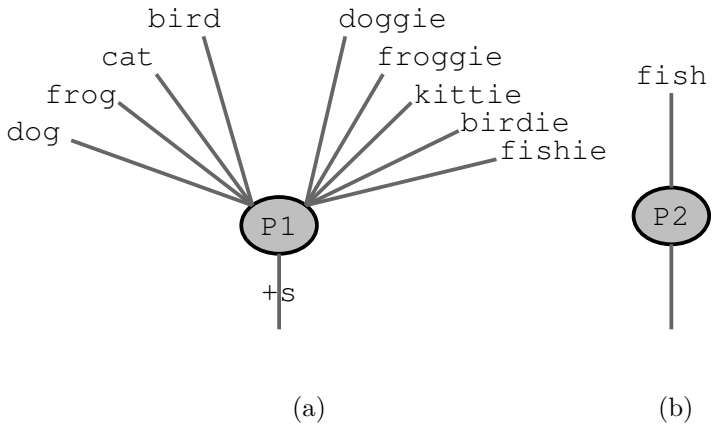


Figure 2.2: The subsystem handling plural-inflection

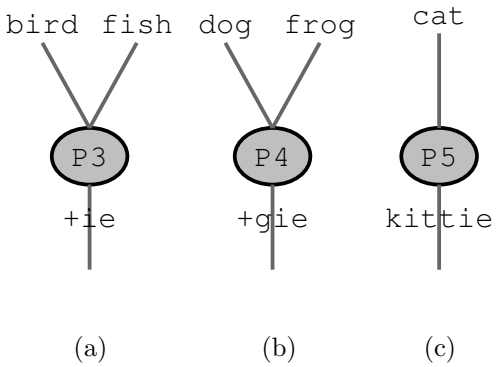


Figure 2.3: The subsystem handling babytalk-derivation

The graphical representations from figures 2.2 and 2.3 show the same data from table 2.3, namely functions and their input and output-values.

Figures 2.2 through 2.8 make use of the convention that unlabelled arcs pass whatever they received as an input to the called function. This value can also be referenced explicitly with the symbol "+", so the appending of an affix can be effectively depicted.

Figure 2.2 shows the functions needed for inflectional, figure 2.3 the ones needed for derivational morphology.

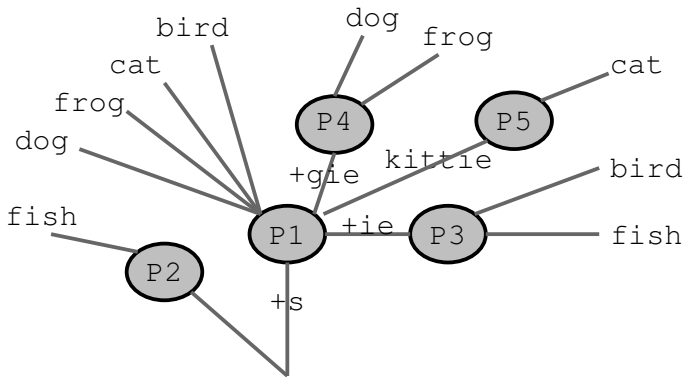


Figure 2.4: A more detailled version of figures 2.2 and 2.3

The next question we ask ourselves, or rather the model, is: Where does the data come from? Where does it go to? If we compose figures 2.2 and 2.3 into figure 2.4, we get a model capable of answering that. Note that so far we have never added any data to the model, we have simply rearranged it. In figure 2.4 it is still possible to make out each "subdiagram" as shown in figure 2.2 or 2.3.

Next we need a final shift in perspectives:

We propose the function q_0 that "produces" all of the input to our system, and some functions $fish$, $bird$, cat, ... each of which gets as input the atomic values. 'fish', 'cat', 'frog', ... This is shown in figure 2.5. The new functions are depicted using a double circle because they do not neccessarily have to do any output. They could simply "swallow" their input.

Next we have to account for plural inflection. This can be done by simply copying the plural-inflection-related parts of the diagrams from figure 2.2 into figure 2.5. The outcome is shown in figure 2.6. Again arcs depict function calls and they are labeled corresponding to the data they pass. We insert the new procedure q_1, which "is interested" in all plural-forms.

Since we also want to cope with babytalk-derivation we do the same for the procedures P_3, P_4 and P_5, in other words copy figure 2.3 into figure 2.5, to get figure 2.7, proposing a function q_2 getting the babytalk forms.

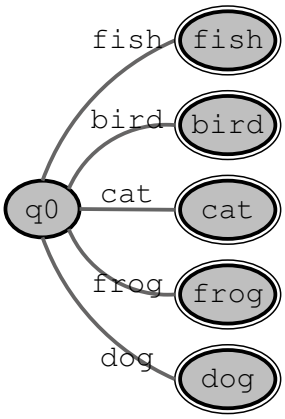


Figure 2.5: A model for basic input

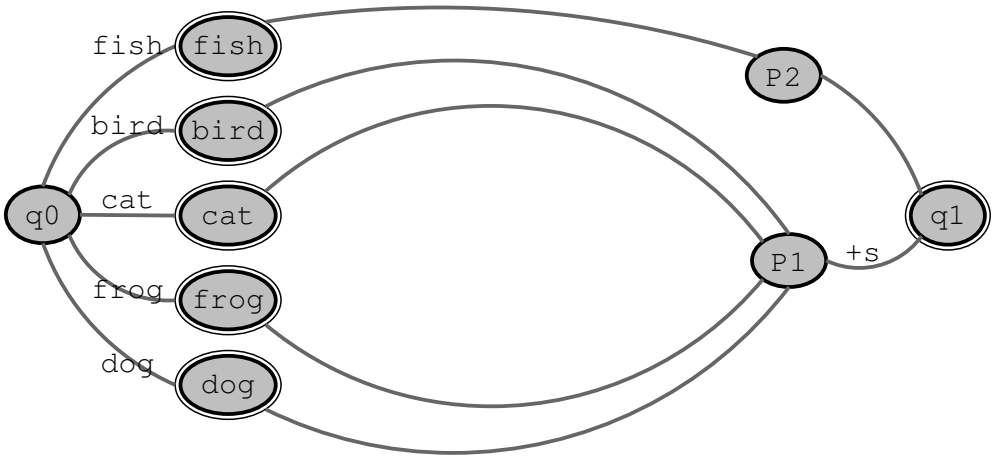


Figure 2.6: A model for basic and plural-inflected input

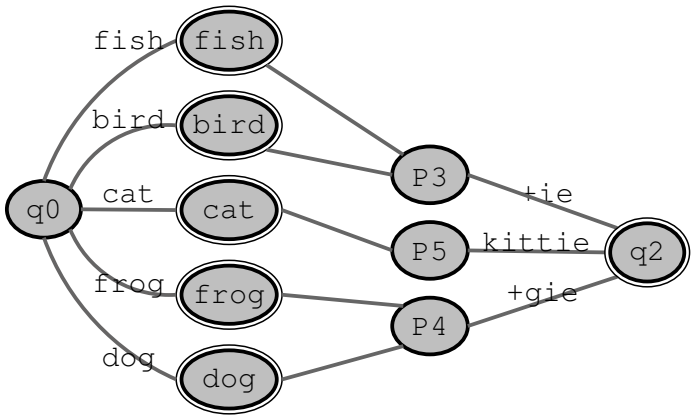


Figure 2.7: A model for basic input and derived babytalk-forms

Figures 2.6 and 2.7 can be easily integrated into figure 2.8, to give a system handling both plural-inflection and babytalk-derivation.

Still we haven't added any information to the model, that couldn't be found in our initial rule-based system described by R_1 through R_5. We have step by step transformed our rule-based approach to a procedural one, and finally into an automaton, as described in the next section.

2.4 Finite State Automata

Although it's syntactically not quite correct, conceptually figure 2.8 can already be interpreted as what is called a "Finite State Automaton", FSA for short.

This section will give the reader a rough introduction to FSAs. More detailled information on FSAs and their applications in Speech and Language Processing can be found in Jurafsky & Martin (2000). Readers already familiar with FSAs might want to skip it.

The first approach to FSAs might come to our minds, when trying to build an effective model for accessing a simple list of words. Think of how one usually looks up a word in the dictionary, say we are trying to look up *dictionary* in our

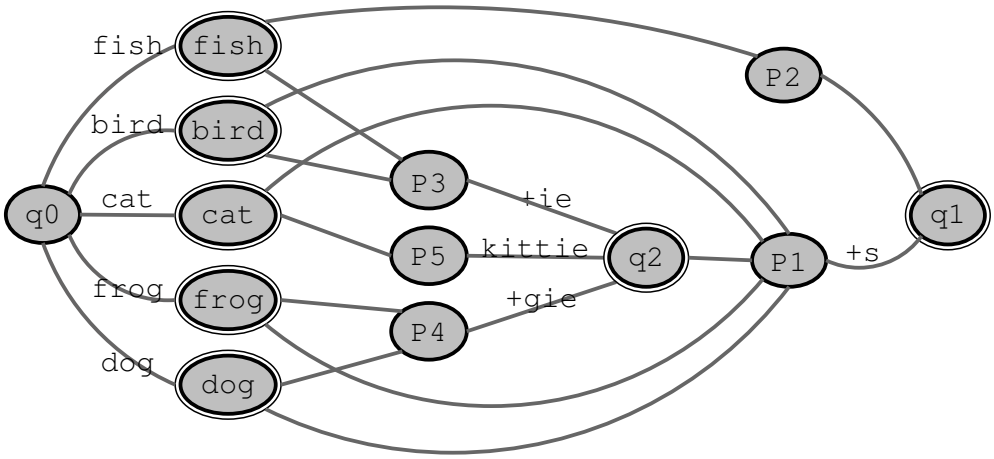


Figure 2.8: A model for the whole system

dictionary.

1. For all items in the dictionary:
2. Find an item that begins with *d*.
3. If there is no item that begins with *d*, *dictionary* can't be in the dictionary.
4. For all items that begin with *d*
   (a) Find an item that begins with *di*.
   (b) If there is no item that begins with *di*, *dictionary* can't be in the dictionary.
   (c) For all items that begin with *di*:
      i. Find an item that begins with *dic*
      ii. ...
      iii. If there is no item that is *dictionary*, *dictionary* can't be in the dictionary.
      iv. If there is an item that is *dictionary*, *dictionary* is in the dictionary.

The recursive nature already gets obvious here.

Now think of a dictionary containing some root-forms and for each one a unique root-form-ID, that identifies the root-form in further processing, as shown in table 2.3.

xabcde	1001
xabfgh	1002
nnki	1003
aabcde	1004
aabfgh	1005
xarki	1006

Table 2.3: A sample dictionary

Figure 2.9 shows a tree-representation of the same table, that fits the above algorithm much better. Now what makes this tree-representation of the above recursive-algorithm and FSA? The interpretation does. One starts at node q_0. Nodes depict the model's idea of "states", so we say, "The automaton IS in state q_0". If the word to be looked up begins with x, one simply follows the arc to q_6, or putting it in a more professional terminology; "The automaton takes the transition to q_6" because arcs depict possible transitions. If the next character is an a, the automaton takes the transition from state q_6 to q_5, etc.

Note that this tree representation is already an "FST", not just an FSA. FST is short for "Finite State Transducer", and it is similar to the idea of an FSA. The only difference is that the transducer has the ability, not just to match a given input-stream, but also to recode it to an output-stream. In our example this can be seen for example at the transition between states q_5 and q_{26}. This transition is the first one, where it is completely determined, that if the input should match an entry in the transducer the output is going to be *1006*, the *xarki*-entry's root-form-ID. The new syntax is this: When a transition is labelled

i:o, then the transition is taken as soon as the input-signal i is read from the input-stream, and whenever that transition is taken the signal o is sent to the output-stream.

Note that this paper doesn't follow the normal convention in that respect. Usually an FST is understood to recode an input-stream to a similar but slightly modified output-stream, which is why the convention that a transition labelled i, means *i:i* is widely used. Since the FSTs we are handling aren't slightly recoding an input-signal, but rather mapping two sets of values to each other it is more practical to follow the convention that i actually means $i{:}\epsilon$, which leads us to a new syntactic element of FSTs and FSAs: The ϵ-signal.

ϵ matching an input-stream means "don't read any signals from the input-stream, simply follow the transition", ϵ feeding an output-stream means "don't do any output". Its meaning is roughly related to the concept of the "empty string", which might be more accessible to readers from a technical background.

In figure 2.9 it is apparent that the subtrees headed by q_3 and q_{17} are completely redundant, as well as the ones headed by q_9 and q_{23}, and the ones headed by q_{26} and q_{12}. q_1, q_7, q_{24}, q_{10}, q_{15} and q_{21} are also completely equal. These redundant nodes or subtrees can now be removed by showing them only once in the diagram, and referencing them in all "situations" needed. The outcome of such a proceeding is shown in figure 2.10.

Figure 2.10 shows us the power of the FST. What we have created is a compact data-structure, highly optimized towards performant lookup. A system checking whether *xabcde* is in the FST would simply traverse it. As soon as it gets to a final state (q_{10} in our example), the string is matched. By the time it's matched, the output will already be ready in the output-stream, and the time used to do the traversal is quite short: It can be expressed as shown in Equation 2.1.

$$\overline{T}(L) \simeq L * \frac{num(states)}{num(transitions)} + O + F \tag{2.1}$$

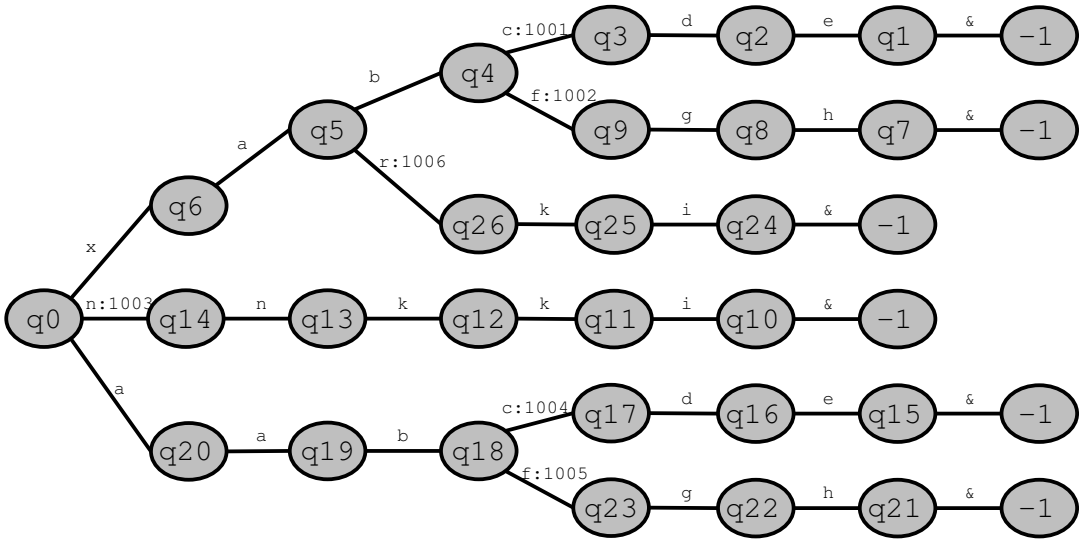


Figure 2.9: A dictionary as a tree

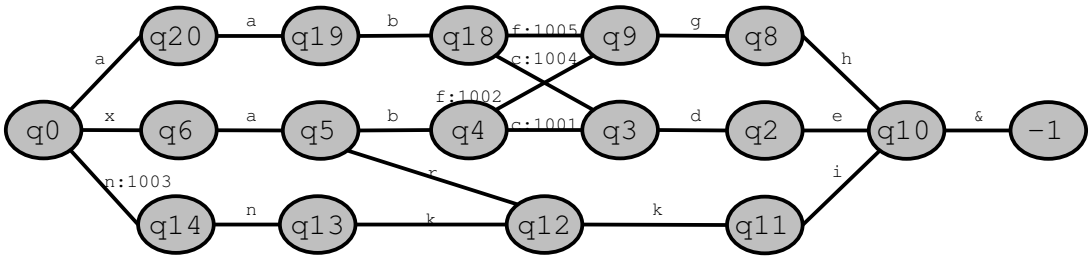


Figure 2.10: The same dictionary as FST

The average time taken to match a string that is in the FST is directly proportional to the product of its length L, and the branching factor B, because the time taken to match a whole string is directly proportional to the time it takes to process a single state and to the number of states that are to be processed. The average time taken to process a single state is directly proportional to the branching factor B, that is the average count of transitions leaving a single node, therefore B is the count of states in the FST divided by the count of transitions. Of course the actual branching-factor along that arc is dependent on the string itself. The number of states to be processed is usually (in the simplified framework presented here) the number of characters in the string. The variable O is symbolic for a constant summand, accounting for the time it takes to do the output. F accounts for the constant amount of time it takes the framework to enter and leave the FST.

The reader interested in a more detailed and technically sophisticated description of FST-processing is referred to the sourcecode-documentation of LISA's FST-toolkit, presented in the second part.

2.5 Finite State Morphological Parsing

In the previous section we saw how to model a simple dictionary using an FST. It has already been mentioned that figure 2.8 can already be viewed as an automaton. Although it is syntactically not quite correct, conceptually it can be interpreted just like an automaton, given that the unlabelled arcs are ϵ-transitions.

Say we wanted to do morphological analysis of the string *birdies*: The automaton would start in state q_0, then read the substring *bird*, then take the ϵ-transitions to P_3, then read the substring *ie*, take the transition to q_2, then take the ϵ-transition to P_1, read the last substring *s*, and get to the final state q_1. If we give procedures *bird*, P_1 and P_3 the ability to set the global flags ROOTFORM=BIRD, NUM=PLURAL and BABYTALK=TRUE (as defined in table 2.2) a

complete morphological analysis will be available.

The concept of ambiguity enters the scene as soon as we have a more detailled look at the word *fish*. After reading the substring *fish* and doing a transition to the state $fish$, the system has no way to determine whether to take that state as a final state, leaving ROOTFORM=FISH, NUM=SINGULAR and BABYTALK=FALSE (given that the default-value of NUM is SINGULAR, and the default-value of BABYTALK is FALSE), or to do the ϵ-transitions to P_2 and q_1 leaving NUM=PLURAL. This is completely intuitive. Not even a human reader has the ability of telling whether the word-form *fish* is singular or plural, when confronted only with the string *fish*. We have to give our system the ability to let the ambiguity arise at this stage of processing, leaving the disambiguation for the syntactic or semantic analysis. This will not be handled in greater detail here, we will only give two key concepts of handling nondeterministic FSAs: The problem could be handled by parallel processing for example by forking the interpreter-process traversing the FSA, leaving one process for each possibility. One might also use backtracking for example by maintaining a stack of return-states to try after processing of one possibility is finished.

The big syntactic flaw of figure 2.8 is that it completely fails to match *kittie*, when interpreted as an FSA, since the substring 'cat' would have to be matched to get to state *cat*. This syntactic error comes from composing the procedure's I/O-diagrams into figure 2.8, but there's nothing simpler than correcting that, as figure 2.11 shows. All that's needed is a new state, let's call it *kit*, that is reached after reading 'kittie', setting both ROOT=CAT and BABYTALK=TRUE.

But still figure 2.11 isn't a very elegant way to do finite state morphological parsing. One of the main problems with figure 2.11 is the great degree of indeterminisms that arise from the ϵ-transitions that were carried over from the conceptual I/O-diagrams.

In order to get rid of these indeterminisms we need to finally get rid of our artificial procedures P_1 and P_5, transferring the program-logic into the transitions

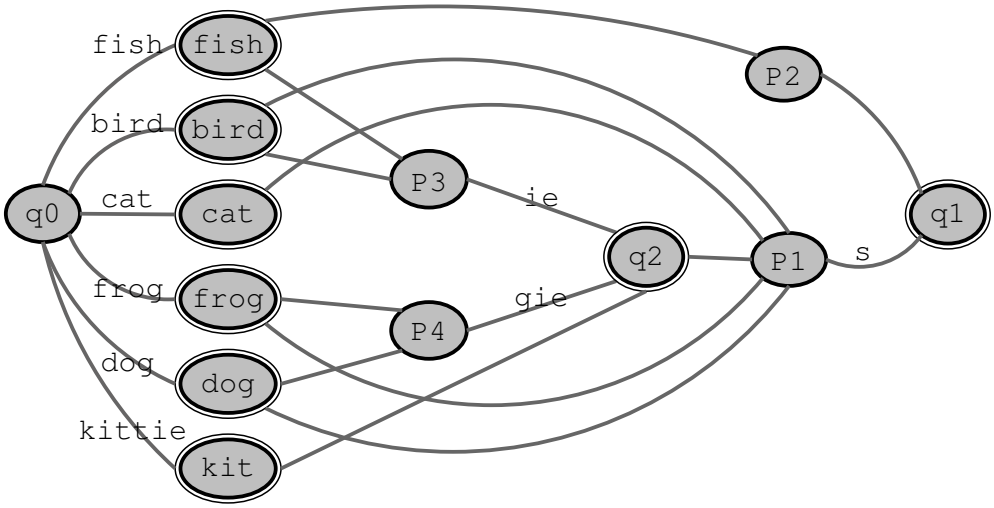


Figure 2.11: A syntactically correct FSA derived from figure 2.8

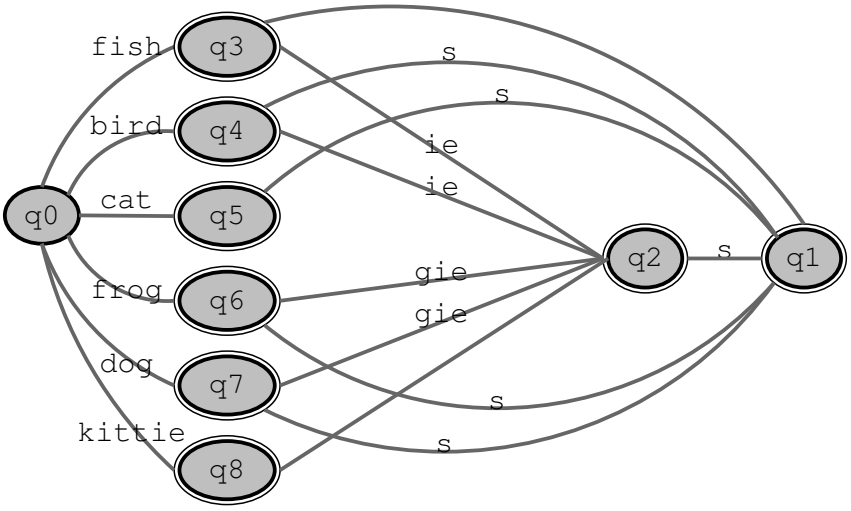


Figure 2.12: Getting rid of the indeterminisms from figure 2.11

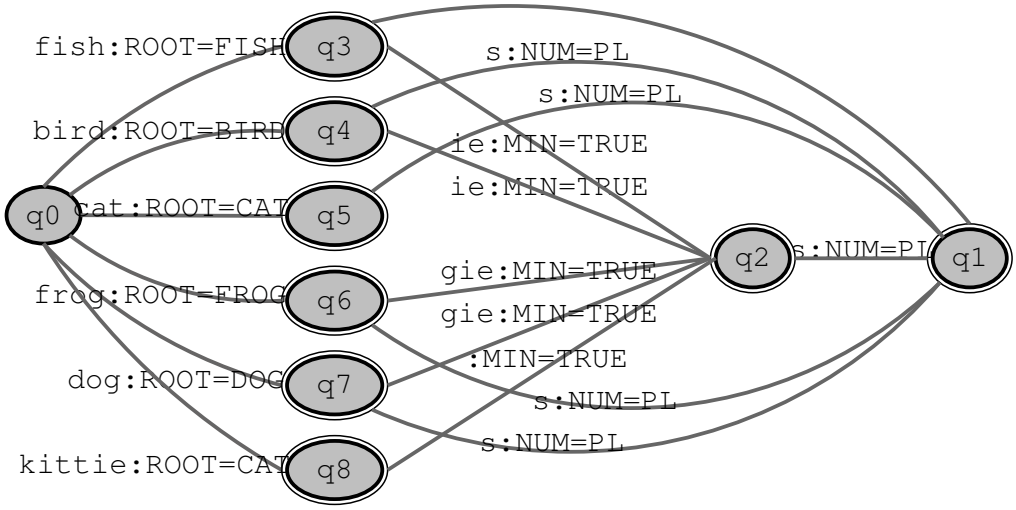


Figure 2.13: A more accurate version of figure 2.12

rather than the states. The idea is depicted in figures 2.12 and 2.13.

The FSA in figure 2.12 has only two kinds of indeterminisms involved. The first one arises from the ambiguity of the interpretation of the word-form *fish*, the other one is deciding whether to stay in a final state if one is reached or trying to do a transition. We can get rid of that kind of indeterminism by proposing a signal that terminates each word. We can then propose a new state q_F, and instead of making all of the other states final ones, we equip them with a transition to this final state q_F that is taken if, and only if, the end-of-word-signal is read.

Of course it is possible to augment an FST representing a dictionary and an FST representing the morphological system around it into a single one. The dictionary-FST would in our example replace states q_3 through q_8, and the transitions leading to them. Such an FST would match single input characters instead of substrings, and it would be possible to directly compile such an FST, to achieve optimal performance.

Chapter 3

Syntax

3.1 Overview

In order to gain a deeper understanding of syntax, it is important to understand the role of syntax in the overall process of "understanding" a meaningful sentence, which turns out to be a rather difficult task. This is also the reason why so many syntactic theories have been developed, and why some of them hardly seem to have anything in common.

Instead of going into a detailed discussion about this, we will only give an overview of the traditional approaches to "sentence-structure".

The word sentence-structure already points us in the right direction of the linguistic approach to syntax. The purpose of syntax is to put a string of symbols in relation to each other. Linguistics is the study of language, and therefore the role of syntax in linguistics is to put words in relation to each other, which is widely related to the idea known as "grammar".

Weisler & Milekic (2000, p124) define this term as follows:

> A *grammar* - a theory of linguistic knowledge - must characterize what we know about the physical signals on one end of the lingusitic equation, and what we know about meaning on the other. [...]

This already confronts us with semantics, which we want to leave for the next chapter. That's why we actually talk about the syntactic aspects of grammar only, whenever we use the term "grammar" in this chapter.

The "Context-Free Grammar", CFG for short, is the kind of grammar we will be concerned with most of the time. The CFG dates back to Chomsky (1956), independently discovered by Backus (1959).

It is based on the idea of "constituency", which states that a group of words may behave as a single unit or phrase, called a "constituent".

(3.1) The big ugly dog bit the boy.

In example 3.1 the phrase *big ugly dog* might be a constituent. Grammatically it behaves just like a single word, which is suggested by the fact that we could freely exchange it.

(3.2) The goldfish bit the boy.

In example 3.2 *big ugly dog* has been replaced by *goldfish*, and it's still grammatical. It doesn't make sense perhaps, but it is grammatical.

The idea of "replacement" is quite central to the formalism of the CFG.

What we have just observed could be expressed as a CFG as:

$$S \leftarrow \text{The } ANIMAL \text{ bit the boy}$$
$$ANIMAL \leftarrow \text{big ugly dog}$$
$$ANIMAL \leftarrow \text{goldfish}$$

A CFG is defined as $G = (V_N, V_T, P, S)$. V_N is a set of so-called non-terminal symbols. In our example we have two non-terminal symbols, namely S and $ANIMAL$. V_T is the set of terminal symbols. These are usually words, or whatever data-structure we get from the morphological analyzer. P is a set of reduction-rules like the ones given above, and S is the start-symbol, that is the symbol we ultimately want to describe the whole sentence with.

The first rule in the above example makes use of a symbolic expression, $ANIMAL$. The other rules give information about what exactly an $ANIMAL$ is. One states that the symbol could be rewritten as *big ugly dog* and the other one lets the interpreter rewrite it as *goldfish*.

Therefore the interpretation of the above grammar would generate examples 3.1 and 3.2.

3.2 A Linguistic Perspective to Sentence-Structure

Now that we know what syntax is all about, and how we can talk about syntax using the notion of grammar, especially the formalism of the CFG, we can go deeper into English sentence structure, introducing the reader to the problems and requirements English sentence-structure confronts a symbolic theory of syntax with.

3.2.1 Parts of Speech

Noun, verb, pronoun, preposition, adverb, conjunction, participle and article: This pretty much summarizes the concept behind the term "Parts of Speech", POS for short.

The above collection of parts of speech goes back to ancient Greece, yet seems surprisingly accurate. This is due to the fact that it became the basis for most subsequent POS-descriptions, which are considerably larger today. The Penn Treebank Marcus et al. (1993) enlists 45 parts of speech.

The central role of the parts of speech in each grammar is due to the significant amount of information the POS gives us about the word, and its surrounding.

It is important to understand that the parts of speech are defined through functions or classes of functions in the grammar. Traditional definitions of parts of speech tend to use a semantic approach rather than a functional/grammatical one:

"A noun is the name of a person, place or thing". Although these definitions head us in the right direction they are too imprecise for our formal theory. Fortunately there are other techniques for grammatical categorization.

A noun, like *dog*, for example, can appear after determiners like *the*; *dog* is a noun because *the dog* is grammatical; *identify* is not a noun, because **the identify* is ungrammatical. How do we know that *the* is a determiner? Because it can appear before a noun like *dog*; *the* is a determiner because *the dog* is grammatical; *easily* is not a determiner because **easily dog* is ungrammatical.

Note that the definitions are circular, yet the circularity isn't in any way problematic for the system.

Let's consider an example: We know about the following facts:

a. *the dog* is grammatical

b. *easily identify* is grammatical

c. **easily dog* is ungrammarical

d. **the identify* is ungrammatical

e. Verbs can appear after adverbs.

f. Nouns can appear after determiners.

Given these facts, our task is now to find the parts of speech of the words *the*, *easily*, *dog* and *identify*.

Let's assume that *the* was a determiner. From facts *a* and *f* we can now conclude that *dog* is a noun. From facts *d* and *f* we can conclude that *identify* is not a noun and considering *e* and *f* (words appearing second in a clause must be either a verb or a noun) we know that it is a verb. Given that *identify* is a verb, we can now conclude that *easily* is an adverb, from *b* and *e*.

Let's assume that *the* was an adverb. From facts *a* and *e* we could now conclude that *dog* is a verb. From *d* and *e* we can conclude that *identify* is not a

verb and, just as we did before, considering e and f, we know that *identify* is a noun. Given that, we can conclude that *easily* is a determiner, from b and f.

Circularity made it possible to conclude that *dog* is a verb, and *identify* is a noun, which is, according to what we learned in school, simply wrong, but why is that completely irrelevant to our computational model of syntax? Let's try to formalize what we've just observed in the CFG-formalism.

The first possiblity (the "correct" one), would be something like

$$\begin{aligned} N &\rightarrow \textit{dog} \\ V &\rightarrow \textit{identify} \\ \textit{Det} &\rightarrow \textit{the} \\ \textit{Adv} &\rightarrow \textit{easily} \\ S &\rightarrow \textit{Det } N \\ S &\rightarrow \textit{Adv } V \end{aligned}$$

The second possibility (the "incorrect" one), would then be

$$\begin{aligned} V &\rightarrow \textit{dog} \\ N &\rightarrow \textit{identify} \\ \textit{Adv} &\rightarrow \textit{the} \\ \textit{Det} &\rightarrow \textit{easily} \\ S &\rightarrow \textit{Adv } V \\ S &\rightarrow \textit{Det } N \end{aligned}$$

What is the difference between these two CFGs? We simply exchanged the names of the parts of speech. Now we call nouns verbs, and we call adverbs determiners, and vice versa, but the names of the parts of speech are as abstract

as can be. Recall that parts of speech are defined in terms of functions or classes of functions in the grammar, and these functions would stay exactly the same. The grammar would match exactly the same strings and put them exactly into the same relations regardless of what symbol we assign to what POS.

One might also use morphological criteria, for assigning words to their POS. If, for example,*determine* has a past-tense form like *determined* it must be a verb. *dog* is not a verb, since a form like **dogged* does not exist.

Note that the same circularity we just showed arises when defining POS in terms of functions in their grammar (at the interface between grammar and word), arises when defining POS in terms of morphology (at the interface between word and morpheme), since a morphological analyzer would have to know about the part of speech, to decide whether rules like that appending the affix -*ed* are appliable in the first place.

3.2.2 A Simple Grammar

Proposing Some Rules to Get Started

We have introduced the concept of a CFG, we have established some equivalence-classes that anchor our analysis, why not start and write a grammar for English.

(3.3) Steve played the guitar brilliantly.

(3.4) Old Bebe played.

(3.5) The guitarist on the left is the best.

Now let's transform these example-sentences into a syntactically correct CFG.

$S \rightarrow$ Steve played the guitar brilliantly

$S \rightarrow$ Old Bebe played

$S \rightarrow$ The guitarist on the left is the best

We have just created a CFG successfully matching all of our example sentences, producing all grammatical and no ungrammatical forms. A perfect grammar, yet completely pointless. Didn't we want to put the words into some kind of relationship?

$$S \rightarrow N\ V\ Det\ N\ Adv \tag{4}$$
$$S \rightarrow Adj\ N\ V$$
$$S \rightarrow Det\ N\ P\ Det\ N\ V\ Det\ N$$

We have built a grammar that is a lot more general, by simply introducing non-terminals for each part of speech, and replacing every word by its POS-symbol. Rule 4 from the above grammar doesn't only take care of *Steve played the guitar brilliantly*, it also matches *Matt screwed the solo completely* or *Brad ruined the guitar entirely*, suggesting some kind of syntactic and semantic isomorphy between these sentences (up to the point where one would have to "fill in" the words for the parts of speech).

The Noun-Phrase

But still we haven't made use of the notion of constituency yet, so we simply introduce the first class of constituents, which we call "noun-phrase". *Steve*, *the guitar*, *I*, *old Bebe*, *the guitarist on the left* are all examples of noun-phrases.

$$S \rightarrow NP\ V\ NP\ Adv \tag{7}$$
$$S \rightarrow NP\ V$$
$$S \rightarrow NP\ V\ NP$$
$$NP \rightarrow N \tag{10}$$

$$NP \rightarrow Det\ NP \tag{11}$$

$$NP \rightarrow Adj\ NP \tag{12}$$

$$NP \rightarrow NP\ P\ NP \tag{13}$$

This introduces the first non-terminal symbol that is not the start symbol, NP (except, of course, for our POS-symbols).

Rule 10 says that any noun can be interpreted as a noun-phrase. That rule would take care of *Steve*, or *guitar*.

Rule 11 states that any determiner followed by a noun-phrase makes up a noun-phrase. This would match for example *the guitar*, *the guitarist*, *the left*, etc.

Rule 12 states that any adjective followed by a noun-phrase makes up a noun-phrase, such as *big red guitar*, etc.

And finally Rule 13 makes it possible to view any two noun-phrases as a single noun-phrase, when they are concatenated by a preposition, such as *guitarist on the left*.

Our grammar is a lot more general now. Rule 7, matches all sentences rule 4 matched, because we got rule 7 by replacing N and $Det\ N$ by NP, and since $NP \leftarrow Det\ N$ and $NP \leftarrow N$ it must match everything it matched before, but now it allows to freely exchange noun-phrases, for example instead of *Steve played the guitar brilliantly*, *The guitarist on the left played the guitar brilliantly*.

Note that our grammar "overgenerates" the noun-phrase a bit. For example if $NP \leftarrow Adj\ NP$ and $NP \leftarrow Det\ NP$, then **big black the guitar* would be a valid noun-phrase. Therefore we introduce a new symbol called N', and redefine rules 10 to 13 that describe the NP by the following ones:

$$NP \rightarrow N'$$

$$NP \rightarrow Det\ N'$$

$$N' \rightarrow N'\ P\ NP$$
$$N' \rightarrow Adj\ N'$$
$$N' \rightarrow N$$

The Verb-Phrase

A closer look at the rules that define the sentence S in our grammar makes another redundancy obvious. It's the pattern $V\ NP$, that we'll call the Verb-Phrase, or VP.

Constituents like *played the guitar brilliantly*, *played*, *is the best* are all VPs.

$$\begin{aligned} S &\rightarrow NP\ VP \\ NP &\rightarrow N' \\ NP &\rightarrow Det\ N' \\ N' &\rightarrow N'\ P\ NP \\ N' &\rightarrow Adj\ N' \\ N' &\rightarrow N \\ VP &\rightarrow V \\ VP &\rightarrow V\ NP \\ VP &\rightarrow VP\ Adv \end{aligned} \tag{22}$$

Again our initial approach overgenerates the VP a little. Recursive application of rule 22 would allow us to stack up adverbs at the end of a VP, as in **[[[[[[played] the guitar] brilliantly] incredibly] suddenly]*, therefore, just as we did with the NP, we redefine the VP using a new non-terminal V'.

$$VP \rightarrow V'$$
$$VP \rightarrow V'\ Adv$$
$$V' \rightarrow V\ NP$$
$$V' \rightarrow V$$

3.2.3 Representations for Sentence Structure

When talking about syntactic structures it is convenient to use a representation for the structural aspects of a string of words.

If we have a constituent like *the boy* we might want to express that *the boy* is an NP consisting of a Det and an N, the Det being *the*, the N being *boy*. We might represent that like $[_{NP}\ [_{Det}$ the$][_{N}$ boy$]]$, using brackets, or using the graphical representation, referred to as "syntax tree" that we'll be introducing in the rest of this section.

A node in a syntax tree represents a nonterminal symbol, a leaf represents a terminal. A set of symbols connected to the same parent-symbol could be viewed as an alternative representation of a grammar rule. Here the parent node can be seen as the left-hand side of a CFG-rule, and the child nodes are the right-hand side symbols in a left-to-right order.

Figure 3.1 gives some examples. Figure 3.1(a) shows the (rather boring) rule $Det \rightarrow the$, and figure 3.1(b) the rule $N \rightarrow boy$. Rule $NP \rightarrow Det\ N$ can be represented as in figure 3.1(c).

To turn this graphical representation of grammar rules into a tree representing sentence structure we simply make use of recursion in the CFG-formalism, by integrating the subtrees of the used rules. We can for example integrate the rules depicted in figure 3.1 as shown in figure 3.2(a) in order to express something like

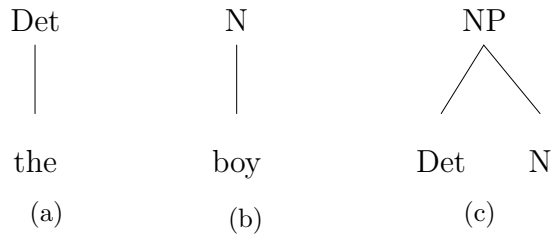


Figure 3.1: Some grammar rules in tree-notation

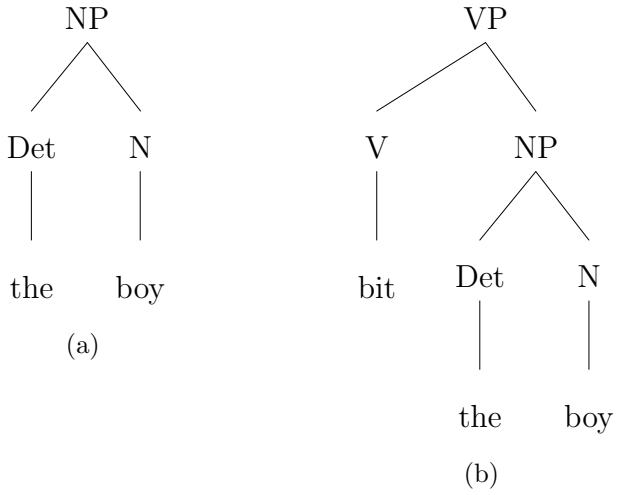


Figure 3.2: Syntax trees

"The whole thing is an NP. The Det is resolved using the $Det \rightarrow$ *the*-rule, the N is resolved using the $N \rightarrow$ *boy*-rule.".

We can use that approach to further extend our syntax tree to represent the VP *bit the boy*, as shown in figure 3.2(b).

3.2.4 Ambiguity

In the previous sections we have developed a grammar that is far from exhaustive, yet generates a basic portion of the English language.

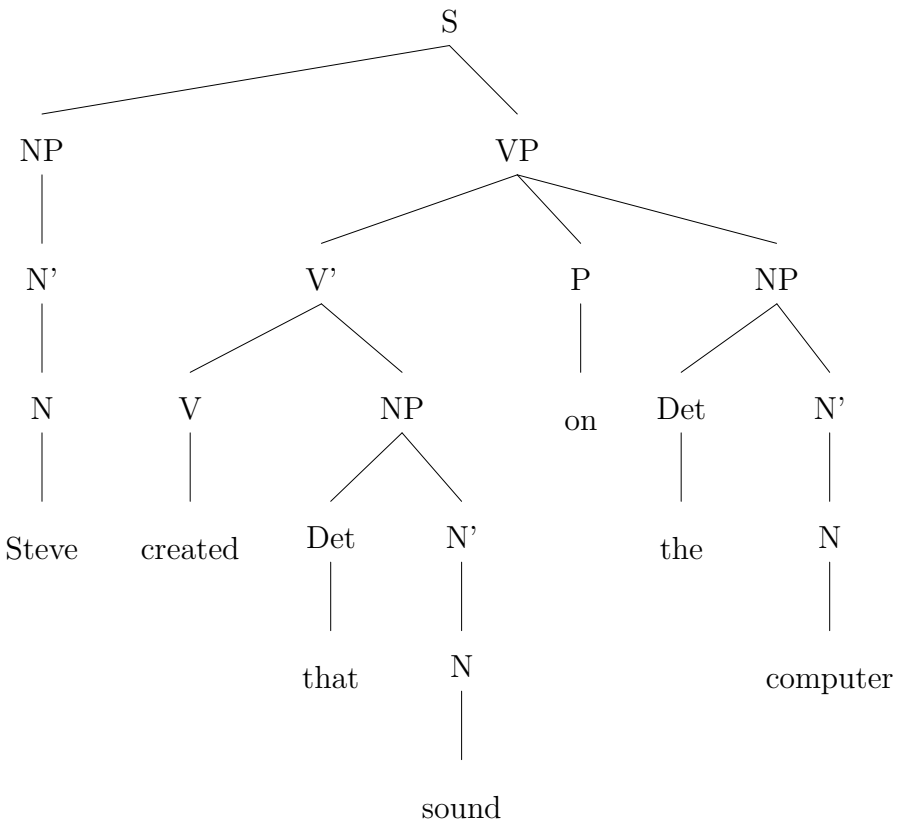


Figure 3.3: A syntax tree for example 3.6

(3.6) Steve created that sound on the computer.

Considering example 3.6, we find another grammatical phenomenon our grammar doesn't handle. Figure 3.3 shows a syntax tree as a native speaker would draw it. Here *created that sound* is one constituent and *on* is used to further specify it, in our case, to give information about the instrument he used, introducing the NP *the computer*. That constituent could be freely exchanged to give *on his guitar* or *on the piano*.

So far our grammar doesn't allow that, which is why we redefine the VP a little:

$$VP \rightarrow V'$$

$$VP \rightarrow V'\ Adv$$
$$VP \rightarrow V'\ P\ NP \tag{27}$$

Rule 27 is new. It allows the grammar to use the subtree of figure 3.3 that handles our new constituent.

After redefining our grammar in such a way our grammar is said to be "ambiguous". This is due to the fact that several parse trees for example 3.6 are derivable from it.

Figure 3.4 gives another, equally possible, parse-tree that can be applied to our example. In this case *that sound on the computer* is one constituent, so *on the computer* doesn't modify *created*, but rather *that sound*, giving information about which sound we are talking about. The phrase *the computer* could in this case be exchanged by other places, where sound can be found, as in *that sound on his new CD* or *that sound on the tape*.

3.2.5 Specifiers

Although this grammar has the ability to relate the words in accordance with their parts of speech quite well, it completely fails to handle other kinds of features such as number, person, tense, transitivity, etc. It, for example, accepts constituents like **two dog*, **a dogs*, **the dogs die yesterday*, **the dog walk into the room*, etc.

What we need is a way to constrain the application of rules based on features of the terminal-symbols that get abstracted by the non-terminals. The model needs the ability to transitively pass that information and to augment rules with constraints controlling their appliability.

In order to prevent overgeneration of, for example, the NP **a dogs*, we would have to view the form *dogs* as an instance of the class of word-forms for *dog*, distinguished from the singular form *dog* by its number, the form *dog* having the feature NUM=SINGULAR, the form *dogs* having the feature NUM=PLURAL.

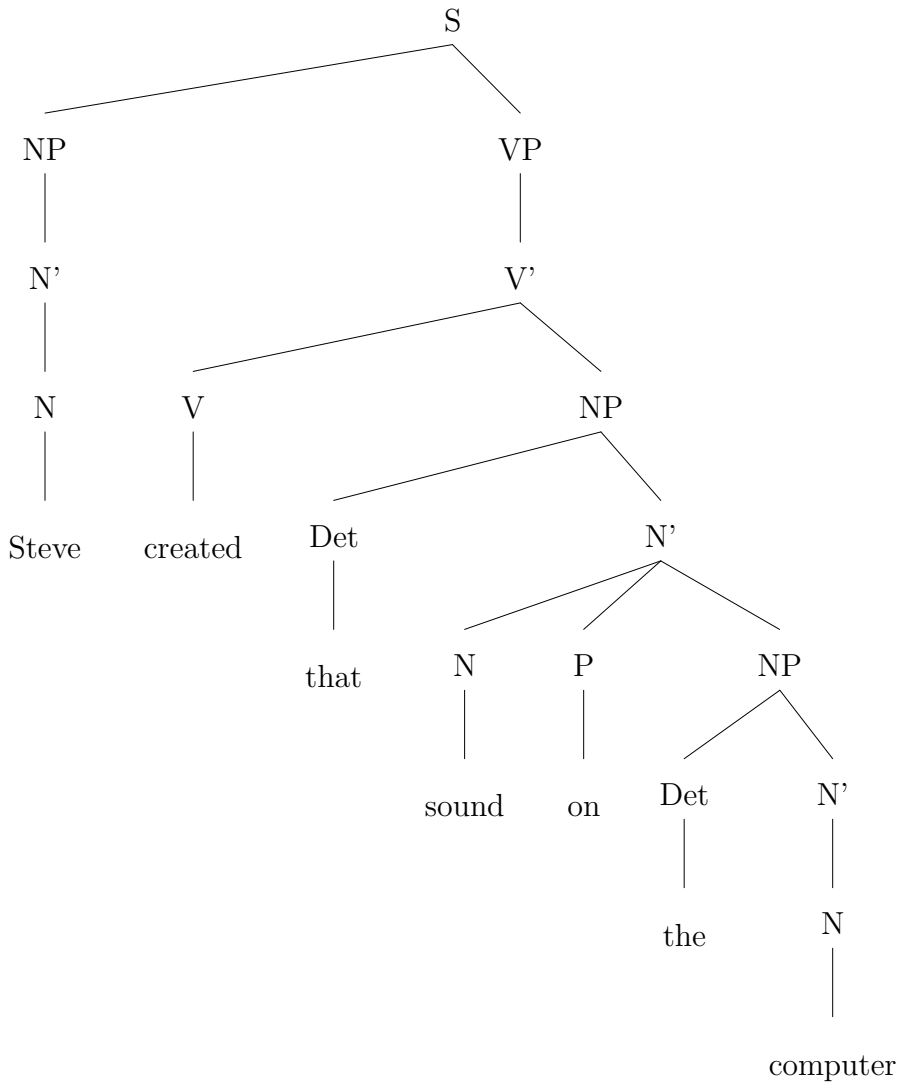


Figure 3.4: Another syntax tree for example 3.6

$$
\begin{aligned}
S &\rightarrow NP\ VP \\
NP &\rightarrow N' \\
NP &\rightarrow Det\ N' \\
N' &\rightarrow N'\ P\ NP \\
N' &\rightarrow Adj\ N' \\
N' &\rightarrow N \\
VP &\rightarrow V' \\
VP &\rightarrow V'\ Adv \\
VP &\rightarrow V'\ P\ NP \\
V' &\rightarrow V\ NP \\
V' &\rightarrow V
\end{aligned}
$$

Figure 3.5: Our complete sample-grammar

Then we could augment our grammar with constraints doing simple operations like equality checks. We would, therefore, rewrite a rule like $NP \rightarrow Det\ N$ to something like

$$NP \rightarrow Det\ N$$

Apply this rule only if the Det's NUM*-feature is equal to the N's* NUM*-feature*

the resulting NP would have the same value for NUM *as the Det and the N*

Such a rule would only match noun-phrases if the Det and the N agree in number, and it would produce an NP that also has a NUM-feature that can be used in further processing. For example, a rule like

$$S \rightarrow NP\ VP$$

Apply this rule only if the NP's NUM*-feature is equal to the VP's* NUM*-feature*

would only match *the dog enters the room*, and *the dogs enter the room*, and not **the dogs enters the room* or **the dog enter the room*, given that the NP *the dog* has the feature NUM=SINGULAR, and the VP *enters the room* has the feature NUM=SINGULAR, while *the dogs* has the feature NUM=PLURAL and *enter the room* NUM=PLURAL, which is exactly what we wanted to achieve.

3.3 Parsing

Thanks to the CFG, it is possible for linguists to provide computer scientists with a detailed description of language and its underlying syntactic structures, in a form adequate for symbolic computation. It serves as a basis for computer-understanding of natural language structure, but makes no statements on what to do with it. It still needs to be interpreted. A computer-program could apply the knowledge it gained from interpreting the CFG to a given input-sentence, and come up with a representation accounting for sentence structure. In this section we will be concerned with this kind of interpretation of a CFG.

Figure 3.6: A tree showing the derivation of *undrinkable*

It is, of course, possible to directly view a CFG-description of a language as a rule-based program. All one would have to do is rewrite the CFG-rules to a form that is syntactically acceptable for a logic programming language such as PROLOG, and provide some underlying logic to the idea of the CFG to get a program capable of interpreting a CFG. A system like PROLOG would usually use search-trees and unification-based algorithms to handle this, but there are strategies specific to CFGs that are a lot more efficient. We will discuss some of these parsing strategies and their underlying ideas in this section.

3.3.1 Excursion: Backtracking through State-spaces

One of the most basic and most generally appliable algorithms in artificial intelligence is that of backtracking. Backtracking is a search-strategy based on the formalism of the state-space, and it is so generally appliable, because most of the problems that arise in artificial intelligence can easily be formalized in the state-space-paradigm.

A state-space is formalized as (S, F, G) representing a set of possible starting-states S, a set of operations appliable to the states F and a set of desired states, or goals G.

We can make things clearer by considering the example of the *three coins problem* Jackson (1985) uses to illustrate the state-space formalism. Figure 3.6 depicts the start-state: three coins, arranged in such a way, that the first one shows the head, the second one shows the head and the third one shows the tail.

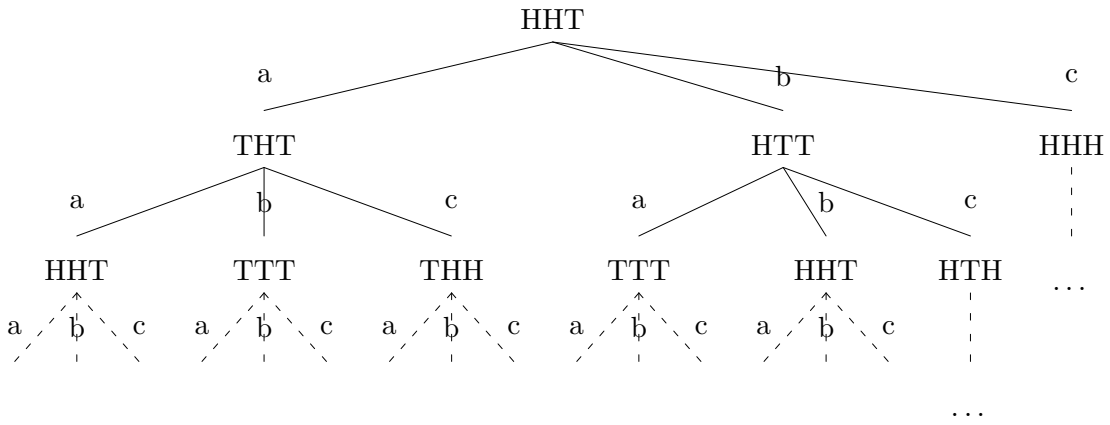


Figure 3.7: A search-tree through the state-space of the *three coins problem*

We could simplify things by using a special representation for these configurations of coins. The start-state could, for example, be represented as HHT (indicating head-head-tail).

There are three possible actions we can apply to this state:

a. Flip the first coin

b. Flip the second coin

c. Flip the third coin

The problem is this: What do we have to do to make all the coins show the same side? This gives us the desired state, the goal, which is either TTT, or HHH.

Figure 3.7 depicts the state-space graphically. It shows some of the possible states, for example, the start-state in the top node. Each of the arcs connecting this node with its child-nodes represents one action. Each action leads to a new state, a child-node, which can itself be the base for subsequent applications of actions. Note that state-spaces don't neccessarily have to take the form of a tree, but we will consider only this class of state-spaces.

Therefore figure 3.7 is also a graphical representation of the search-tree itself.

There can be three procedures involved in finding a solution. One takes care of generating the state-space, one checks the state-space for a solution. A procedure doing both is called a "search procedure". Usually a computer has to be somewhat selective in generating a state-space, because state-spaces can get very large. This is where a third procedure comes in. It evaluates actions, giving information about their likelihood of containing a solution. A search-procedure making use of such an evaluator, to selectively generate only the most promising parts of a state-space is said to be a "heuristic search".

This is mentioned only for the sake of completeness. We won't make use of any heuristics here, but simply look at the most prominent search-procedure, which is known as backtracking.

Backtracking is simply the depth-first-traversion of a search-tree. Putting it strongly simplified, we can think of backtracking as doing the following:

- Generate the state S_1 the first action (a_1) leads to.
- Is it impossible to create S_1? Then we know that a_1 doesn't lead to a solution.
- Is the S_1 the desired state? Then we know that a_1 leads to a solution.
- Is S_1 not the desired state? Use this algorithm recursively to find out whether S_1 is the top of a subtree that does contain the solution, which would imply that a_1 leads to a solution.
- Does a_1 lead to a solution? Then we can terminate.
- Otherwise: Generate the state S_2 the second action (a_2) leads to.
- Is it impossible to create S_2? ...

Keep in mind that backtracking is one of the simplest, but most prominent, and most widely used search-strategies, but a great variety of other search-

strategies have evolved from artificial-intelligence-research. Some are, for example, based on a breadth-first-traversion of a search-tree. It is possible to traverse the tree from the top down or from the bottom up, which is widely known as goal-directed respectively data-directed search. It is even possible to traverse the tree in both directions at the same time. Traversal can be done in parallel, for example by forking a process, or pseudo-parallely, by maintaining a stack, representing a TODO-List containing "states still to be examined", and so on.

Note that the three-coins-problem would turn out to be rather problematic, when solved using backtracking, since it contains "left recursion". It would start by applying a to HHT, generating THT. Since THT is not the desired state it would recursively apply itself to THT, which means that it, again, starts by applying a to this newly created state, leading to HHT. It would then recursively apply itself to HHT, again applying action a, and would therefore never terminate.

3.3.2 Basic Parsing Strategies

We have already pointed out the parallelity between the runtime-structure of a system of logical inference and a parser. Just like a rule-based program, a grammar is a system containing rules and facts (terminal-symbols), and the interpreter is supposed to process a query (start symbol), by systematically applying rules to each other, until a rule matching the query is deduced from the system. (Which is, again, closely related to state-spaces. Just think of the application of a rule as being an action, and the resulting rule with terminals/facts filled in as a state.)

Thinking of parsing, as solving the problem of assigning the correct parse-tree to a given input, there are two ways of formalizing this problem. One could either think of parsing as the problem of expanding a given top-node (usually S) in such a way that it matches the input (then the inital state would be the top-node S, and the goal would be a state, where the terminals match the input), or one could think of parsing as the problem of relating a given input in such a way that

it can be shown to be an S (then the initial state would be the input, and the goal would be any state with the top-node S). These two ways of formalizing the problem of parsing give rise to the basic parsing-strategies, known as "top-down" and "bottom-up".

A "top-down"-parser would work its way from the top of the syntax-tree, down, until it discovers a terminal-symbol, either backtracking if the symbol doesn't match, or continuing if it does match the given input-symbol.

The "bottom-up"-strategy would, in contrast, start by examining the input-symbol, and work its way up the syntax-tree, until it finds the start-symbol.

While a top-down parser doesn't waste time in examining structures that will never lead to the start-symbol, the bottom-up-parser doesn't look at structures that will never match the input, which is why the size of the grammar and the length of the input are important considerations in deciding which strategy is best.

3.3.3 Parsing by Problem-Solving

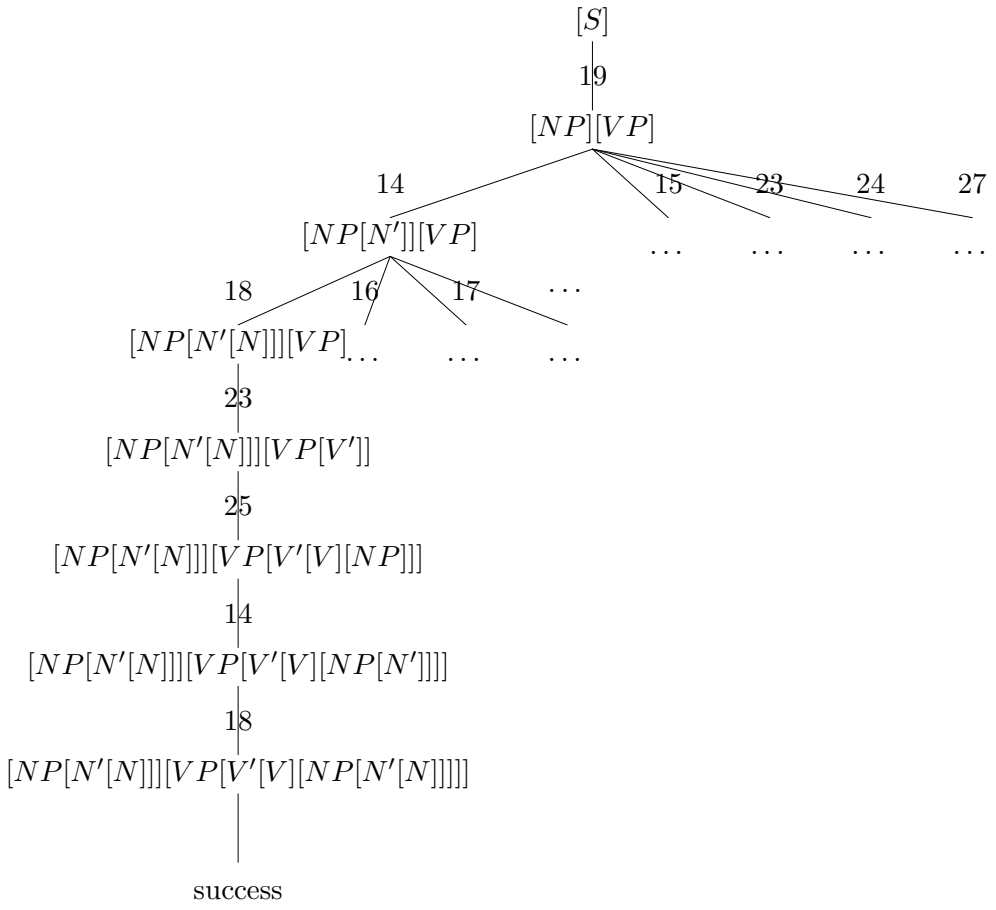
(3.7) Steve plays chess.

A problem-solver can directly be used as a parser. What we have is a state space (S, F, G). States are partial parse-trees. We could represent the initial state by something like $[S]$, when we do a top-down parse. Each of the rules in the grammar is then an action.

We will consider example 3.7, using the grammar given in figure 3.5.

Figure 3.8 shows a state-space (which is, again, also a search-tree). The first nodes are depicted in full detail, the others are left out, due to limitations in space, but the principle should get clear anyway.

Each of the rules in the sample-grammar is an action. There is one action, appliable to the state $[S]$, which is the rule $S \rightarrow NP\ VP$, because it is the only one expanding the symbol we are looking for. By applying this action, we deduce



$$
\begin{aligned}
S &\rightarrow NP\ VP \\
NP &\rightarrow N' \\
NP &\rightarrow Det\ N' \\
N' &\rightarrow N'\ P\ NP \\
N' &\rightarrow Adj\ N' \\
N' &\rightarrow N \\
VP &\rightarrow V' \\
VP &\rightarrow V'\ Adv \\
VP &\rightarrow V'\ P\ NP \\
V' &\rightarrow V\ NP \\
V' &\rightarrow V
\end{aligned}
$$

Figure 3.8: A search-tree through the state-space of a parsing-problem

the new state $[NP][VP]$. Now we can apply all the actions expanding either NP or VP, namely rules 14, 15, 23, 24 and 27. In the figure only the path to the goal is shown, but our parser usually has no way of telling which one is "right". It might as well try, for example, rule 15 first. Applying rule 14, which is $NP \rightarrow N'$ to the state $[NP][VP]$ creates the state $[NP[N']][VP]$. This time all rules expanding either N' or VP are appliable, and so fort. We can go on like this till we discover a state consisting only of terminals. If these terminals match the input, the state is a goal-state, if not, we use backtracking to try out another path through the state-space that might lead to the goal-state.

The backtracking-approach, in general, suffers from vast ineffectivity in parsing because it runs in exponential-time, spending most of it examining fruitless subtrees and reexamining the same fruitless subtrees over and over. Although it is possible to improve these algorithms somewhat by adding "bottom-up-filtering" to a top-down parser respectively "top-down-filtering" to a "bottom-up-parser" or heuristic methods, the major disadvantage of ineffectivity, when using exponential-time problem-solvers like backtracking, remains the same.

3.3.4 The Earley Algorithm

A great way to improve efficiency of exponential-time algorithms comes from a framework called "dynamic programming", and is sometimes referred to as "memoization".

If we apply this technique to the basic idea of a top-down parser with bottom-up filtering, we get an algorithm similar to that of Earley (1970), which we'll describe in greater detail in this section.

The Earley-Algorithm avoids reexecuting computations done in the course of parsing an input, by "remembering" subproblems and their solutions in the so-called "chart". That's why approaches similar to Earley's are often referred to as "chart-parsers".

chart[0]

[0]	$\lambda \rightarrow \bullet S$	[0, 0, 0]
[1]	$S \rightarrow \bullet NP\ VP$	[0, 0, 0]
[2]	$NP \rightarrow \bullet N'$	[0, 0, 0]
[3]	$NP \rightarrow \bullet Det\ N'$	[0, 0, 0]
[4]	$N' \rightarrow \bullet N'\ P\ NP$	[0, 0, 0]
[5]	$N' \rightarrow \bullet Adj\ N'$	[0, 0, 0]
[6]	$N' \rightarrow \bullet N$	[0, 0, 0]

chart[1]

[0]	$N \rightarrow steve\bullet$	[0, 1, 1]
[1]	$N' \rightarrow N\bullet$	[0, 1, 1]
[2]	$NP \rightarrow N'\bullet$	[0, 1, 1]
[3]	$N' \rightarrow N' \bullet P\ NP$	[0, 1, 1]
[4]	$S \rightarrow NP \bullet VP$	[0, 1, 1]
[5]	$VP \rightarrow \bullet V'$	[1, 1, 0]
[6]	$VP \rightarrow \bullet V'\ Adv$	[1, 1, 0]
[7]	$VP \rightarrow \bullet V'\ P\ NP$	[1, 1, 0]
[8]	$V' \rightarrow \bullet V$	[1, 1, 0]
[9]	$V' \rightarrow \bullet V\ NP$	[1, 1, 0]

chart[2]

[0]	$V \rightarrow created\bullet$	[1, 2, 1]
[1]	$V' \rightarrow V\bullet$	[1, 2, 1]
[2]	$V' \rightarrow V \bullet NP$	[1, 2, 1]
[3]	$VP \rightarrow V'\bullet$	[1, 2, 1]
[4]	$VP \rightarrow V' \bullet Adv$	[1, 2, 1]
[5]	$VP \rightarrow V' \bullet P\ NP$	[1, 2, 1]
[6]	$NP \rightarrow \bullet N'$	[2, 2, 0]
[7]	$NP \rightarrow \bullet Det\ N'$	[2, 2, 0]
[8]	$S \rightarrow NP\ VP\bullet$	[0, 2, 2]
[9]	$N' \rightarrow \bullet N'\ P\ NP$	[2, 2, 0]
[10]	$N' \rightarrow \bullet Adj\ N'$	[2, 2, 0]
[11]	$N' \rightarrow \bullet N$	[2, 2, 0]
[12]	$\lambda \rightarrow S\bullet$	[0, 2, 1]

chart[3]

[0]	$Det \rightarrow that\bullet$	[2, 3, 1]
[1]	$NP \rightarrow Det\ N'\bullet$	[2, 3, 1]
[2]	$N' \rightarrow \bullet N'\ P\ NP$	[3, 3, 0]
[3]	$N' \rightarrow \bullet Adj\ N'$	[3, 3, 0]
[4]	$N' \rightarrow \bullet N$	[3, 3, 0]

chart[4]

[0]	$N \rightarrow sound\bullet$	[3, 4, 1]
[1]	$N' \rightarrow N\bullet$	[3, 4, 1]
[2]	$NP \rightarrow Det\ N'\bullet$	[2, 4, 2]
[3]	$N' \rightarrow N' \bullet P\ NP$	[3, 4, 1]
[4]	$V' \rightarrow V\ NP\bullet$	[1, 4, 2]
[5]	$VP \rightarrow V'\bullet$	[1, 4, 1]
[6]	$VP \rightarrow V' \bullet Adv$	[1, 4, 1]
[7]	$VP \rightarrow V' \bullet P\ NP$	[1, 4, 1]
[8]	$S \rightarrow NP\ VP\bullet$	[0, 4, 2]
[9]	$\lambda \rightarrow S\bullet$	[0, 4, 1]

chart[5]

[0]	$P \rightarrow on\bullet$	[4, 5, 1]
[1]	$N' \rightarrow N'\ P \bullet NP$	[3, 5, 2]
[2]	$VP \rightarrow V'\ P \bullet NP$	[1, 5, 2]
[3]	$NP \rightarrow \bullet N'$	[5, 5, 0]
[4]	$NP \rightarrow \bullet Det\ N'$	[5, 5, 0]
[5]	$N' \rightarrow \bullet N'\ P\ NP$	[5, 5, 0]
[6]	$N' \rightarrow \bullet Adj\ N'$	[5, 5, 0]
[7]	$N' \rightarrow \bullet N$	[5, 5, 0]

chart[6]

[0]	$Det \rightarrow the\bullet$	[5, 6, 1]
[1]	$NP \rightarrow Det \bullet N'$	[5, 6, 1]
[2]	$N' \rightarrow \bullet N'\ P\ NP$	[6, 6, 0]
[3]	$N' \rightarrow \bullet Adj\ N'$	[6, 6, 0]
[4]	$N' \rightarrow \bullet N$	[6, 6, 0]

chart[7]

[0]	$N \rightarrow computer\bullet$	[6, 7, 1]
[1]	$N' \rightarrow N\bullet$	[6, 7, 1]
[2]	$NP \rightarrow Det\ N'\bullet$	[5, 7, 2]
[3]	$N' \rightarrow N' \bullet P\ NP$	[6, 7, 1]
[4]	$N' \rightarrow N'\ P\ NP\bullet$	[3, 7, 3]
[5]	$VP \rightarrow V'\ P\ NP\bullet$	[1, 7, 3]
[6]	$NP \rightarrow Det\ N'\bullet$	[2, 7, 2]
[7]	$N' \rightarrow N' \bullet P\ NP$	[3, 7, 1]
[8]	$S \rightarrow NP\ VP\bullet$	[0, 7, 2]
[9]	$V' \rightarrow V\ NP\bullet$	[1, 7, 2]
[10]	$\lambda \rightarrow S\bullet$	[0, 7, 1]
[11]	$VP \rightarrow V'\bullet$	[1, 7, 1]
[12]	$VP \rightarrow V' \bullet Adv$	[1, 7, 1]
[13]	$VP \rightarrow V' \bullet P\ NP$	[1, 7, 1]

Figure 3.9: A chart for a run of our Earley parser against example 3.6

The chart can be seen as the algorithm's "agenda". While executing it, it appends new items to this "agenda", therefore dynamically manipulating its runtime-structure. When finished, the chart contains all subproblems and their solutions, and the solution to the whole problem. Figure 3.9 shows such a chart.

This data-structure contains a set of states. A state represents a rule applied for a particular set of symbols at a specific point in the progress of proving it to be appliable to these input symbols. This can be achieved by simply inserting a symbolic "•" at some position in the rule, indicating that everything to the left of the • has already been read, and everything to the right still has to be read, and remembering two positions in the input, one giving information about where the state begins, and one giving information about where the • from this rule, can be found.

Let's have a look at some examples. Figure 3.9 was created by an Earley-Parser using our sample-grammar, and parsing example 3.6, that is repeated here as example 3.8, with a numbered set of bullets added.

(3.8) $\bullet_0$ Steve $\bullet_1$ created $\bullet_2$ that $\bullet_3$ sound $\bullet_4$ on $\bullet_5$ the $\bullet_6$ computer $\bullet_7$

Let's have a closer look at the state from figure 3.9, with the number [1] in *chart*[5], repeated here.

$$N' \rightarrow N'\ P \bullet NP[3,5]$$

In this state the parser tries to prove the rule $N' \rightarrow N'\ P\ NP$ to be appliable to the input, beginning at $\bullet_3$. The • can be found in the input-stream as $\bullet_5$, and given that it's right of $N'\ P$, and left of NP, it means that an N' and the P have already been recognized, and the NP still has to be read.

For each state in the chart, the Earley-Algorithm applies one of three operations known as PREDICTOR, SCANNER and COMPLETER, each of which add new states to the current or next chart. A state that is already in the chart is never

added a second time, even if the operation would normally do so. This is how data- and runtime-structure effectively avoid redundancy.

The Predictor is applied to each rule that still has non-terminals immediately to the right of the •, that is, to each state that still has to prove a new non-terminal to appear next. In order to do so the PREDICTOR adds new states to the chart, "expanding" the non-terminal-symbol in question by the rules replacing this particular symbol, therefore running in a "top-down"-manner.

Let's again consider an example. The initial state of the parser is the "artificial" state

$$\lambda \rightarrow \bullet S[0,0]$$

the chart is initialized with. When the algorithm comes across this state it finds the non-terminal S to the right of the dot. There is one rule expanding S, namely $S \rightarrow NP\ VP$, therefore the PREDICTOR would add the new state $S \rightarrow \bullet NP\ VP[0,0]$ to the chart. Next the PREDICTOR comes across this newly created state, finding the non-terminal NP to the right of the •. There are two rules that expand the NP, namely $NP \rightarrow N'$ and $NP \rightarrow Det\ N'$, therefore the PREDICTOR creates two new states, $NP \rightarrow \bullet N'[0,0]$ and $NP \rightarrow \bullet Det\ N'[0,0]$, and so on.

The Scanner is, as the name suggests, used to advance a state by scanning the input. Let's look at the operation of the SCANNER, by considering the state

$$N' \rightarrow N' \bullet P\ NP[3,4]$$

that can be found in the figure 3.9 in *chart*[4] at position [3]. In this state the • is to the left of the P, therefore the parser has to prove a symbol of category P to appear next in the input. Now it's the SCANNER's job to look at the input, and verify whether the next input-symbol is a P or not. In the input, the signal to the

right of $\bullet_4$ is "on", so the SCANNER adds the new rule $P \rightarrow on \bullet [4,5]$ to the next chart. When the parser proceeds to the next word, the COMPLETER takes care of proceeding the $\bullet$ in state $N' \rightarrow N' \bullet P\ NP[3,4]$ to give $N' \rightarrow N'\ P \bullet NP[3,5]$.

The Completer is called for a state which is "complete". A state is considered complete, when the $\bullet$ is at the far right of the rule. In the last paragraph we added the state $P \rightarrow on \bullet [4,5]$. Therefore, any state "looking" for a P in the input at $\bullet_4$, can be advanced, which means the $\bullet$ can be moved from the left of a P to the right.

The COMPLETER, coming across $P \rightarrow on \bullet [4,5]$ in $chart[5]$ would, therefore, look through all the states in $chart[4]$. When it finds, for example, $N' \rightarrow N' \bullet P\ NP[3,4]$ in $chart[4]$, it advances the $\bullet$, and adds the new state $N' \rightarrow N'\ P \bullet NP[3,5]$ to $chart[5]$.

An Example-Run

Because a fundamental understanding of the Earley-algorithm is vital to this project, we will not introduce any new concepts in this section, but dedicate it to revise what we've heard about the Earley-algorithm so far, by following an Earley-parser all the way through a parse.

(3.9) Steve plays chess.

The chart in figure 3.10 represents a run of an Earley-parser, parsing example 3.9 with a simplified version of our grammar.

The first state ($\lambda \rightarrow \bullet S[0,0]$) is added as an initialization-value for the chart, and is handled just like any other state. Since it has a non-terminal to the right of the $\bullet$, the PREDICTOR is called to handle that state. The PREDICTOR finds the symbol S to the right of the $\bullet$, and looks up all the rules in the grammar that have the form $S \rightarrow \ldots$ and adds them to the current chart (which is $chart[0]$). In this case, there is only one rule of that form in the grammar, which is $S \rightarrow NP\ VP$.

chart[0]

[0]	$\lambda \rightarrow \bullet S$	[0, 0, 0]
[1]	$S \rightarrow \bullet NP\ VP$	[0, 0, 0]
[2]	$NP \rightarrow \bullet Det\ N$	[0, 0, 0]
[3]	$NP \rightarrow \bullet N$	[0, 0, 0]

chart[1]

[0]	$N \rightarrow steve\bullet$	[0, 1, 1]
[1]	$NP \rightarrow N\bullet$	[0, 1, 1]
[2]	$S \rightarrow NP \bullet VP$	[0, 1, 1]
[3]	$VP \rightarrow \bullet VNP$	[1, 1, 0]

chart[2]

[0]	$V \rightarrow plays\bullet$	[1, 2, 1]
[1]	$VP \rightarrow V \bullet NP$	[1, 2, 1]
[2]	$NP \rightarrow \bullet Det\ N$	[2, 2, 0]
[3]	$NP \rightarrow \bullet N$	[2, 2, 0]

chart[3]

[0]	$N \rightarrow chess\bullet$	[2, 3, 1]
[1]	$NP \rightarrow N\bullet$	[2, 3, 1]
[2]	$VP \rightarrow V\ NP\bullet$	[1, 3, 2]
[3]	$S \rightarrow NP\ VP\bullet$	[0, 3, 2]
[4]	$\lambda \rightarrow S\bullet$	[0, 3, 1]

Figure 3.10: A chart for a run of our Earley parser on example 3.9

Therefore the state $S \rightarrow \bullet NP\ VP[0,0]$ is added to the chart. Whenever a grammar-rule is newly added to the chart by the PREDICTOR its $\bullet$ is on the far left (since we haven't read any symbols for this rule so far), which is why its rule-position must match its global position. The global position itself is always carried over from the original state the PREDICTOR was called for (which is $\lambda \rightarrow \bullet S[0,0]$ in our case), to ensure that when "working off" the new state, the parser looks for the symbols at the same position in the input where the original state would have expected them.

Working off the agenda, our parser now finds the state $S \rightarrow \bullet NP\ VP[0,0]$ we just added and interprets it in exactly the same way by calling the PREDICTOR, since the symbol to the right of the $\bullet$ is NP, which is a non-terminal. The PREDICTOR looks up all the rules in the grammar that have the form $NP \rightarrow \ldots$. In our grammar there are two such rules: $NP \rightarrow Det\ N$ and $NP \rightarrow N$, which is why the two states $NP \rightarrow \bullet Det\ N[0,0]$ and $NP \rightarrow \bullet N[0,0]$ are added.

Now the parser finds the state $NP \rightarrow \bullet Det\ N[0,0]$ that was just added. This time the symbol to the right of the bullet is Det, which is a terminal. That's why we call the SCANNER now, instead of the PREDICTOR. Instead of looking up a rule in the grammar, the SCANNER looks up the word in the input. In our case it would look for a Det appearing in the input to the right of $\bullet_0$ (or, to put it simply, the first word). The first word is *Steve*, and *Steve* can under no circumstances be a Det. That's why the SCANNER doesn't add any states.

The next state the parser considers is $NP \rightarrow \bullet N[0,0]$. Again, the SCANNER is called, but this time the input (*Steve*) does match the terminal to the right of the $\bullet$ (N), which is why the state $N \rightarrow steve \bullet [0,1]$ gets added to the *next* chart (which is $chart[1]$). The rule-position is still 0, because we, or merely the rule in the state, are still talking about the first word in the input. The rule $N \rightarrow steve$ has been proven to be appliable for the first word in the input. The $\bullet$ is now to the right of *steve*, since we have already read *steve*. Of course, this increases the global position, since the $\bullet$ from the state $N \rightarrow steve \bullet [0,1]$ now corresponds to

$\bullet_1$ from the input, and not to $\bullet_0$.

Since there is no further rule in the current chart (*chart*[0]), the parser can now move on to the next chart (*chart*[1]). The first state it finds there is the state $N \rightarrow steve \bullet [0, 1]$, that was just added by the SCANNER. This state is said to be *complete*, since there is no further symbol to the right of the $\bullet$, which is why the COMPLETER is called to handle this state. The COMPLETER now does the job of looking through the old states to see if any of the states were looking for the symbol we just found in the input, so it looks through the states in *chart*[0]. The symbol we just found in the input is the lefthand-side of the state we were called for ($N \rightarrow steve \bullet [0, 1]$) which is N, so it is interested in all the states of the form $? \rightarrow \ldots \bullet N \ldots$ in that chart the rule-position of the complete state points to. (In our case the rule-position is 0, so it looks in *chart*[0].) There is only one such state in *chart*[0], which is $NP \rightarrow \bullet N[0, 0]$. This state is now "advanced", which is simply the process of moving the $\bullet$ from the lefthand-side of the symbol over to the righthand-side, therefore our COMPLETER would use the state $NP \rightarrow \bullet N[0, 0]$ from *chart*[0], and advance it by adding the state $NP \rightarrow N \bullet [0, 1]$ to *chart*[1] according to the state it was called for ($N \rightarrow steve \bullet [0, 1]$).

Again, the parser can move on in the agenda. This time it finds the state $NP \rightarrow N \bullet [0, 1]$ we just added. This is, again, a complete state (there is no further symbol to the right of the $\bullet$), which is why the COMPLETER is called to handle this state, which goes through the same procedure of searching *chart*[0] for states looking for an NP to be advanced. There is one such state, which is $S \rightarrow \bullet NP\ VP[0, 0]$, so the COMPLETER adds the state $S \rightarrow NP \bullet VP[0, 1]$.

Now our parser is confronted with the state $S \rightarrow NP \bullet VP[0, 1]$, which has a non-terminal (VP) to the right of the $\bullet$, so the PREDICTOR is called, to add states from the grammar, that have the form $VP \rightarrow \ldots$. In the grammar there is one such rule, namely $VP \rightarrow V\ NP$, so the rule $VP \rightarrow \bullet V\ NP[1, 1]$ is added (recall that the bullet for predicted states is always at the far left, and the rule-position as well as the global position are initialized to the original state's global

position).

When handling state $VP \rightarrow \bullet V\ NP[1,1]$, the parser calls the SCANNER to scan for a V, appearing in the input at $\bullet_1$. The SCANNER is, in this case, successful in doing so, since the second word is *plays*. Therefore it adds a new complete state, regarding that V, so the COMPLETER takes care of advancing $VP \rightarrow \bullet V\ NP[1,1]$ to give $VP \rightarrow V \bullet NP[1,2]$.

This state is then handled by the PREDICTOR which adds the states $NP \rightarrow \bullet Det\ N[2,2]$ and $NP \rightarrow \bullet N[2,2]$, both of which are handled by the SCANNER, which fails for the first one, and succeeds for the second one, because the word it finds at $\bullet_2$, *chess* is a N, and can't be a Det. That's how state $N \rightarrow chess \bullet [2,3]$ enters $chart[3]$. It's used by the COMPLETER to advance state $NP \rightarrow \bullet N[2,2]$, which is why it adds state $NP \rightarrow N \bullet [2,3]$. This state itself is complete and therefore, once again, the COMPLETER is called, this time to advance state $VP \rightarrow V \bullet NP[1,2]$ to give $VP \rightarrow V\ NP \bullet [1,3]$, again a complete state, used by the COMPLETER to advance state $S \rightarrow NP \bullet VP[0,1]$ to give $S \rightarrow NP\ VP \bullet [0,3]$.

This can then be used to complete the initial state $\lambda \rightarrow \bullet S[0,0]$ to give $\lambda \rightarrow S \bullet [0,3]$, which is the end of our odyssee through the chart shown in figure 3.10.

Building the Parse-Forest

Actually, the algorithm we just showed is not a parser, but rather, a recognizer. It is a way of telling whether a given input is grammatical, but there's no way to retrieve a parse-tree from the chart, shown in figure 3.9. When the algorithm terminates, it can search the last chart for a state like $\lambda \rightarrow S\bullet$. If there is no such state, the input does not match, if there is one, it does match the input.

In figure 3.11 we showed how a chart of an Earley-parser could look, in contrast to the chart of an Earley-recognizer. We have added a column containing the pointers that make up the so-called parse-forest. Recalling that natural-language

chart[0]

[0]	$\lambda \rightarrow \bullet S$	[0, 0, 0]	[]
[1]	$S \rightarrow \bullet NP\ VP$	[0, 0, 0]	[]
[2]	$NP \rightarrow \bullet N'$	[0, 0, 0]	[]
[3]	$NP \rightarrow \bullet Det\ N'$	[0, 0, 0]	[]
[4]	$N' \rightarrow \bullet N'\ P\ NP$	[0, 0, 0]	[]
[5]	$N' \rightarrow \bullet Adj\ N'$	[0, 0, 0]	[]
[6]	$N' \rightarrow \bullet N$	[0, 0, 0]	[]

chart[1]

[0]	$N \rightarrow steve\bullet$	[0, 1, 1]	[]
[1]	$N' \rightarrow N\bullet$	[0, 1, 1]	[[(1, 0)]]
[2]	$NP \rightarrow N'\bullet$	[0, 1, 1]	[[(1, 1)]]
[3]	$N' \rightarrow N' \bullet P\ NP$	[0, 1, 1]	[[(1, 1)]]
[4]	$S \rightarrow NP \bullet VP$	[0, 1, 1]	[[(1, 2)]]
[5]	$VP \rightarrow \bullet V'$	[1, 1, 0]	[]
[6]	$VP \rightarrow \bullet V'\ Adv$	[1, 1, 0]	[]
[7]	$VP \rightarrow \bullet V'\ P\ NP$	[1, 1, 0]	[]
[8]	$V' \rightarrow \bullet V$	[1, 1, 0]	[]
[9]	$V' \rightarrow \bullet V\ NP$	[1, 1, 0]	[]

chart[2]

[0]	$V \rightarrow created\bullet$	[1, 2, 1]	[]
[1]	$V' \rightarrow V\bullet$	[1, 2, 1]	[[(2, 0)]]
[2]	$V' \rightarrow V \bullet NP$	[1, 2, 1]	[[(2, 0)]]
[3]	$VP \rightarrow V'\bullet$	[1, 2, 1]	[[(2, 1)]]
[4]	$VP \rightarrow V' \bullet Adv$	[1, 2, 1]	[[(2, 1)]]
[5]	$VP \rightarrow V' \bullet P\ NP$	[1, 2, 1]	[[(2, 1)]]
[6]	$NP \rightarrow \bullet N'$	[2, 2, 0]	[]
[7]	$NP \rightarrow \bullet Det\ N'$	[2, 2, 0]	[]
[8]	$S \rightarrow NP\ VP\bullet$	[0, 2, 2]	[[(1, 2)], [(2, 3)]]
[9]	$N' \rightarrow \bullet N'\ P\ NP$	[2, 2, 0]	[]
[10]	$N' \rightarrow \bullet Adj\ N'$	[2, 2, 0]	[]
[11]	$N' \rightarrow \bullet N$	[2, 2, 0]	[]
[12]	$\lambda \rightarrow S\bullet$	[0, 2, 1]	[[(2, 8)]]

chart[3]

[0]	$Det \rightarrow that\bullet$	[2, 3, 1]	[]
[1]	$NP \rightarrow Det\ N'\bullet$	[2, 3, 1]	[[(3, 0)]]
[2]	$N' \rightarrow \bullet N'\ P\ NP$	[3, 3, 0]	[]
[3]	$N' \rightarrow \bullet Adj\ N'$	[3, 3, 0]	[]
[4]	$N' \rightarrow \bullet N$	[3, 3, 0]	[]

chart[4]

[0]	$N \rightarrow sound\bullet$	[3, 4, 1]	[]
[1]	$N' \rightarrow N\bullet$	[3, 4, 1]	[[(4, 0)]]
[2]	$NP \rightarrow Det\ N'\bullet$	[2, 4, 2]	[[(3, 0)], [(4, 1)]]
[3]	$N' \rightarrow N' \bullet P\ NP$	[3, 4, 1]	[[(4, 1)]]
[4]	$V' \rightarrow V\ NP\bullet$	[1, 4, 2]	[[(2, 0)], [(4, 2)]]
[5]	$VP \rightarrow V'\bullet$	[1, 4, 1]	[[(4, 4)]]
[6]	$VP \rightarrow V' \bullet Adv$	[1, 4, 1]	[[(4, 4)]]
[7]	$VP \rightarrow V' \bullet P\ NP$	[1, 4, 1]	[[(4, 4)]]
[8]	$S \rightarrow NP\ VP\bullet$	[0, 4, 2]	[[(1, 2)], [(4, 5)]]
[9]	$\lambda \rightarrow S\bullet$	[0, 4, 1]	[[(4, 8)]]

chart[5]

[0]	$P \rightarrow on\bullet$	[4, 5, 1]	[]
[1]	$N' \rightarrow N'\ P \bullet NP$	[3, 5, 2]	[[(4, 1)], [(5, 0)]]
[2]	$VP \rightarrow V'\ P \bullet NP$	[1, 5, 2]	[[(4, 4)], [(5, 0)]]
[3]	$NP \rightarrow \bullet N'$	[5, 5, 0]	[]
[4]	$NP \rightarrow \bullet Det\ N'$	[5, 5, 0]	[]
[5]	$N' \rightarrow \bullet N'\ P\ NP$	[5, 5, 0]	[]
[6]	$N' \rightarrow \bullet Adj\ N'$	[5, 5, 0]	[]
[7]	$N' \rightarrow \bullet N$	[5, 5, 0]	[]

chart[6]

[0]	$Det \rightarrow the\bullet$	[5, 6, 1]	[]
[1]	$NP \rightarrow Det \bullet N'$	[5, 6, 1]	[[(6, 0)]]
[2]	$N' \rightarrow \bullet N'\ P\ NP$	[6, 6, 0]	[]
[3]	$N' \rightarrow \bullet Adj\ N'$	[6, 6, 0]	[]
[4]	$N' \rightarrow \bullet N$	[6, 6, 0]	[]

chart[7]

[0]	$N \rightarrow computer\bullet$	[6, 7, 1]	[]
[1]	$N' \rightarrow N\bullet$	[6, 7, 1]	[[(7, 0)]]
[2]	$NP \rightarrow Det\ N'\bullet$	[5, 7, 2]	[[(6, 0)], [(7, 1)]]
[3]	$N' \rightarrow N' \bullet P\ NP$	[6, 7, 1]	[[(7, 1)]]
[4]	$N' \rightarrow N'\ P\ NP\bullet$	[3, 7, 3]	[[(4, 1)], [(5, 0)], [(7, 2)]]
[5]	$VP \rightarrow V'\ P\ NP\bullet$	[1, 7, 3]	[[(4, 4)], [(5, 0)], [(7, 2)]]
[6]	$NP \rightarrow Det\ N'\bullet$	[2, 7, 2]	[[(3, 0)], [(7, 4)]]
[7]	$N' \rightarrow N' \bullet P\ NP$	[3, 7, 1]	[[(7, 4)]]
[8]	$S \rightarrow NP\ VP\bullet$	[0, 7, 2]	[[(1, 2)], [(7, 5), (7, 11)]]
[9]	$V' \rightarrow V\ NP\bullet$	[1, 7, 2]	[[(2, 0)], [(7, 6)]]
[10]	$\lambda \rightarrow S\bullet$	[0, 7, 1]	[[(7, 8)]]
[11]	$VP \rightarrow V'\bullet$	[1, 7, 1]	[[(7, 9)]]
[12]	$VP \rightarrow V' \bullet Adv$	[1, 7, 1]	[[(7, 9)]]
[13]	$VP \rightarrow V' \bullet P\ NP$	[1, 7, 1]	[[(7, 9)]]

Figure 3.11: A chart for a run of our Earley parser against example 3.6, this time with the parse-forest

grammars are ambiguous, we need a representation dealing with a whole set of possible parse-trees, rather than a single one. Such a set of parse-trees is sometimes referred to as a parse-forest.

Let's start our examination of these pointers with the state $\lambda \rightarrow S\bullet$, that represents the successfully parsed input. The pointer we added, (7, 8), points to the state numbered [8] in *chart*[7], which is $S \rightarrow NP\ VP\bullet$. This could be read as, "This S was successfully parsed because $S \rightarrow NP\ VP\bullet$". This state, again, is augmented with the pointer

$$[[(1,2)],[(7,5),(7,11)]]$$

which demonstrates the full complexity of this data-structure. The array of arrays has to be read in the same order as the rule. Therefore $[(1,2)]$ is related to the NP, and $[(7,5),(7,11)]$ is related to the VP, meaning the NP is the one expanded in state [2] in *chart*[1], and the VP could either be the one from state [5] or [11] in *chart*[7].

This parse-forest can be easily built by augmenting the COMPLETER a bit. Every time the COMPLETER advances a rule by "applying" a complete state, it adds a pointer to the "applying" state (the incomplete one, the one that needs to be advanced), pointing to the "applied" one (the complete state), corresponding to the position of the $\bullet$ in the rule. This would leave the chart with pointers that can easily be traversed using standard tree-handling algorithms.

A more detailled description of an actual implementaion of an Earley-parser is presented in the second part of this paper.

3.4 Feature Structures

In the previous section we already mentioned specifiers from a linguistic point of view. Feature structures are a computational model linguistic specifiers are traditionally implemented with.

We mentioned that terminal-symbols, in our case morphological items, have a set of specifiers associated with them, and that it should be possible that these specifiers constrain the appliability of certain rules.

In the first chapter we showed how a word-form like "doggies" goes through the morphological analyzer, that sets some "global flags", as we called them, like, ROOTFORM=DOG, NUM=PLURAL, BABYTALK=TRUE. Such a morphological item could be returned by the morphological analyzer using a feature-structure that, when denoted as an attribute-value matrix, AVM for short, looks like

$$_{doggies}\begin{bmatrix} \text{CATEG} & N \\ \text{ROOTFORM} & dog \\ \text{NUM} & plural \\ \text{BABYTALK} & true \end{bmatrix}$$

This is simply a way of encoding a set of feature-value pairs. The value of each feature can either be atomic or another feature structure. Let's consider an example that is a little bit more complex.

$$_{swims}\begin{bmatrix} \text{CATEG} & V \\ \text{ROOTFORM} & swim \\ \text{TRANS} & intrans \\ \text{AGREEMENT} & \begin{bmatrix} \text{NUMBER} & SG \\ \text{PERSON} & 3 \end{bmatrix} \end{bmatrix}$$

This example shows the feature AGREEMENT that takes another feature structure as its value. This hierachical organization of feature structures doesn't only allow to logically group subsets of related features together, and easily access them, it also makes it possible to build more complex data-structures like lists or trees, to encode not only lexical items, but all data-structures that arise in NLP, including rules and complete parse-forests.

Although practical systems are often built in a way, that the only data-structure they need to deal with is the feature-structure, encoding rules and even complete parse-forests as feature structures, we will describe the basic idea

using the simplified approach given in Jurafsky & Martin (2000), to avoid going into too much detail. The interested reader is referred to Copestake (2002) for a description of implementing so-called "typed feature structure grammars". This book also puts a strong emphasis on how they are used in the LKB-system (the system for grammar-engineering distributed by Stanford's LinGO-initiative).

In our simplified approach we will augment a "normal" CFG with a feature structure, that constrains its use. The rule $NP \rightarrow Det\ N$ serves as a good example. Say we wanted to augment this rule, so it accepts *this dog*, and it rejects **these dog*.

Let's first consider the feature structures for these words.

$$_{dog}\begin{bmatrix}\text{CATEG} & N\\ \text{ROOTFORM} & dog\\ \text{NUM} & SG\end{bmatrix} \qquad _{these}\begin{bmatrix}\text{CATEG} & Det\\ \text{ROOTFORM} & this\\ \text{NUM} & PL\end{bmatrix} \qquad _{this}\begin{bmatrix}\text{CATEG} & Det\\ \text{ROOTFORM} & this\\ \text{NUM} & SG\end{bmatrix}$$

We could use a rule like:

$$NP \rightarrow Det\ N \begin{bmatrix}\text{NUM} & SG & \\ \text{DET} & \left[\text{NUM}\ \ SG\right] & \\ \text{N} & \left[\text{NUM}\ \ SG\right] & \end{bmatrix}$$

Such a rule reads "Replace a Det and an N by an NP, if, and only if, the Det has the feature NUM=SG and the N has the feature NUM=SG. The resulting NP should then also get the feature NUM=SG.

Feature-structures can use so-called "reentrant" structures. The above feature-structure could be rewritten as follows, using reentrancy.

$$\begin{bmatrix}\text{NUM} & \boxed{1} & \\ \text{DET} & \left[\text{NUM}\ \boxed{1}\right] & \\ \text{N} & \left[\text{NUM}\ \boxed{1}\ SG\right] & \end{bmatrix}$$

In this case the features "refer" to each other's value, which wouldn't change anything in this example, since all the NUM-features would still be bound to SG. The advantage of such a notation comes in, when considering feature-structures

like the following one, that don't give a feature an actual value, yet still require it to be equal.

$$\begin{bmatrix} \text{NUM } \boxed{1} \\ \text{DET } \begin{bmatrix} \text{NUM } \boxed{1} \end{bmatrix} \\ \text{N } \begin{bmatrix} \text{NUM } \boxed{1} \end{bmatrix} \end{bmatrix}$$

This would enable us to use one augmented rule, to handle both singular and plural NPs, rather than using two distinct rules, the only difference being NUM's value.

Therefore our complete rule, accepting NPs like *this dog*, and rejecting **these dog* would look like:

$$NP \rightarrow Det\ N \begin{bmatrix} \text{NUM } \boxed{1} \\ \text{DET } \begin{bmatrix} \text{NUM } \boxed{1} \end{bmatrix} \\ \text{N } \begin{bmatrix} \text{NUM } \boxed{1} \end{bmatrix} \end{bmatrix}$$

3.4.1 Unification

In this section we will introduce one of the mightiest operators in computer-science, called "unification" and how it is applied to feature structures, in order to equip them with a limited degree of "intelligent" behavior. The unification-operator is simple, yet so powerful, that it suffices to give programming-languages (like PROLOG) the ability to read and write simple and complex data-structures, to perform logical and therefore numerical operations and to control its program's runtime-flow (usually in combination with a search-strategy like backtracking).

This section will only give the basic ideas of unification by showing a few examples. The reader is again referred to Jurafsky & Martin (2000) and Copestake (2002) for a detailled description of unification and its applications to language-processing, and to Sterling & Shapiro (1994) which gives the details on how unification is used in logic programming languages.

Let's start by considering the most basic example of the unification operator (written $\sqcup$ in the following).

$$\left[\text{NUMBER } SG\right] \sqcup \left[\text{NUMBER } SG\right] = \left[\text{NUMBER } SG\right]$$

$$\left[\text{NUMBER } SG\right] \sqcup \left[\text{NUMBER } PL\right] \textit{ Fails!}$$

This illustrates how unification can be used for simple equality checks. Unification succeeds, if two completely specified feature-structures are actually equal, returning that feature structure, and failing otherwise.

$$\left[\text{NUMBER } SG\right] \sqcup \left[\text{NUMBER } []\right] = \left[\text{NUMBER } SG\right]$$

$$\left[\text{NUMBER } SG\right] \sqcup \left[\text{PERSON } 3\right] = \begin{bmatrix} \text{NUMBER} & SG \\ \text{PERSON} & 3 \end{bmatrix}$$

In these cases the feature-structures are "compatible". Values that are left unspecified can be matched against any value. This is how unification "merges" compatible feature-structures.

$$\begin{bmatrix} \text{NUM} & \boxed{1} \\ \text{DET} & \left[\text{NUM } \boxed{1}\right] \\ \text{N} & \left[\text{NUM } \boxed{1}\right] \end{bmatrix} \sqcup \left[\text{DET } \begin{bmatrix} \text{ROOTFORM} & \textit{this} \\ \text{NUM} & PL \end{bmatrix}\right] = \begin{bmatrix} \text{NUM} & \boxed{1} \\ \text{DET} & \begin{bmatrix} \text{ROOTFORM} & \textit{this} \\ \text{NUM} & \boxed{1}\ PL \end{bmatrix} \\ \text{N} & \left[\text{NUM } \boxed{1}\right] \end{bmatrix}$$

This shows how unification works for reentrant data-structures. While reentrancy from the first argument is preserved, the data from the second argument can be successfully "merged in". This has the consequence, that the value for the *Det*'s NUM-feature gets bound to *PL*, and so do the other NUM-feature's values, that are required to be equal to it.

From a linguistic point of view this behavior comes very handy. Consider a morphological analyzer confronted with the word-form *fish*. The morphological analyzer has no way of telling whether it is singular or plural. However a human understander would have no problem telling that *fish* in *this fish* is singular, and *fish* in *these fish* is plural.

This can be easily implemented using unification. A morphological analyzer would return a feature-structure that doesn't specify the NUM-feature for *fish*.

When unifying that item in the course of parsing the NP *these fish*, unification would automatically bind the *fish*'s NUM-feature to *PL*.

$$\begin{bmatrix} \text{NUM } \boxed{1} \\ \text{DET} \begin{bmatrix} \text{ROOTFORM } \textit{this} \\ \text{NUM } \boxed{1}\ \textit{PL} \end{bmatrix} \\ \text{N} \begin{bmatrix} \text{NUM } \boxed{1} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{N} \begin{bmatrix} \text{ROOTFORM } \textit{fish} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{NUM } \boxed{1} \\ \text{DET} \begin{bmatrix} \text{ROOTFORM } \textit{this} \\ \text{NUM } \boxed{1}\ \textit{PL} \end{bmatrix} \\ \text{N} \begin{bmatrix} \text{ROOTFORM } \textit{fish} \\ \text{NUM } \boxed{1} \end{bmatrix} \end{bmatrix}$$

3.4.2 Parsing with Feature Structures

In this chapter we will refine our Earley-Parser with two goals in mind: blocking constituents violating unification-constraints, and providing a richer representation for constituents using our framework based on feature-structures.

This can be done with two slight modifications to the Earley-Parser presented earlier. The first one concerns the representation of a state in the chart. A state like $N' \rightarrow N'\ P \bullet NP[3,5]$ would now be represented like

$$N' \rightarrow N'\ P \bullet NP[3,5] \begin{bmatrix} \text{N'} & \begin{bmatrix} \text{N} \begin{bmatrix} \text{ROOTFORM } \textit{sound} \\ \text{NUM } \textit{SG} \end{bmatrix} \end{bmatrix} \\ \text{P} & \begin{bmatrix} \text{ROOTFORM } \textit{on} \end{bmatrix} \\ \text{NP} & [] \end{bmatrix}$$

This new representation simply adds a new field to the state, carrying the feature-structure. A machine might represent it as a directed acyclic graph, DAG for short, which is basically the data-structure behind our feature-structures.

The second change affects the algorithm itself, the COMPLETER to be precise. Recall that the COMPLETER is that part of the Earley-algorithm that takes care of advancing every state that "is looking for" a symbol that has just been completed. In our revised Earley-COMPLETER, we can now try to unify the feature-structure of the state that is to be advanced with the feature-structure of the state that is already complete. The result of this unification is stored as the new feature-structure associated with this state. If the unification fails, the state is not

advanced at all, therefore blocking a constituent violating a unification-constraint from "taking part" in the parse.

Although we will not give a full Earley-chart for a parse (which is left as an exercise for a reader with a really long sheet of paper), we will try to give an example.

(3.10) Thick strings are better than thin ones.

(3.11) Thick strings is better than thin ones.

In the course of parsing example 3.10 the COMPLETER will come across a state

$$N' \rightarrow Adj\ N' \bullet [0,2] \begin{bmatrix} \text{ADJ} & \begin{bmatrix} \text{ROOTFORM} & \textit{thick} \\ \text{NUM} & \boxed{1} \end{bmatrix} \\ \text{N'} & \begin{bmatrix} \text{N} & \begin{bmatrix} \text{ROOTFORM} & \textit{string} \\ \text{NUM} & \boxed{1}\ \textit{PL} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

This state represents the complete constituent *Thick strings* as an N'. The COMPLETER can now use this state to complete the state

$$NP \rightarrow \bullet N'[0,0] \begin{bmatrix} \text{NUM} & \boxed{1} \\ \text{N'} & \begin{bmatrix} \text{NUM} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

This can be done by unifying the complete state with the $fs4.n'$-part of the one that is to be completed, to give

$$NP \rightarrow N' \bullet [0,2] \begin{bmatrix} \text{NUM} & \boxed{1} & & \\ \text{N'} & \begin{bmatrix} \text{ADJ} & \begin{bmatrix} \text{ROOTFORM} & \textit{thick} \\ \text{NUM} & \boxed{1} \end{bmatrix} \\ \text{N'} & \begin{bmatrix} \text{N} & \begin{bmatrix} \text{ROOTFORM} & \textit{string} \\ \text{NUM} & \boxed{1}\ \textit{PL} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

What we have parsed so far is the NP *thick strings*, and we have found out that it is plural. Now we can advance the S rule using our completed NP.

$$S \rightarrow NP \bullet VP[0,2] \left[\begin{array}{l} \text{NP} \left[\begin{array}{l} \text{NUM } \boxed{1} \\ \text{N'} \left[\begin{array}{l} \text{ADJ} \left[\begin{array}{ll} \text{ROOTFORM} & \textit{thick} \\ \text{NUM} & \boxed{1} \end{array}\right] \\ \text{N'} \left[\text{N} \left[\begin{array}{ll} \text{ROOTFORM} & \textit{string} \\ \text{NUM} & \boxed{1}\ \textit{PL} \end{array}\right]\right] \end{array}\right] \end{array}\right] \\ \text{VP} \left[\text{NUM } \boxed{1}\right] \end{array}\right]$$

After a while the parser will also have the VP, that is in question accessible. The VP from example 3.10 would give a state like

$$VP \rightarrow V' \bullet [2,7] \left[\begin{array}{l} \text{V'} \left[\begin{array}{l} \text{NUM } \boxed{1} \\ \text{V} \quad \left[\begin{array}{ll} \text{ROOTFORM} & \textit{be} \\ \text{NUM} & \boxed{1}\ \textit{PL} \end{array}\right] \\ \ldots \end{array}\right] \end{array}\right]$$

while the VP from example 3.11 would look like

$$VP \rightarrow V' \bullet [2,7] \left[\begin{array}{l} \text{V'} \left[\begin{array}{l} \text{NUM } \boxed{1} \\ \text{V} \quad \left[\begin{array}{ll} \text{ROOTFORM} & \textit{be} \\ \text{NUM} & \boxed{1}\ \textit{SG} \end{array}\right] \\ \ldots \end{array}\right] \end{array}\right]$$

(the only difference being NUM's value). When the completer tries to advance the S-rule by unifying this VP with the state for the S, unification succeeds in example 3.10, to give

$$S \rightarrow NP\ VP \bullet [0,7] \left[\begin{array}{l} \text{NP} \left[\begin{array}{l} \text{NUM } \boxed{1} \\ \text{N'} \left[\begin{array}{l} \text{ADJ} \left[\begin{array}{ll} \text{ROOTFORM} & \textit{thick} \\ \text{NUM} & \boxed{1} \end{array}\right] \\ \text{N'} \left[\text{N} \left[\begin{array}{ll} \text{ROOTFORM} & \textit{string} \\ \text{NUM} & \boxed{1}\ \textit{PL} \end{array}\right]\right] \end{array}\right] \end{array}\right] \\ \text{VP} \left[\begin{array}{l} \text{V'} \left[\begin{array}{l} \text{NUM } \boxed{1} \\ \text{V} \quad \left[\begin{array}{ll} \text{ROOTFORM} & \textit{be} \\ \text{NUM} & \boxed{1} \end{array}\right] \\ \ldots \end{array}\right] \end{array}\right] \end{array}\right]$$

In example 3.11 such a unification would fail, and therefore the S-state would not be completed by applying this state.

Chapter 4

Semantics

McCarthy (1989) makes a statement about his notion of "common-sense knowledge", that fits perfectly into our idea of semantics.

> Common-sense knowledge includes the basic facts about events (including actions) and their effects, facts about knowledge and how it is obtained, facts about beliefs and desires.

We already pointed out that FOPC and other equivalent first-order logic languages are commonly used to describe these facts about events, actions and so on, which is why, in this section, we will mainly be concerned with the problem of providing an interface between what syntactic analysis left us with (let's say a parse forest, to keep it simple), and an FOPC-representation of the underlying meaning. Such a representation would enable a computer-system to draw conclusions from it, which shall suffice as evidence that the computer understood the sentence, in our rather pragmatic approach to the "myths and magics" of the true nature of intelligence and understanding.

4.1 Overview

(4.1) Steve plays the guitar.

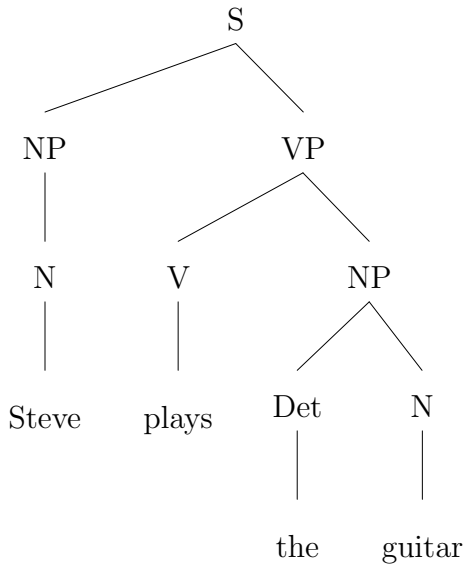


Figure 4.1: A parse-tree for example 4.1

In order to give the reader a rough overview of semantic analysis, in this section we will consider example 4.1 and try to develop an adequate meaning representation by composing simpler meaning representations of the constituents, based on the syntax-tree given in figure 4.1.

Let's first try to capture the meaning of the word *guitar* in this sentence, which turns out to be rather trivial: The term *guitar* can be used "for every g such that g is a guitar".

$$\forall g IsA(g, Guitar)$$

Neglecting the meaning of the determiner *the*, we can move on to the verb heading the VP containing this phrase, namely *plays*.

In example 4.1, we can say that *play* means that someone (p) is capable of playing something (i), which could be captured by

$$\exists p \forall i Plays(p, i)$$

The NP *the guitar* can be viewed as a parameter that is passed as an i to this fact.

$$\exists p \forall i Plays(p, i) \wedge IsA(i, Guitar)$$

If we move up another level to the S headed by the NP *Steve* with a semantic representation

$$\exists p HasName(p, Steve)$$

we can do the same thing again, to get

$$\exists p \forall i Plays(p, i) \wedge IsA(i, Guitar) \wedge HasName(p, Steve)$$

What our system just did was successfully understand that *Steve plays the guitar*, describes a relation $Plays$ that holds between a p and every i such that another relation IsA holds between i and some constant $Guitar$ and yet another relation $HasName$ between p and some constant $Steve$.

Note that terms like $Guitar$ and $Steve$ are simple constants bearing no meaning as such. They have to be defined in the semantic framework. If, for example, there is a rule $IsA(Evo, Guitar)$, we can deduce that $Steve$ is capable of playing Evo, which could be the name of Steve's favourite guitar.

4.1.1 Ambiguity

Note that a computer trying to do what we just did would, of course, be confronted with ambiguity. The word *play* can have several meanings, some of which are given in M1-M5.

$$\exists p \forall i IsCapableOfPlayingInstrument(p, i)$$

as in *Steve plays the guitar*, or

$$\exists p, g IsCapableOfPlayingGame(p, g)$$

as in *Steve plays chess*, or

$$\exists p, tUsuallyPlaysWithToy(p, t)$$

as in *Steve plays with dolls*,

or

$$\exists p, mPlaysInABandTogetherWith(p, m)$$

as in *Steve plays with Joe* etc.

Differences can only be observed in predicate names and quantifiers, which is another piece of evidence that we've successfully singled out semantic problems, since disambiguation can only be done on a semantic basis (it requires knowledge of the problem-domain, and logical inference to disambiguate, for example, that if *the guitar* is an instrument and not a game, the meaning of *play* is M1, rather than M2).

Syntactic ambiguities, such as deciding in the sentence *We bought the dog* whether *bought* is transitive or ditransitive as in *We bought the dog a nice toy* would never "make it that far" in our analysis, since a rule like $VP \rightarrow VNPNP, cns1$ and another one like $VP \rightarrow VNP, cns2$ would already have distinguished between the two lexemes, and we can freely provide them with different semantic representations, given

$$_{cns1}\left[\text{V}\left[\text{TRANS}\ \ ditrans\right]\right]$$

$$_{cns2}\left[\text{V}\left[\text{TRANS}\ \ trans\right]\right]$$

The problem we are facing when we have to choose the "correct" meaning from M1-M5, is often referred to as "sense-ambiguity".

4.1.2 Knowledge

One might easily be misled into thinking that ambiguity in the above example was created quite artificially by introducing so many predicates for *play*, and that the problem could easily be circumvented by proposing only a single predicate $Plays$ as in the previous section.

The point that is crucial to the understanding of FOPC and its use as a "meaning representation" is that, by themselves, predicates and symbols do not carry meaning. They rather introduce it indirectly by establishing equivalence classes between equal (or rather unifiable) symbols.

Ambiguity arises as soon as semantic knowledge, that is modelled neither in a unique "dictionary definition" of the lexeme or grammar rule, nor in the specification of the problem domain (the machine's knowledge of the world it operates in), is neccessary to successfully create a unique meaning representation. In a way it could be viewed as the consequences of incompleteness in modelling the neccessary knowledge.

Say we wanted our system to tell whether somebody is intelligent, and whether somebody is musically talented, and we equip our system with a few facts about its problem-domain:

$$IsDifficult(Chess)$$

$$IsDifficult(Mastermind)$$

$$\forall x, gIntelligent(x) \Leftarrow IsCapableOfPlayingGame(x, g) \wedge IsDifficult(g)$$

$$IsDifficult(Violin)$$

$$IsDifficult(Clarinet)$$

$$\forall x, iTalented(x) \Leftarrow IsCapableOfPlayingInstrument(x, i) \wedge IsDifficult(i)$$

Given this scenario it would, of course, be possible to use a single predicate like $Plays$, but this would require us to rewrite these rules as:

$$IsDifficult(Chess)$$

$$IsDifficult(Mastermind)$$

$$IsA(Chess, Game)$$

$$IsA(Mastermind, Game)$$

$$\forall x, gIntelligent(x) \Leftarrow Plays(x, g) \wedge IsA(x, Game) \wedge IsDifficult(g)$$

$$IsDifficult(Violin)$$

$$IsDifficult(Clarinet)$$

$$IsA(Violin, Instrument)$$

$$IsA(Clarinet, Instrument)$$

$$\forall x, iTalented(x) \Leftarrow Plays(x, i) \wedge IsA(i, Instrument) \wedge IsDifficult(i)$$

By doing so we actually disambiguated the input by providing the system with information about what is an instrument and what is a game. This new information, that is additionally (to the model of the problem domain) required for disambiguation could be formalized as follows:

$$IsA(Chess, Game)$$

$$IsA(Mastermind, Game)$$

$$IsA(Violin, Instrument)$$

$$IsA(Clarinet, Instrument)$$

$$\forall x, oIsCapableOfPlayingGame(x, o) \Leftarrow Plays(x, o) \wedge IsA(o, Game)$$

$$\forall x, oIsCapableOfPlayingInstrument(x, o) \Leftarrow Plays(x, o) \wedge IsA(x, Instrument)$$

A human understander takes this knowledge required for semantic disambiguation of natural language from a knowledge-repository known as "common sense". Making it accessible to machines is a rather difficult task and one of the most central problems for symbolic artificial intelligence. Maybe efforts like CYC Lenat (1995) will make "common sense" possible for machines one day, but this is certainly not within the scope of this paper.

4.2 A Linguistic Perspective to Meaning

In the above section it got obvious that meaning is a rather difficult concept to formalize. What is meaning? Is it possible to capture meaning in a formal system? Does a single word have meaning? Is it possible to tell how many meanings a single word has? Does a sentence have meaning? Can the meaning of a sentence be composed by combining "smaller meanings" of the words and grammatical rules that make up the sentence?

These are questions we are facing here, and the answer to most of them might be "no", possibly uttered by an "old-school"-philosopher, followed by a big "but" uttered by someone following the more recent tradition of analytic philosophy: No, but if we make the right presumptions, accept little weaknesses and apply a certain degree of pragmatism, we can have considerable success when measured in terms of practical application of our theory.

We might easily agree that language is, at its core, a symbolic system, and any symbolic system can be viewed as a language. This is why the process of "understanding" language is sometimes viewed as the process of "translating" from one language, such as English, into another language, such as FOPC.

Note that this step might be seen as illegal, because such a view would put the semantic level into the interface between one language and another. Let's consider the example of a human trying to translate an English sentence into the German language. The translator would have to "understand" the English sentence - something we tried to deal with using the notion of capturing the meaning of a sentence in a "semantic representation". This "understanding" would then be used by a human translator to form a German sentence. Now, if we wanted a computer to translate an English sentence into FOPC, and it would first have to capture the meaning in a "semantic representation", what would this representation be? Certainly not FOPC.

In a way, we completely skip the idea of meaning. What we want to develop

here could be viewed as a model of translating from one language into another without having to really understand. Such a model is backed by presumptions which have turned out to be fruitful in their application rather than the metaphysical truth behind the concept of meaning.

4.2.1 Sense

A tool that's very useful for translating from one language into another is a dictionary - a dataset associating a set of words in one language with a set of words in the other language making statements about their interchangability. If I didn't know the German word for *guitar* and I looked it up in a dictionary it would give me *Gitarre*, suggesting that *guitar* is a linguistic symbol used by the English language to refer to the same sense as *Gitarre* in German.

Note that sense is, in our theory, a purely virtual level. This can be seen, when we consider ambiguity. The example we just gave was very simple: *guitar* refers to exactly one sense, the same and the only sense that *Gitarre* refers to. Let's consider an example that involves ambiguity. If we looked up *bass* in a dictionary it would give us two German words: *Bass* and *Barsch*. The first one is a musical instrument, the second one is a kind of fish. A good dictionary helps in doing this kind of disambiguation by giving a glossary describing the terms.

Note the parallelity between this concept, and the exemplaric formalization of an English sentence in FOPC we just did. The glossary is exactly the same "information that is additionally (to the model of the problem domain) required for disambiguation", we were using in the previous section.

What does this mean for the concept of sense? Our dictionary, disambiguating its entries by giving glossaries, is now using a semantic level to anchor the process of associating English and German words. And again we run into the same limitation we were just talking about. What semantic level can be used by a computer translating English into FOPC?

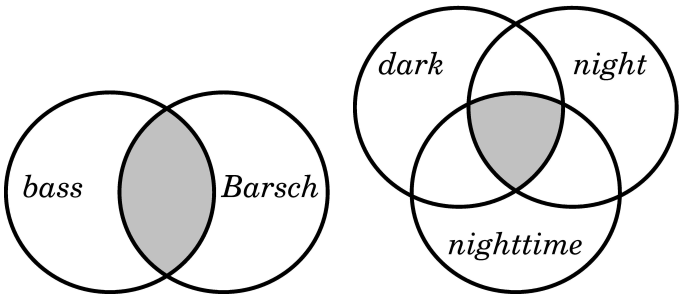
We can get around that problem by sticking with our view of understanding by translating and viewing the semantic level in between as a black-box. There is no problem in modelling a data-structure that is aware of one sense described by the tuple [*bass*,*Barsch*] and one sense by [*bass*,*Bass*]. We can view "true meaning" as attached to this data-structure like *[bass,Bass] is a musical instrument* and *[bass,Barsch] is a kind of fish*, yet when modelling language for use by an automaton we simply leave out this attachment.

Putting it simply: A traditional English-German-dictionary makes statements like "You can use the word *Bass* for *bass* if you are talking about a musical instrument" and "You can use the word *Barsch* for *bass* if you are talking about fish". In our analytic dictionary we cannot make these statements, because we can't model the "if ..."-part of that statement, but we can make statements like "There exists a sense that is the intersection of the meanings of *Bass* and *bass*" and "There exists *another* sense that is the intersection of the meanings of *Barsch* and *bass*". If we do that we establish a so called *differential theory* of semantics.

We developed this idea by following an example of translating from English to German, which is why we came up with English/German-tuples like [*bass*,*Barsch*] to account for our black-box-notion of sense, but it is important to note that this is not the only way one could go about this. The only thing that really matters is to capture meaning by some sort of unambiguous data-structure mapping to linguistic symbols.

The creators of WordNet[1], who emphasized the importance of synonymy in capturing meaning did something similar. In WordNet so-called *synsets* are used to account for sense. These synsets aren't tuples as in our example, but rather sets of English words that can be used interchangably in some sense. For example (*night*,*nighttime*,*dark*) is one synset. The words could be used interchangably in a sentence like *She walked home alone in the night*, but not in others like *She*

[1]see Miller et al. (1993), Miller (1993), Fellbaum et al. (1993), Fellbaum (n.d.) and Beckwith et al. (n.d.) for details on WordNet



(a) (b)

Figure 4.2: A black-box-view of sense

wants to go out Saturday night.

The parallelity between our approach, and that of WordNet can also be depicted graphically as in figure 4.2. Figure 4.2(a) shows how sense is created in our example as an intersection of the meanings of an English and a German word, and figure 4.2(b) accounts for WordNet's idea of synsets. Figure 4.3 shows how different words contribute to two distinct senses of the same word, *dark*. One sense is created by the synset (*night*,*nighttime*,*dark*) and one by (*black*,*sinister*,*dark*). The examples chosen here shouldn't mislead the reader into thinking that a synset always consists of three words. There can, of course, be more or less than three words contributing to a synset, the example was chosen for the sake of readability of the diagram.

4.2.2 Reference

Although sense is a very important theoretic aspect in describing meaning it is still somewhat abstract. If Steve's son asked him what a guitar was, he probably wouldn't come up with something like "The concept behind what could be described by the English word *guitar* and the German word *Gitarre*", he would merely grab *Evo*, his favourite guitar, and say, "This is a guitar", making use of

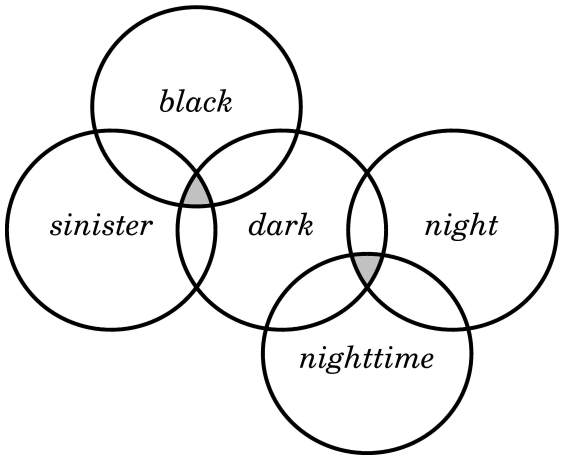


Figure 4.3: Two senses of *dark*

the concept of reference, which could be viewed as the ultimate goal of language. The symbols used in language are ultimately used to refer to concrete or abstract entities we have in mind.

While Steve might think of *Evo*, a white electric guitar with steel-strings, when he hears the word *guitar*, Jack might think of his favourite guitar, *Liz*, a wooden acoustic guitar with nylon-strings.

This suggests a kind of referential ambiguity that arises when mapping from a sense to its referent, analogously to sense-ambiguity that arises when mapping from a word to its sense.

Not only is it possible that one sense might be associated with different referents, it is also possible that distinct senses resolve to the same referent. For example, *the capital of Norway* and *Oslo* refer to the same place, somewhere in Scandinavia. Does this mean that *the capital of Norway* and *Oslo* have the same sense - are synonyms? Certainly not, because if the Norwegian government decides to declare Nuuk the new capital, *Oslo* would still be Oslo.

4.2.3 Lexical Semantics

Now that we know how a word is related to its sense, which is again related to its referent, we can have a more detailled look at how words and their senses relate to each other, a study widely known as lexical semantics. This field has seen a lot of research in the recent past. WordNet was definitely one of the more ambitioned projects, with its attempt to actually build a dictionary organizing words and their senses and providing valuable information about how they relate to each other. These are sometimes referred to as *lexical databases*. In this section we will have a closer look at those relations WordNet attempts to cover.

Synonymy

The most important relation in WordNet is synonymy, because it is the synonymy-relation that enables WordNet to capture meaning. It is usually viewed to hold between two expressions, if the substitution of one for the other never changes the truth-value of the sentence the substitution is made in. This is where the concept of meaning comes in. If two expressions are substitutable, then they could be said to *mean* the same. Miller et al. (1993) point out a problem with this definition of synonymy, and offer a solution.

> By that definition, true synonyms are rare, if they exist at all. A weakened version of this definition would make synonymy relative to a context: two expressions are synonyms in a linguistic context C if the substitution of one for the other in C does not alter the truth value.

This is why, in WordNet, words are considered synonyms, if there is a linguistic context where the words are synonymous.

The use of substitutability to define synonymy has two important consequences: First, two words can only be synonymous if they belong to the same

POS. A verb and a noun, for example, are never substitutable. Secondly synonymy is symmetric: if x is synonymous to y, then y is synonymous to x.

Lexical Relations

Lexical relations are relations that hold between word-forms, and not neccessarily between their senses. We will give a brief overview of the most commonly used lexical relations.

Polysemy/Homonymy Although polysemy is a relation that appears in WordNet only implicitly, it is of central importance, for example, for sense-disambiguation. Two words are polysemous if their word-forms are the same, but their senses aren't. In a way polysemy is closely related to synonymy. Both arise from ambiguity in the mapping between word-form and word-sense. While when we're mapping from a sense to its word-forms we are dealing with synonymy, we are dealing with polysemy when we're mapping from a word-form to its sense. Homonymy is very similar to polysemy. Homonymy is usually viewed to hold between word-forms whose senses are completely unrelated, while two words can be polysemous also when their senses are somehow related, as long as they aren't equal. This was mentioned only for preventing confusion, because homonymy and polysemy are widely used in that way. For our differential theory of sense, this is somewhat awkward, because here sense can never be related yet unequal, which is why we will use the terms *polysemy* and *homonymy* interchangably in the rest of this paper.

Antonymy is, although speakers of English have little difficulty recognizing it, rather difficult to formalize. It could be thought of as the opposite of polysemy. Sometimes the antonym of x is *not-x*, but not always. Miller et al. (1993) use the example of *rich* and *poor*. These words are antonymous, but if someone is not rich, it doesn't neccessarily mean that he is poor. It is interesting to mention, that

antonymy, besides synonymy, is the only relation that is maintained in WordNet for all parts of speech.

Semantic Relations

Semantic relations are relations that hold between senses, in contrast to the lexical relations we mentioned in the previous section, which hold between word-forms.

Hyponymy and Hypernymy are also called subordination and superordination. They are used to organize the lexical database hierachically (for example, to set up an inheritance system, a concept we will deal with in greater detail later). For example *guitar* is a hyponym of *stringed instrument* which is again a hyponym of *instrument*. Generally one can say, that x is a hyponym of y if a native speaker accepts sentences like *x is a kind of y*. Hyponymy is transitive, therefore if x is a hyponym of y and y is a hyponym of z, then x is a hyponym of z. Unlike synonymy, hyponymy is asymmetrical. If x is a hyponym of y, y is not a hyponym of x, but rather a hypernym. WordNet maintains hyponymy and hypernymy for nouns and verbs.

Meronymy and Holonymy can also be used to organize a database hierachically, with some reservations. For example *neck* is a meronym of *guitar*. Generally one can say that x is a meronym of y if a native speaker accepts sentences like *an x is a part of a y*. This relation is also transitive and asymmetrical. Again if x is a meronym of y and y is a meronym of z, then x is a meronym of z and if x is a meronym of y, then y is a holonym of x. Sometimes additional classification of meronymy is done. WordNet uses three kinds of meronymy/holonymy: member-, substance- and part-meronymy, which are maintained only for nouns (which doesn't come as a surprise, but was mentioned for completeness).

	F_1	F_2	F_3	...	F_n
M_1	$E_{1,1}$	$E_{1,2}$			
M_2		$E_{2,2}$			
M_3			$E_{3,3}$		
...				...	
M_m					$E_{m,n}$

Figure 4.4: A Lexical Matrix

Entailment is used in WordNet to organize verbs. For example *snore* entails *sleep*. The term *entailment* is defined in logic (where it is also known as *strict implication*) as follows: A proposition P entails a proposition Q if it is under no circumstances possible to make P true and Q false. In lexical semantics a verb p entails a verb q if the statement *Someone p* logically entails the statement *Someone q*. Lexical entailment is a unilateral relation: If q entails p, then p cannot entail q. See Fellbaum (n.d.) for the details on entailment in WordNet.

The lexical matrix

Figure 4.4 shows a matrix, that could be thought of as a datastructure for mapping words to their senses, and senses to their words. This table, taken from Miller et al. (1993), could be thought of as another way of depicting the same concept as figure 4.2(b), namely formalizing sense in a differential approach to meaning. We could view $F_1..F_n$ as symbols representing all possible word-forms, and $M_1..M_n$ as representing all possible meanings. An entry like $E_{1,1}$ would be read *The word-form* F_1 *can be used to express the meaning* M_1.

This simple data-structure deals elegantly with the two most important relations of lexical semantics: synonymy and polysemy. If we wanted to look up the meaning of a word-form F_2, we would simply have to look at column F_2 to find two appropriate meanings: M_1 and M_2, which confronts us with polysemy. If

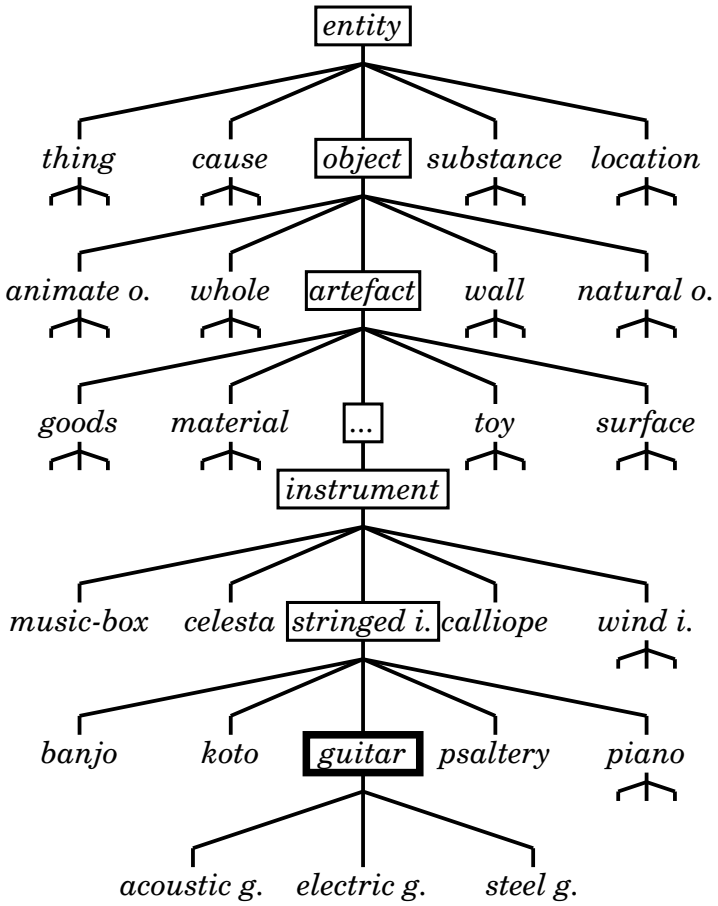


Figure 4.5: A sample of WordNet's hyponymy-structure

we wanted to look up a word-form for a meaning we have in mind, say M_1, we would have a look at the row M_1, discovering two possible word-forms, namely F_1 and F_2, which confronts us with synonymy.

The lexical matrix also helps to visualize the layer in between word-form and word-meaning, namely the data-cells denoted $E_{j,i}$. In our simple approach this would be a truth-value, saying "this association is valid" or "this association is invalid", but in a more sophisticated approach to lexical semantics it might be desireable to use a numeric value, for example saying "this association is true at a probability of 0.67", or signals helping with disambiguation, etc.

The Lexical Inheritance System

A lexical inheritance system is used, for example, in WordNet to organize nouns and equip them with a limited degree of semantic information. How do conventional dictionaries get semantic information across? If we looked up the word *guitar* in a dictionary, it would give us a glossary like *a stringed instrument that is small, light, made of wood, and has six strings usually plucked by hand or a pick*. Now what is a *stringed instrument*? If we looked that word up in the dictionary, we would get something like *a musical instrument producing sound through vibrating strings*. What does that tell us about guitars? Obviously, that a guitar is *a musical instrument producing sound through vibrating strings, that is small, light, made of wood, and has six strings usually plucked by hand or a pick*.

What we just did was, we resolved the lexical inheritance system of our dictionary. We could go on like this for quite a while, looking up *guitar*, then *stringed instrument*, then *instrument* until we end up at a word, that stands for itself, like *entity*.

That we already mentioned the sequence *guitar-stringed instrument-instrument* in this paper is not a coincidence: We mentioned it as an example for hyponymy, which is the basic building-block organizing the nouns in our dictionary into a hierachical system as depicted in figure 4.5. In WordNet the top of this tree-structure is the synset for the word *entity* which is the most abstract "thing" a noun can be. Then WordNet tells us about different kinds of "entities", including objects, places, agents etc. When we go a step down this hierachy towards, say, *object* the concept of inheritance allows us to view an object as an entity. This implies that an *object* has all attributes, parts and functions, that an *entity* has. If we go down another step in this hierachy, say to the synset for *artifact* we are again allowed to view an artifact as an object. Therefore *artifact* inherits all attributes, parts and functions from *object*, and implicitly also from *entity*,

because *object*, as we just mentioned, inherits all attributes, parts and functions from *entity*. If we keep on doing this, going down the hierachy, until we arrive at *guitar* we know that a guitar is something "which is perceived or known or inferred to have its own physical existence", although this can not be found in the definition of a *guitar*, because this fact was inherited all the way down from *entity* to *guitar*. The application of this concept of inheritance to our hierachical system, created by the hyponymy-relations, is what makes this hierachy an inheritance system. The details on lexical inheritance and its use in WordNet can be found in Miller (1993).

4.3 A Formal Perspective to Meaning

Now that we know how to capture the meaning of a single word we can go on to develop a formal approach to meaning and its representation in FOPC.

4.3.1 Representing Lexemes

(4.2) Steve plays the guitar.

When developing a meaning representation for example 4.1, repeated here as example 4.2, in section 4.1 we started out by representing the meaning of the word *play* as the following FOPC-expression:

$$\exists p \forall i Plays(p, i)$$

Mapping words to this kind of expression is the dictionary's job, since both words and expressions like the above one could, in their language, somehow be viewed as the nuclear meaning-carrying unit.

First of all, it is important to recognize the need for the consistent use of predicates and predicate-structures in dictionary definitions. FOPC doesn't as such

provide data-structures for handling knowledge or common-sense, it's just a formalism for describing relations among symbolic expressions. The way a problem-domain is actually modelled in FOPC places some restrictions on what dictionary-definitions of words might look like. These restrictions could be thought of as an interface between the "natural-language-part" (dictionary, grammar, etc.) and the "processing-part" (the formalization of the problem domain) of our natural-language-processor.

Let's return to the example from section 4.1.2: We had a problem domain for a system telling whether someone is intelligent and whether someone is talented, that looked like:

$$IsDifficult(Chess)$$

$$IsDifficult(Mastermind)$$

$$\forall x, gIntelligent(x) \Leftarrow IsCapableOfPlayingGame(x, g) \wedge IsDifficult(g)$$

$$IsDifficult(Violin)$$

$$IsDifficult(Clarinet)$$

$$\forall x, iTalented(x) \Leftarrow IsCapableOfPlayingInstrument(x, i) \wedge IsDifficult(i)$$

Given this problem domain it would not make sense for the dictionary to map the word *play* to $\exists p \forall i Plays(p, i)$, because given $Plays(Steve, Violin)$ our program cannot deduce $Talented(Steve)$. This is how the definition of the problem-domain implicitly creates an interface that the dictionary has to implement if the whole system is to be operational. This interface would make statements like "For expressing meaning the dictionary can use the predicates $IsDifficult$, $IsCapableOfPlayingGame$ and $IsCapableOfPlayingInstrument$".

This interface is rather problematic since it gives rise to ambiguity and knowledge-problems. How should the dictionary know the difference between $IsCapableOfPlayingGame$ and $IsCapableOfPlayingInstrument$ when deciding how to translate the word *plays*? It is clearly not the dictionary's job to find out (at least in our ap-

proach), because it would require further knowlege about the problem domain to do so, and therefore we have to redefine the problem domain, just as we did in section 4.1.2, so that the interface reads "For expressing meaning the dictionary can use the predicates $IsDifficult$ and $Plays$ as well as $IsA(X, Game)$ and $IsA(X, Instrument)$". This eliminated ambiguity, because every symbolic expression in the English language, e.g. *plays* has exactly one corresponding symbolic expression in the FOPC-modelled problem-domain, $\exists p \forall i Plays(p, i)$ for instance.

4.3.2 Knowledge-Representation in Natural Language Processing

To maintain a certain degree of generality in their problem-domains linguists usually use models from artificial intelligence which were originally targeted towards accounting for true "common-sense" or "knowledge" or whatever term you prefer for McCarthy's level-4-use of logic (McCarthy 1989, p9), but never really reached that goal, yet turned out to be very useful for modelling natural language, which is a symbolic system that operates exactly at that level.

In such a formalism you wouldn't find a predicate like $Plays(p, i)$ to account for the meaning of the verb *play*, but rather a notion of events that can sometimes seem somewhat artificial. Such a formalism would also make use of some very important predicates like IsA or AM to represent certain aspects of modelling nouns, referents and their properties.

Before we will have a look at these conventions in this section, recall that according to McCarthy (1989), common-sense knowledge includes facts about

- events (including actions) and their effects
- knowledge and how it is obtained
- beliefs and desires

- objects and their properties

Verbs: Events and Actions

Therefore to formalize a verb like *play* we would introduce an event to account for the action of *playing*. The action of playing involves two so-called "semantic roles", that of the agent, and that of the experiencer. The agent is that object, which causes the action to happen, the player, in our example. The experiencer is that object which experiences the action, in our case, that "which gets played", the "playee" so-to-speak. (Although words like playee are somewhat awkward, they are commonly used to emphasize the semantic-role-concept, showing the analogy of the playee in a playing-event, the employee in an employing-event, the trustee in a trusting-event and so on.)

Given that, we can define the verb *play* by the playing-event it describes as:

$$\exists p, s, g IsA(p, Playing) \wedge Player(p, s) \wedge Playee(p, g)$$

Therefore the word *play* indicates that "there exists a p, such that p is the event of playing something, the player taking part in p is s and the playee taking part in p is g".

Nouns: Objects ...

The meaning of nouns is widely covered by the IsA-relation we've been using all the time. We can simply describe nouns by atomic symbols. A guitar would then be $Guitar$, as bass a $Bass$ and so on. The Isa-relation associates noun-senses like $Guitar$ and possible referents like Evo. (Evo is, as we've mentioned earlier, Steve's favourite guitar.) Making assertions like $Isa(Evo, Guitar)$.

Distinguishing sense and reference is sometimes a pitfall: *Steve plays the guitar* in its *Steve is capable to play the guitar*-sense cannot be formalized as

$$\exists p IsA(p, Playing) \wedge Player(p, Steve) \wedge Playee(p, Guitar)$$

Given that we use the IsA relation to account for reference (and the $HasName$ relation to account for names) the above statement would not imply

$$\exists p IsA(p, Playing) \wedge Player(p, Steve) \wedge Playee(p, Evo)$$

Once we decide we want to use the IsA-relation, we have to capture a noun like *guitar* by $\forall x Isa(x, Guitar)$ or $\exists x Isa(x, Guitar)$ whenever we are actually talking about the class of all guitars or a specific guitar.

We would therefore have to formalize *Steve plays the guitar* as

$$\exists p \forall s, gIsA(p, Playing) \wedge Player(p, s)$$
$$\wedge HasName(s, Steve) \wedge Playee(p, g) \wedge IsA(g, Guitar)$$

Because given that $Isa(Evo, Guitar)$ and $HasName(Steve, Steve)$ this does entail

$$\exists p IsA(p, Playing) \wedge Player(p, Steve)$$
$$\wedge HasName(Steve, Steve) \wedge Playee(p, Evo) \wedge Isa(Evo, Guitar)$$

which is exactly what we wanted to achieve.

Relations like $HasName(Steve, Steve)$ might seem awkward, but this is due to this example. Weisler & Milekic (2000) show why naming can be dealt with on a separate "linguistic level".

Adjectives: ... and their Properties

In a framework called "intersective semantics" the noun-phrase *a great guitar* would be formalized in the following way:

$$\exists x Isa(x, Guitar) \wedge Isa(x, Great)$$

The meaning of *a great guitar* is, in this framework, viewed as the intersection of the set containing all guitars and the set containing all great things.

Jurafsky & Martin (2000) use three examples for showing why this approach is a bit peculiar.

(4.3) small elephant

(4.4) former friend

(4.5) fake gun

that would be formalized as

$$\exists x Isa(x, Elephant) \wedge Isa(x, Small)$$

$$\exists x Isa(x, Friend) \wedge Isa(x, Former)$$

$$\exists x Isa(x, Gun) \wedge Isa(x, Fake)$$

This would state that a small elephant is a member of the set of small things, that a former friend is a member of the set of friends, which is simply false, and a member of the set of former things, which is somewhat unreasonable, similarly to a fake gun, which is, in this model, considered a gun.

Unfortunately there is no easy way out of this problem, as long as we stick with the principle of compositionality, but we might at least distinguish the IsA relation from the AM-relation, just like Jurafsky & Martin (2000) did, and leave further processing to the problem-domain, defining *a great guitar* as

$$\exists x Isa(x, Guitar) \wedge AM(x, Great)$$

4.3.3 Lambda-Expressions

Before we can move on to account for the semantic representation of grammar-rules we first have to provide a means to represent meanings that are "not yet finished". The dictionary definition of a single word or a phrase of a sentence cannot usually stand for itself, it is rather a partial meaning, a subgoal on our way to the complete meaning-representation of a sentence. This gives rise to the need for an intermediate meaning-representation that accounts for these partial meanings.

The approach is simple: a partial meaning can be viewed as a template, like a form that has to be filled out before it carries any relevant meaning. Representing this can be easily achieved, using the FOPC-formalism, if we make one fundamental extension: the Lambda-symbol, which acts similarly to a quantifier, and states "this is a form-variable that has to be specified in detail in later processing".

(4.6) I play no instrument

(4.7) Nobody plays the piano

Turning back to our example of $\exists p \forall i Plays(p, i)$, one might criticize the use of the quantifiers $\exists$ and $\forall$. Does the word *play* really imply that there exists some p, such that p plays i, and that p can really play every i? Examples 4.6 and 4.7 provide enough evidence, that another formalism is needed to account for the "missing parts" in meaning: The lambda-expression.

Using a lambda-expression we could formalize the maining of *play* as

$$\lambda p, iPlays(p, i)$$

which roughly reads "there is a p and an i, that still have to be specified in detail, but we already know that there is a relation $Plays$ that holds between them".

Let's consider an example that's a bit more interesting. In section 4.1 we represented the VP *plays the guitar* as

$$\exists p \forall i Plays(p, i) \wedge Isa(i, Guitar)$$

Of course this VP doesn't yet "know" who will be the agent, therefore we would have to use the following lambda-expression

$$sampleVP = \lambda p \forall i Plays(p, i) \wedge Isa(i, Guitar)$$

This time we also gave the lambda-expression a name, because we want to introduce the following notation, which creates an expression where the variable

marked by λ in $sampleVP$ gets replaced by the expression $Steve$:

$$sampleVP(Steve)$$

This would be the same as writing

$$\forall i Plays(Steve, i) \wedge Isa(i, Guitar)$$

This can also be done with complex-terms, like

$$sampleVP(\exists e HasName(e, Steve))$$

In general complex-terms take the form

$$< \ quantifier \ variable \ body \ >$$

Our example would, in the first place, resolve to something like

$$\forall i Plays(\ < \ \exists e HasName(e, Steve) \ > \ , i) \wedge Isa(i, Guitar)$$

which isn't really syntactically correct FOPC, but it is possible to convert it back to syntactically correct FOPC by rewriting the predicate, that uses the complex-term

$$P(\ < \ quantifier \ variable \ body \ > \ )$$

as

$$quantifier \ variable \ body \ connective \ P(variable)$$

The connective depends on the quantifier. Variables that are quantified with $\exists$ are connected with $\wedge$, variables that are quantified with $\forall$ are connected with $\Rightarrow$.

Therefore our sample-expression would be rewritten as

$$\forall i \exists e HasName(e, Steve) \wedge Plays(e, i) \wedge Isa(i, Guitar)$$

Such a proceeding is called *lambda-reduction*.

4.3.4 Representing Grammar-Rules

We might easily agree that the way a sentence is put together, the syntax, the grammar-rules putting together the words in order to make up a meaningful sentence, do themselves carry meaning.

A question that is a bit more tricky is what a grammar carrying out semantic analysis should look like. We will use a quite simplistic approach called *syntax-driven semantic analysis*, based on the presumption that semantic analysis can be carried out in exactly the same structure as syntactic analysis.

To be more specific, this means, that if it is, on a syntactic level, possible to deduce the syntax of a VP like

$$[_{VP}\ [_{V}\ plays\ ][_{NP}\ [_{Det}\ the\ ][_{N}\ guitar\ ]]]$$

from its parts, namely the V

$$[_{V}\ plays\ ]$$

the NP

$$[_{NP}\ [_{Det}\ the\ ][_{N}\ guitar\ ]]$$

and the rule putting them together

$$VP \rightarrow V\ NP$$

then it is also possible, on a semantic level, to deduce the semantics of the corresponding VP

$$\lambda s \exists p, g IsA(p, Playing) \wedge Player(p, s)$$
$$\wedge Playee(p, g) \wedge IsA(g, Guitar)$$

from the same parts, namely the V

$$V = \lambda g, s \exists p IsA(p, Playing) \wedge Player(p, s) \wedge Playee(p, g)$$

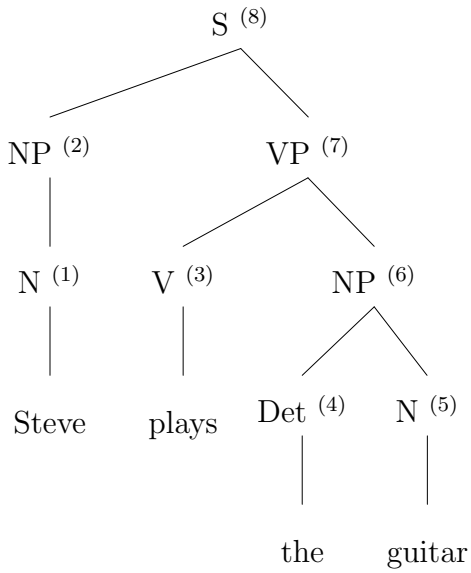


Figure 4.6: Compositional structure

the NP

$$NP = \exists g IsA(g, Guitar)$$

and the rule putting them together, which is

$$V(NP)$$

Figure 4.6 depicts the same idea graphically in greater detail. Figure 4.3.4 is basically a syntax tree of example 4.1, this time in its *There is a guitar that is currently played by Steve*-sense. The presumption of syntax-driven semantic analysis allows us to use this syntax-tree not only as the structure building up the whole syntax from its parts but also for building up the whole semantics from its parts. Figure 4.7 shows what these nodes would in detail look like on a semantic and on a syntactic level. Figure 4.8 shows the grammatical rules that made the derivation of nodes $^{(2)}$,$^{(6)}$,$^{(7)}$ and $^{(8)}$ possible, again on a semantic and on a syntactic level. (The derivation of the other nodes isn't particularly interesting, since they are based on simple dictionary look-ups.)

The semantic rules are, of course, lambda-reductions. The reader is invited

	syntax	semantics
(1)	$[_N$ Steve]	$\exists s HasName(s, Steve)$
(2)	$[_{NP}[_N$ Steve]]	$\exists s HasName(s, Steve)$
(3)	$[_V$ plays]	$\lambda g, s \exists p IsA(p, Playing) \wedge Player(p, s) \wedge Playee(p, g)$
(4)	$[_{Det}$ the]	nil
(5)	$[_N$ guitar]	$\exists g IsA(g, Guitar)$
(6)	$[_{NP}[_{Det}$ the$][_N$ guitar]]	$\exists g IsA(g, Guitar)$
(7)	$[_{VP}[_V$ plays$][_{NP}[_{Det}$ the] $[_N$ guitar]]]	$\lambda s \exists p, g IsA(p, Playing) \wedge Player(p, s)$ $\wedge Playee(p, g) \wedge Isa(g, Guitar)$
(8)	$[_S\ [_N$ Steve$][_{VP}[_V$ plays] $[_{NP}[_{Det}$ the$][_N$ guitar]]]]	$\exists s, p, g IsA(p, Playing) \wedge Player(p, s)$ $\wedge HasName(s, Steve) \wedge Playee(p, g) \wedge Isa(g, Guitar)$

Figure 4.7: Tree-nodes and their syntactic and semantic content

	syntactic rule	semantic attachment
(2)	$NP \rightarrow N$	N
(6)	$NP \rightarrow Det\ N$	N
(7)	$VP \rightarrow V\ NP$	$V(NP)$
(8)	$S \rightarrow NP\ VP$	$VP(NP)$

Figure 4.8: grammatical production of the analysis-tree

to use the process of lambda-reduction, introduced in the previous section, to see that it is in fact possible to derive the meaning-representation of the whole sentence, given in node (8) in figure 4.7 from the "semantic grammar" given in figure 4.8, and the "semantic dictionary", given in nodes (1), (3), (4) and (5) of figure 4.7, and that this derivation really has the same structure as the syntax-tree given in figure 4.3.4.

As we've just shown, the principle of syntax-driven-semantic analysis allows us to guide the semantic derivation along the lines of the parsing-tree. This allows us to handle semantics by simply adding a new field to the sample-grammar we've been using all the time, just as we did, when we augmented the CFG-rules with constraints based on feature-structures.

The S-rule, we've been using so far would have looked like:

$$S \rightarrow NP\ VP \qquad \begin{bmatrix} \text{NUM } \boxed{1} \\ \text{NP } \left[\text{NUM } \boxed{1}\right] \\ \text{VP } \left[\text{NUM } \boxed{1}\right] \end{bmatrix}$$

Remember that it consists of a CFG-rule saying that an NP and V can be replaced by an S, if it is possible to unify both the feature-structure $S.NP$ with the NP's feature structure and the feature-structure $S.VP$ with the VP's feature-structure, and the feature-structure in this case enforcing number-agreement.

In order to handle semantic processing we would now add another data-structure to our grammar rule, leaving the S-rule as something like:

$$S \rightarrow NP\ VP \qquad \begin{bmatrix} \text{NUM } \boxed{1} \\ \text{NP } \left[\text{NUM } \boxed{1}\right] \\ \text{VP } \left[\text{NUM } \boxed{1}\right] \end{bmatrix} \qquad VP(NP)$$

Of course the implementation of such complex grammars isn't usually done in this datastructure, containing a CFG-rule, a FS-constraint and a semantic attachment, but rather, in an integrated way. The ERG, for example, is based solely on feature-structures, because semantic attachments, as well as CFG-rules,

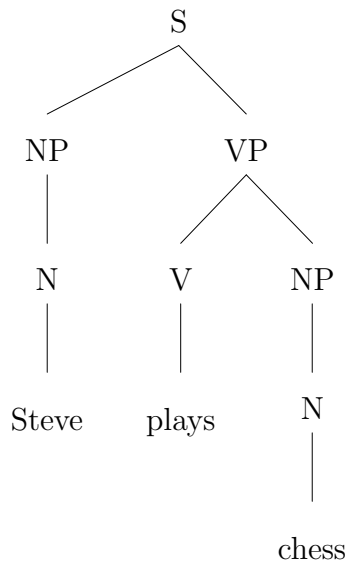


Figure 4.9: A parse-tree for example 4.8

can all be integrated into one powerful feature structure. This formalism was chosen simply for the sake of readability.

4.4 Augmenting a Parser with a Semantic Analyzer

Now we know what a semantic dictionary could look like, how a grammar can be augmented with semantic attachments, and how these partial meanings coming from the dictionary and the grammar can be combined based on the presumptions of syntax-driven semantic analysis using lambda-reductions.

What we want to do in this section is try to get our parser to do this combination-task. Given that we've made the semantic dictionary and the semantic grammar available to our parser, we want it to be able to come up with a complete meaning-representation of a natural-language sentence it parses.

(4.8) Steve plays chess.

We will, therefore, turn to example 4.8 (repeated from example 3.9). Its syntax-tree is given in figure 4.9. This time we'll follow an Earley-parser on its way through the chart, and show how it does semantic analysis.

Figure 4.10 shows the chart created by a simple Earley-parser, when parsing example 4.8 using the simplified grammar from the "syntactic production"-column of figure 4.8.

Very similarly to what we did to the Earley-parser, when we extended it to handle feature structures in section 3.4.2 we have to make two changes to the parser: The first one concerns the representation of states. Additionally to the normal fields of the state, and the associated feature structure we need a field carrying the current semantic content of that state. Therefore a state like

$$NP \rightarrow N \bullet [2, 3]$$

which can be found in entry [1] of $chart[3]$ would now be

$$NP \rightarrow N \bullet [2, 3], \exists g IsA(g, Chess)$$

Feature-structures and backpointers for are left out for readability.

The second change concerns the Completer. Recall that the Completer is that part of the Earley-algorithm that takes care of advancing every state that "is looking for" a symbol that has just been completed, and that a state is considered complete if its $\bullet$ is at the far right of the rule in the state, for example as a result of the Scanner having successfully read an input-token that matches the POS we are looking for. The Completer would, therefore, be the part of the program responsibe for carrying out the lambda-reduction indicated in the grammar-rule of the state that was just completed, carrying over the result of this lambda reduction to the state "looking for" the constituent that was just completed and, by the lambda-reduction, semantically analyzed.

In our example, the first time the Completer is called is for the complete

chart[0]

[0]	$\lambda \rightarrow \bullet S$	[0, 0, 0]	[]
[1]	$S \rightarrow \bullet NP\ VP$	[0, 0, 0]	[]
[2]	$NP \rightarrow \bullet Det\ N$	[0, 0, 0]	[]
[3]	$NP \rightarrow \bullet N$	[0, 0, 0]	[]

chart[1]

[0]	$N \rightarrow steve\bullet$	[0, 1, 1]	[]
[1]	$NP \rightarrow N\bullet$	[0, 1, 1]	[[(1, 0)]]
[2]	$S \rightarrow NP \bullet VP$	[0, 1, 1]	[[(1, 1)]]
[3]	$VP \rightarrow V \bullet NP$	[1, 1, 0]	[]

chart[2]

[0]	$V \rightarrow plays\bullet$	[1, 2, 1]	[]
[1]	$VP \rightarrow V \bullet NP$	[1, 2, 1]	[[(2, 0)]]
[2]	$NP \rightarrow \bullet Det\ N$	[2, 2, 0]	[]
[3]	$NP \rightarrow \bullet N$	[2, 2, 0]	[]

chart[3]

[0]	$N \rightarrow chess\bullet$	[2, 3, 1]	[]
[1]	$NP \rightarrow N\bullet$	[2, 3, 1]	[[(3, 0)]]
[2]	$VP \rightarrow V\ NP\bullet$	[1, 3, 2]	[[(2, 0)], [(3, 1)]]
[3]	$S \rightarrow NP\ VP\bullet$	[0, 3, 2]	[[(1, 1)], [(3, 2)]]
[4]	$\lambda \rightarrow S\bullet$	[0, 3, 1]	[[(3, 3)]]

Figure 4.10: A chart for a run of our Earley parser on example 4.8 (Same as figure with backpointers added)

state [0] in $chart[1]$

$$N \rightarrow steve \bullet [0,1], \exists s HasName(s, Steve)$$

(the value of the semantic attachment was provided by the SCANNER, which simply did a dictionary-lookup in the semantic dictionary, after having recognized the word *steve*).

The completer then finds a state that can be advanced because of this newly completed N-constituent: state [3] in $chart[0]$, which is

$$NP \rightarrow \bullet N[0,0], nil$$

Because advancing the $\bullet$ over the N in this state would create a complete state, the completer can now carry out semantic analysis. Note that it is usually necessary to wait for all constituents used by the current state to be completely parsed and analyzed, before analysis of the current state can be done. In this case, we have all constituents used by the $NP \rightarrow N$-state, namely the N available, so we can do the lambda-reduction which isn't particularly exciting, given that the semantic attachment to the $NP \rightarrow N$-rule is simply N, (as can be seen in figure 4.8), ordering the parser to simply carry over the meaning from the N. Therefore the COMPLETER creates state [1] from $chart[1]$

$$NP \rightarrow N \bullet [0,1], \exists s HasName(s, Steve)$$

And because this state the COMPLETER just created is itself complete, the COMPLETER would now be called for that state. Looking for states in need of an NP, the COMPLETER would now find state [1] from $chart[0]$

$$S \rightarrow \bullet NP\ VP[0,0], nil$$

The $\bullet$ in this state can now be advanced over the NP. In this case the COMPLETER doesn't carry out any semantic action, because the new state wouldn't be complete, therefore creating state [2] in $chart[1]$ as

$$S \rightarrow NP \bullet VP[0,1], nil$$

The PREDICTOR, finding the non-terminal-symbol VP in this state to the left of the $\bullet$, would now take care of adding state [3] to *chart*[1], and we can move on to the next chart.

The SCANNER would now find the token *plays* in the input, and, after looking the word up in the semantic dictionary create state [0] in *chart*[2] as

$$V \rightarrow plays \bullet [1,2], \lambda g, s \exists p IsA(p, Playing) \wedge Player(p, s) \wedge Playee(p, g)$$

This state is complete, and therefore it's now the COMPLETER's turn again, adding state [1] to *chart*[2] without doing semantic analysis, because, as we've just mentioned, this would require all constituents to be completed. In our case it is the NP that we lack information about. The PREDICTOR would then create states [2] and [3] in *chart*[2], and we could again go on to the next chart.

In *chart*[3] the SCANNER would find the token *chess* in the input and therefore create state [0] in *chart*[3] as

$$N \rightarrow chess \bullet [2,3], \exists g IsA(g, Chess)$$

Using this state the COMPLETER can now complete state [3] in *chart*[2]. In this case the resulting state would again be a complete one, therefore semantic analysis is carried out in this case, doing the lambda-reduction from the $NP \rightarrow N$-rule which is simply N, therefore carrying over the semantic attachment from the N to the new state for the NP which is added as state [1] to *chart*[3] as

$$NP \rightarrow N \bullet [2,3], \exists g IsA(g, Chess)$$

This state is exactly what state [3] in *chart*[1] "has been looking for". The COMPLETER when called for the state we just created, would, therefore, try to advance state [3] in *chart*[1], using state [1] in *chart*[3]. Because advancing the $\bullet$ in $VP \rightarrow V \bullet NP$ over the NP would create a completed state, the COMPLETER has to do semantic analysis, which is in this case a bit more interesting, because the semantic attachment to the $VP \rightarrow V\ NP$-rule is $V(NP)$. The NP is what

we just completed, and the V can be found by the lambda-reducer using the backpointers. The new semantic attachment must be the result of the lambda-reduction $V(NP)$, which, fully written, looks like

$$\lambda g, s \exists p \; IsA(p, Playing) \wedge Player(p, s)$$
$$\wedge Playee(p, g) \; (\; < \; \exists g IsA(g, Chess) \; > \;)$$

A technique called *currying* allows us to handle lambda-expressions like this one, where there are two variables marked with lambda, but only one is to be reduced. It simply states, that in such a case the lambda-variable at the left end of the quantification-block is reduced, and the result is itself a lambda-expression, containing the lambda-variables that were not reduced. In our case we would reduce g, and leave s a lambda-variable. This lambda-reduction would therefore evaluate to

$$\lambda s \exists g, p \; IsA(p, Playing) \wedge Player(p, s)$$
$$\wedge Playee(p, g) \wedge IsA(g, Chess)$$

Now the completer can add the new state [2] to $chart[3]$

$VP \rightarrow V \; NP \bullet [1, 3], \lambda s \exists g, p \; IsA(p, Playing) \wedge Player(p, s) \wedge Playee(p, g) \wedge IsA(g, Chess)$

This state can now be used by the COMPLETER to complete state [2] in $chart[1]$. This time the state we want to complete comes from the rule $S \rightarrow NP \; VP$ which has the semantic attachment $VP(NP)$. In our case that resolves to $sem9(sem3)$ which is

$$\lambda s \exists g, p \; IsA(p, Playing) \wedge Player(p, s)$$
$$\wedge Playee(p, g) \wedge IsA(g, Chess) \; (\; < \; \exists s HasName(s, Steve) \; > \;)$$

that evaluates to

$$\exists s, g, p \; IsA(p, Playing) \wedge Player(p, s) \wedge HasName(s, Steve)$$
$$\wedge Playee(p, g) \wedge IsA(g, Chess)$$

QED.

Part II

LISA: A Prototype

Chapter 5

Introduction

In this chapter we will give the reader an overview of what we expect our prototype to do. We will show some of the overall design of the prototype and provide the reader with a first glance into the logics behind the actual problem-solving specific to the Kommisar Klug-problem we will confront our prototype with.

5.1 What we want our Prototype to do

To demonstrate that our prototype really understands natural language, we confront it with a text, written in natural language, such as the one shown in figure 5.1. Then we use a parser and a morphological analyzer, that turn this text into a parseforest. That parseforest and some additional facts and rules are then loaded into a PROLOG-processor, which executes a query.

Figure 5.1 and 5.2 show our problem domain. These two texts also serve as the only source we used for training the dictionary and the grammar, which is why it is highly unlikely, that our prototype would work for any other text.

The problem domain is based on a character, Professor Oberaigner used for teaching boolean logics at HTL-Leonding.

Kommissar Klug is on the verge of solving one of his most complicated cases. He knows that at least **two** of the suspects Peter, Frank, Richard and Steve are involved. If he can prove Richard's guilt, he will also know that Frank and Peter are involved. Proving Steve's participation he can confirm Peter's innocence. "Allright", he mumbles under his breath, "The case isn't closed yet, but we can already arrest someone".

Who is Kommissar Klug talking about?

Figure 5.1: Kommissar Klug's problem (version 1)

Kommissar Klug is on the verge of solving one of his most complicated cases. He knows that at least **three** of the suspects Peter, Frank, Richard and Steve are involved. If he can prove Richard's guilt, he will also know that Frank and Peter are involved. Proving Steve's participation he can confirm Peter's innocence. "Allright", he mumbles under his breath, "The case isn't closed yet, but we can already arrest someone".

Who is Kommissar Klug talking about?

Figure 5.2: Kommissar Klug's problem (version 2)

5.1.1 The Kommissar Klug Problem

Based on the Kommissar-Klug problem, it is easily possible to formalize the statements in boolean logics:

First we define some variables:

p Peter is guilty

q Frank is guilty

r Richard is guilty

s Steve is guilty

Given these variables we can formalize the statement, "If he can prove Richard's guilt, he will also know that Frank and Peter are involved." as $r \rightarrow p \wedge q$, and the statement "Proving Steve's participation he can confirm Peter's innocence." as $s \rightarrow \neg p$. We can use a table such as the one shown in figure 5.3 to solve the problem. The first 4 columns show all the permutations of truth-values for p, q, r and s ("T" indicates the boolean value "true" and "F" indicates "false"). For each of these permutations, we can calculate the value of the two statements. The seventh column is true for all permutations that make the statement, "He knows that at least two [...] are involved.", true. In the last column we build the conjunction of all of these statements, and we find that three permutations are not contradictory:

- Peter, Frank and Richard could have done it together. (In this case, Steve can't be involved).
- Peter and Frank could have done it. Richard isn't neccessarily involved.
- Frank and Steve could have done it.

We find that in each of these permutations Frank is involved, which is why we consider Frank guilty, and our prototype seems to agree on that, as shown in figure 5.4 (note that the others are not guilty, since Kommissar Klug is "in dubio pro reo").

p	q	r	s	$r \rightarrow p \wedge q$	$s \rightarrow \neg p$	at least 2	conjunction
T	T	T	T	T	F	T	F
T	T	T	F	T	T	T	T
T	T	F	T	T	F	T	F
T	T	F	F	T	T	T	T
T	F	T	T	F	F	T	F
T	F	T	F	F	T	T	F
T	F	F	T	T	F	T	F
T	F	F	F	T	T	F	F
F	T	T	T	F	T	T	F
F	T	T	F	F	T	T	F
F	T	F	T	T	T	T	T
F	T	F	F	T	T	F	F
F	F	T	T	F	T	T	F
F	F	T	F	F	T	F	F
F	F	F	T	T	T	F	F
F	F	F	F	T	T	F	F

Figure 5.3: Permutations for boolean truth-values (version 1)

In order to show that our prototype is appliable to a bigger problem domain than only outputting `X = semFRANK`, we consider another problem, which is shown in figure 5.2. It's the same problem, but this time we know that three of the suspects must have been involved, which leaves only one permutation valid, which is the one where Peter, Frank and Richard are involved, and, again, our prototype agrees on that, as shown in figure 5.5.

5.1.2 Some DOs and DON'Ts

What we've just shown doesn't, by itself, have much to say. Consider figure 5.6, which took me about three minutes to write, which reacts very similarly to the real LISA-prototype, which took me almost a year to write.

The big difference is that the program outlined in figure 5.6 can't really be said to *understand* the problem, because in this program, the programmer models his own knowledge of the particular problem and the language, rather than provide the program with a means to autonomously understand the problem stated in a

```
KOMMISSAR KLUG IS ON THE VERGE OF SOLVING ONE OF HIS MOST COMPLICATED CASES .                                SUCCESS!
HE KNOWS THAT AT LEAST TWO OF THE SUSPECTS PETER FRANK RICHARD AND STEVE ARE INVOLVED .                      SUCCESS!
IF HE CAN PROVE RICHARDS GUILT HE WILL ALSO KNOW THAT FRANK AND PETER ARE INVOLVED .                         SUCCESS!
PROVING STEVES PARTICIPATION HE CAN CONFIRM PETERS INNOCENCE .                                               SUCCESS!
" ALLRIGHT " HE MUMBLES UNDER HIS BREATH " THE CASE ISNT CLOSED YET BUT WE CAN ALREADY ARREST SOMEONE " .    SUCCESS!
WHO IS KOMMISSAR KLUG TALKING ABOUT ?                                                                        SUCCESS!
%  dictionary compiled 0.00 sec, 1,372 bytes
%  parseforestpl compiled 0.07 sec, 164,440 bytes
%  lang compiled 0.00 sec, 9,076 bytes
%  probdom compiled 0.00 sec, 3,336 bytes
% ./test.pl compiled 0.08 sec, 178,380 bytes
Welcome to SWI-Prolog (Version 5.0.8)
Copyright (c) 1990-2002 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- guilty(X).

X = semFRANK ;

No
?-
```

Figure 5.4: Running the program on Kommissar Klug's problem (version 1)

```
KOMMISSAR KLUG IS ON THE VERGE OF SOLVING ONE OF HIS MOST COMPLICATED CASES .                                SUCCESS!
HE KNOWS THAT AT LEAST THREE OF THE SUSPECTS PETER FRANK RICHARD AND STEVE ARE INVOLVED .                    SUCCESS!
IF HE CAN PROVE RICHARDS GUILT HE WILL ALSO KNOW THAT FRANK AND PETER ARE INVOLVED .                         SUCCESS!
PROVING STEVES PARTICIPATION HE CAN CONFIRM PETERS INNOCENCE .                                               SUCCESS!
" ALLRIGHT " HE MUMBLES UNDER HIS BREATH " THE CASE ISNT CLOSED YET BUT WE CAN ALREADY ARREST SOMEONE " .    SUCCESS!
WHO IS KOMMISSAR KLUG TALKING ABOUT ?                                                                        SUCCESS!
%  dictionary compiled 0.00 sec, 1,372 bytes
%  parseforestpl compiled 0.08 sec, 164,440 bytes
%  lang compiled 0.01 sec, 9,076 bytes
%  probdom compiled 0.00 sec, 3,336 bytes
% ./test.pl compiled 0.10 sec, 178,380 bytes
Welcome to SWI-Prolog (Version 5.0.8)
Copyright (c) 1990-2002 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- guilty(X).

X = semPETER ;

X = semFRANK ;

X = semRICHARD ;

No
?-
```

Figure 5.5: Running the program on Kommissar Klug's problem (version 2)

```
#!/bin/sh

if cat test.txt | grep -e two > /dev/null
then
  echo "X = semFRANK"
fi

if cat test.txt | grep -e three > /dev/null
then
  echo "X = semPETER"
  echo "X = semFRANK"
  echo "X = semRICHARD"
fi
```

Figure 5.6: A big DON'T

language.

A program, which is able to understand a problem, when stated in a language is allowed to use knowledge

- about the language
- about the problem *domain*

as a basis to deduce its knowledge of the particular problem from it, but is not allowed to use any knowledge of the particular problem stated by the programmer in an explicit way.

5.2 Modules and how they Interact

Figure 5.7 is a diagram showing the most important modules and how they interact. The overall goal is that the prototype should be able to execute the query `guilty(X)` and come up with the solution `X = semFRANK`. This should be done using knowledge stated in English.

There are two programs involved in doing so. One is a PROLOG-program. We simply load facts and rules making up the semantic analyzer, as well as facts and

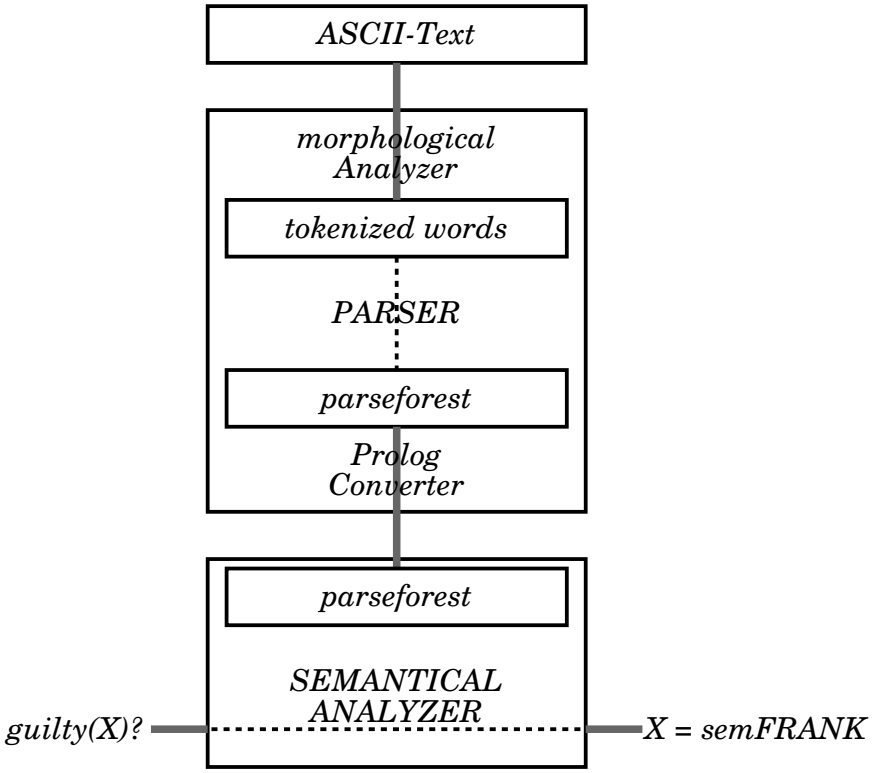


Figure 5.7: The most important modules

rules making up the parse-forest for the English text into a PROLOG-interpreter, but how do we obtain these facts and rules that make up the parse-forest?

This task is done by the second program, which is written in C. It is an Earley-parser. It opens the ASCII-Text and reads one word after the other, performing some filtering that should be done prior to morphological analysis (splitting the text up into tokens separated by whitespace, bringing the strings into an uppercase representation, handling special characters such as punctuation, etc.). These words are then run through the morphological analyzer. As far as the parser is concerned, this is done by simply calling a function `run`.

Its implementation can be found in a separate module. This module is created, not by the C-Compiler, but by a compiler we wrote especially for the LISA-project. This compiler is capable of turning an ASCII or XML-representation of an FST into machine-executable code.

These tokenized and morphologically analyzed words can then be used by an Earley-parser to build up an internal representation of a parseforest. The next task of our C-Program is then to bring its internal representation into a representation compatible with PROLOG, by writing a PROLOG-file containing facts and rules that make up the parseforest.

5.3 The Kommissar Klug Problem Domain

5.3.1 Knowledge from the Text

In this section we will propose some predicates that we would have to derive from the text.

The first predicate is `semsuspect(X)`. It is derivable for any `X` that is mentioned in the text as a suspect. In our case this would be:

```
semsuspect(peter).
semsuspect(frank).
```

```
semsuspect(richard).
semsuspect(steve).
```

`seminnocentif(A,B)` is derivable if the text says something like, "A is innocent if B is guilty". In our case, this would be

```
seminnocentif(peter,steve).
```

`semguiltyif(A,B)` is used analogously if the text says something like, "A is guilty if B is guilty":

```
semguiltyif(peter,richard).
semguiltyif(frank,richard).
```

`semsuspectnumber(N)` is derivable if the text says something like, "At least N suspects were involved".

```
semsuspectnumber(two).
```

Note that we would first have to tell the system what "two" means. We can use something like a semantic dictionary, telling the system the most important fact about "two", that it is the number that comes after "one":

```
s(two,one).
s(three,two).
```

Here "s" stands for the "successor"-relation.

5.3.2 Knowledge about the Problem Domain

Given this textual knowledge we can start to give the system some way of dealing with it. One of the most important rules is that if Y is innocent, assuming that X is guilty, then X is innocent assuming that Y is guilty. This seems to make sense, but it is also derivable from boolean logics:

$$b \rightarrow \neg a$$

$$\neg b \vee \neg a$$
$$\neg a \vee \neg b$$
$$a \rightarrow \neg b$$

We can formalize this by

```
innocentif(A,B):-seminnocentif(B,A).
innocentif(A,B):-seminnocentif(A,B).
```

The next relation states that, if we assume that R is guilty, and are therefore allowed to conclude that P is guilty, from the `semguiltyif`-relation, and if we assume that S is guilty, and are therefore allowed to conclude that P is innocent from the `innocentif`-relation, then we also know that, assuming that S is guilty, we can conclude that R must be innocent.

```
innocentif(R,S):-semguiltyif(P,R),innocentif(P,S).
```

This can also be shown on a formal basis:

$$(r \rightarrow p) \wedge (s \rightarrow \neg p)$$
$$(r \rightarrow p) \wedge (p \rightarrow \neg s)$$
$$(r \rightarrow \neg s)$$
$$(s \rightarrow \neg r)$$

We can formalize this by

```
innocentif(R,S):-semguiltyif(P,R),innocentif(P,S).
```

5.3.3 Further Considerations

In addition to to these rules, we need rules dealing with permutations of suspects, taking into account the number of suspects involved etc, and we need rules deriving the exemplaric facts given in the section about textual rules from

the parseforest. Of course, we don't state the facts we showed as examples, because this would mean modelling knowledge about the specific problem explicitly. These facts are derived from a simple algorithm that looks for certain patterns of parsetrees in the parseforest we were given. We won't describe these parts of the semantic processor in greater detail since they mainly consist of technicalities.

Chapter 6

FST-Tools

A Few Words on this Document

While reading this document it is important to keep a few things in mind: First of all, this document was never intended to be a scientific text or a textbook on FST-processing. I have to admit that, except for a few pages in an introductory textbook, I've never read anything about the subject. Also the algorithms presented here are merely a "quick 'n dirty" hack. This document was created purely by the need for a working system processing FSTs in the LISA-project.

If this document is not a scientific text, then what is it? It is Sourcecode-documentation, in the style of Donald Knuth's "Literate Programming"-paradigm. That is why this document is mainly targeting myself, as a reader, should I be trying to "get into" that code again, in a few years' time. It is also targeted towards other developers who want to work with this sourcecode, and to people who are seeking a technically detailed description of an actual implementation of an FST-compiler. In fact it is so detailled, it can actually be weaved, and run - the magic of literate programming. What the reader certainly shouldn't be doing is take it as a reference or an academic resource of any kind.

I hope that, in this light, one might also excuse some of the stylistic and

didactic flaws in this document.

6.1 An Environment for Handling FSTs

In order to effectively handle FSTs in the LISA-project it is neccessary to create a small toolkit, containing several classes, each of which handles a different task related to FST-processing.

Although there already existed some toolkits for handling FSTs, at the time of this writing, I couldn't find one that was both, affordable, and suited for the needs of this project, this is why I decided to quickly hack a set of small python classes doing that work.

Python seemed to me a very well suited programming-language for handling that task, since it is easy to learn and well documented, and since it comes with a huge set of useful modules, and defines some excellent data-types that make code more readable - that is why I use Python almost every time I do rapid prototyping.

The main-program of this toolkit is a compiler, converting a description of a transducer into assembly-code.

The input file takes the form of an XML-definition of the transducer, (conforming to `fst.dtd`). `test1.xml` is an example of such an input-file. The output-file can be assembled by GNU-Assembler. The code is targeted towards the Intel i386-Processor and follows C-call-conventions.

When assembled the object exports a symbol like `int run(char *in, unsigned long int *out)` that, runs the FST. All resources that are needed are compiled into the program, and the compiled code doesn't involve any interpretive code, only pure machine-code operating without any external resources.

This design makes it possible to achieve very high performance when running the transducers, while preserving maintainability of the finite-state-transducers and interoperability with the C-program that uses it.

6.1.1 Representing an FST

XML-Representation

This toolkit actually uses two different types of representations for FSTs. The first one is a machine-readable format, reflecting the internal representation of that data-structure in python. (We are going to hear about that soon). In this section I want to focus on the second one, which was intended to be both machine-readable and human-readable.

I decided to represent these FSTs using XML-files, conforming to `fst.dtd`. The choice was XML since it is easily readable by a human, and because of the available libraries, that make their handling easier.

"fst.dtd" 130 ≡

```
<!ELEMENT transducer (finstate|state)+>
<!ELEMENT state (transition)+>
<!ELEMENT finstate EMPTY>
<!ELEMENT transition EMPTY>
<!ELEMENT transduction EMPTY>
<!ATTLIST transducer
  fin IDREF #IMPLIED
  ini IDREF #REQUIRED>
<!ATTLIST state
  id ID #REQUIRED>
<!ATTLIST finstate
  id ID #REQUIRED>
<!ATTLIST transition
  signal CDATA #REQUIRED
  output CDATA #IMPLIED
  target IDREF #REQUIRED>
<!ATTLIST transduction
  target IDREF #REQUIRED
  txt CDATA #REQUIRED>
```

◇

The root-element is, of course, the transducer, which has both finstates (final states) and states. There are two attributes: The required "`ini`"-attribute refers to the initial state in the transducer. Optionally the "`fin`"-attribute can be used to define the final state, if this transducer is designed to have only one final state.

Finstates and states must have a unique `id`. While a finstate is by definition empty, and only serves as a "declaration" of a valid final state, a normal state has one or more transitions or transductions.

A transition is also empty by definition. Nevertheless some attributes are

required, one of which is `signal` containing a string-representation of the signal, that fires the transition. The other one is the `target` identifying the state, this transition leads to. If the optional attribute `output` is given, then this signal is sent to the output-"tape" of the transducer.

It is important to notice, that the output-"tape" of a transducer in the LISA project is by definition an array of double-words. (In C-syntax that would be an `unsigned long int[]`), while the input-tape is a byte-array. (That is a `char[]` in C).

Although output-symbols have to be a full double-word, input symbols are merely considered partial strings of the input. Input symbols can have lengths of 1, 2, 3 or 4 bytes. The limitation is four, because that is the biggest quantity an i386-Processor can effectively compare (That is with one `CMPL`-statement).

There are also some special-purpose signals. The first one is the `&`-signal which is the default transition, the transition that fires when no other one in this state does. If no default transition is specified a transition to a reject-state is implied.

`fstcomp-dta2.txt` gives some sample strings, and associated numbers:

`"fstcomp-dta2.txt"` 131 ≡

```
1001 xabcde
1002 xabfgh
1003 nnkki
1004 aabcde
1005 aabfgh
1006 xarki
```

◇

A valid FST, representing these mappings would be `fstcomp-dta2.xml`

"fstcomp-dta2.xml" 132 ≡

```
<?xml version="1.0"?>
<!DOCTYPE transducer SYSTEM "file:///massdata/neword/src/lisa/tst/dta/fst.dtd">


<transducer ini="q0" fin="fin">
  <finstate id="fin"/>
  <state id="q0">
    <transition target="q5" signal="xa"/>
    <transition target="q11" signal="nnkk" output="1003"/>
    <transition target="q18" signal="aab"/>
  </state>
  <state id="q4">
    <transition target="q10" signal="cde" output="1001"/>
    <transition target="q10" signal="fgh" output="1002"/>
  </state>
  <state id="q5">
    <transition target="q10" signal="rki" output="1006"/>
    <transition target="q4" signal="b"/>
  </state>
  <state id="q10">
    <transition target="fin" signal="&"/>
  </state>
  <state id="q11">
    <transition target="q10" signal="i"/>
  </state>
  <state id="q18">
    <transition target="q10" signal="cde" output="1004"/>
    <transition target="q10" signal="fgh" output="1005"/>
  </state>
</transducer>
```

◇

We haven't yet mentioned the texttttransductions, from the above DTD. Instead of a transition, one can use a transduction to insert an external FST into the transducer. The external FST can be another XML-File (not yet implemented) or a TXT-File with simple string-mappings.

"fstcomp-dta1.xml" 133 ≡

```
<?xml version="1.0"?>
<!DOCTYPE transducer SYSTEM "file:///massdata/neword/src/lisa/tst/dta/fst.dtd">

<transducer ini="q0" fin="fin">
  <finstate id="fin"/>
  <state id="q0">
    <transition target="q1" signal="x"/>
    <transition target="q1" signal="y"/>
  </state>
  <state id="q1">
    <transduction target="q2" txt="tst/dta/fstcomp-dta2.txt"/>
    <transduction target="q3" txt="tst/dta/fstcomp-dta3.txt"/>
  </state>
  <state id="q2">
    <transition target="fin" signal="ab" output="42"/>
    <transition target="fin" signal="cd"/>
  </state>
  <state id="q3">
    <transition target="fin" signal="ef" output="43"/>
    <transition target="fin" signal="gh"/>
  </state>
</transducer>
```

◇

"fstcomp-dta3.txt" 134 ≡

```
1007 this
1008 is
1009 a
1010 test
```

◇

`fstcomp-dta1.xml` shows how this feature can be used. To get to state q_1 the transducer has to either match `x` or `y`. If the transducer `fstcomp-dta2.txt` accepts the input it goes to state q_2, if the transducer `fstcomp-dta3.txt` accepts it, it goes to state q_3 and so on.

The transducers `fstcomp-dta2.txt` and `fstcomp-dta3.txt` are not externally run, they are rather cascaded into the current transducer at compile-time. It is part the program's job to read the text-file, convert it to a transducer, and insert it into the current FST.

Internal Representation

Of course these XML-files are a bit inconvenient to work with in a python-script. (It would, perhaps, be technically possible to directly access DOM-Objects during processing, but it would only artificially complicate things).

Instead we will use an array that will usually be called `self.fst` in the scripts. The `testdta1`-array below is what such an array would look like for our example-data from `fstcomp-dta2.txt`.

⟨ fstcompdta2 135 ⟩ ≡

```
fstcompdta2[0] = {'xa': [5, 0], 'nnkk': [11, 1003], 'aab': [18, 0]}
fstcompdta2[4] = {'cde': [10, 1001], 'fgh': [10, 1002]}
fstcompdta2[5] = {'rki': [10, 1006], 'b': [4, 0]}
fstcompdta2[10] = {'&': [-1, 0]}
fstcompdta2[11] = {'i': [10, 0]}
fstcompdta2[18] = {'cde': [10, 1004], 'fgh': [10, 1005]}
```

◇

Macro never referenced.

`fstcompdta2[]` holds all possible states. Each state is a dictionary mapping the signal to its transition.

Each transition is a tuple of the form `(target, output)`. A `target` is either an index, pointing to another state in `testdta1[]`, or `-1` if the transition points to the final state. `output` is the output, associated with that transition or `0`, if there is no output associated with this transition.

6.2 Design of the Toolkit

The FST-compiler uses an object-oriented design. This is what the main program looks like, and how the classes facilitate its work:

"fstcompxml.py" 136 ≡

```
#!/usr/bin/python

from optimizer import optimizer;
from xmlloader import xmlloader;
from compiler import compiler;

import sys;

if len(sys.argv)!=3:
  print "usage: fstcompile.py <funcname> <xmlfile>"
else:
  fs3=[{}];
  xmlloader(fs3).load(sys.argv[2]);
  optimizer(fs3).optimize();
  compiler(fs3).compileit(sys.stdout);
```

◇

"fstcomptxt.py" 137 ≡

```
#!/usr/bin/python

from optimizer import optimizer;
from txtloader import txtloader;
from compiler import compiler;

import sys;

if len(sys.argv)!=3:
  print "usage: fstcompile.py <funcname> <txtfile>"
else:
  fs3=[{}];
  txtloader(fs3).load(sys.argv[2]);
  optimizer(fs3).optimize();
  compiler(fs3).compileit(sys.stdout);
```

◇

The following classes operate in the background, to make this kind of usage possible:

loader is the base-class for all classes, that are somehow capable of loading an FST from an external resource.

xmlloader is derived from it. As the name suggests, its job is to load an FST from an XML-file.

txtloader is the second loader in the toolkit. It can load an FST by reading a file such as `testdta1.txt` and convert it to an FST.

opunit stands for OPerational UNIT, and it implements the core algorithms, such as cascading or concatenating FSTs.

optimizer enters the scene, when the FST is ready to be compiled. The optimizer can apply some optimizations, such as concatenating linear paths through the FST to longer signals or joining redundant subtrees.

compiler is the class that can convert an FST to GNU-Assembler.

debugger is a set of functions that were useful when developing this. These functions aren't usually called in a productive environment. They implement things like printing an FST, or traversing the FST to list all possible paths through it, etc. This class isn't described in this document.

This is quite a big number of classes for a system having less than 500 lines of code. I chose this object-oriented design, anyway to maintain a greater degree of readability.

6.3 The Compiler

It is the job of this class to read the internal representation of the FST after it's loaded, and create an ASCII-File that can be assembled by the GNU-Assembler. The target platform is i386, although optimizations for i486 were applied.

This is the basic outline of the class:

"compiler.py" 139 ≡

```
import sys;
import string;

class compiler:
  def __init__(self,fst):
    self.fst=fst;
    if len(sys.argv)==2:
      self.funcname=sys.argv[1];
    else:
      self.funcname='run';

  ⟨ define the littleend-function 143 ⟩

  ⟨ define the dotransition-function 149 ⟩

  def compileit(self,out):
    ⟨ initialize the transducer 140 ⟩

    curstate=0;
    while curstate<len(self.fst):
      if len(self.fst[curstate])>0:
        ⟨ initialize the new state 141b ⟩
        ⟨ remember the transitions 142a ⟩
        ⟨ assemble the state 142b ⟩
      curstate=curstate+1;

    ⟨ finalize the transducer 150b ⟩
```

◇

The argument that is read from the command-line, (`funcname`), is the name of the

symbol, that will get exported by the output-file. For example `fsacompile.py test input.xml` will create an object, that exports a symbol

`int test(char* in, unsigned long int* out)`,

`in` being the input-tape of the FST, and `out` being the FST's output-tape (that is the input of the caller then). The ability to specify the symbol-name is particularly useful, when more transducers are linked against the same C-program.

Recall that all output, done by this compiler, should conform to C-call-conventions, which is why I use C-syntax to denote data-types and function-declarations.

The default function-name is `run`. `out` is the file-handler the output is written to.

6.3.1 Initializing the FST

⟨ initialize the transducer 140 ⟩ ≡

```
self.curl=0;


print """⟨ initial chunk of assembly-code 141a ⟩""" % \
  {'func':self.funcname};◇
```

Macro referenced in 139.

All we have to do is initialize the `self.curl`-variable, which is a counter for naming intermediate labels, and print an initial chunk of assembly-code.

This code basically just exports the symbol (by the name that was specified on the command-line), reads the first parameter (that is `char* in`) into `eax` and the second one (that is `unsigned long int* out`) into `ebx`, and then jumps to the initial state. It also builds up the stack-frame for this function.

⟨ initial chunk of assembly-code 141a ⟩ ≡

```
.text
        .align 16

.globl %(func)s
        .type %(func)s, @function

%(func)s:
        pushl %%ebp
        movl %%esp, %%ebp
        push %%ebx
        push %%ecx
        push %%edx
        movl 8(%%ebp),%%eax
        movl 12(%%ebp),%%ebx
        movl %%ebx,%%ecx
        jmp q0◇
```

Macro referenced in 140.

6.3.2 Collecting Data About the State

All we have to do is initialize the `trns`-variable.

⟨ initialize the new state 141b ⟩ ≡

```
trns={0:[], 1:[], 2:[], 3:[], 4:[]};◇
```

Macro referenced in 139.

It is a dictionary of five arrays, each holding the transitions that have input symbols of the length indicated by the keys in the dictionary. The zeroth holds the transition with signal `&`. Recall that this is the default transition that fires when no other transition does, and that a transition to the `reject`-state is implied, if there is no default transition.

That way we'll find all transitions with input-symbols of length `n` in `trans[n]`, when we need them.

And this is how it's filled:

⟨ remember the transitions 142a ⟩ ≡

```
for key in self.fst[curstate].keys():
  idx=0;
  if key!='&':
    idx=len(key);
  trns[idx].append(\
    (self.fst[curstate][key][0],key,self.fst[curstate][key][1])\
  );◇
```

Macro referenced in 139.

What we do is put together a tuple holding `(target,signal,output)`, and append it to the appropriate array in the `trns`-dictionary.

6.3.3 Assembling States

After initializing the data-structures for handling a state, and collecting data about the state's transitions, all we have to do is print assembly-code for that state.

⟨ assemble the state 142b ⟩ ≡

⟨ read the next signal from memory 144 ⟩

```
prev=0;
```

⟨ handle signals of length one 145a ⟩

⟨ handle signals of length two 145b ⟩

⟨ handle signals of length four 146 ⟩

⟨ handle signals of length three 147 ⟩

⟨ handle the default transition 148 ⟩◇

Macro referenced in 139.

In this code-segment we also initialize the prev-pointer. We'll hear about the details of this pointer in the next section.

When we define how to handle the signals, we will need a small helper function, converting a signal in its string-representation, to a hexadecimal littleendian number. (as will be found in a 32-bit-register, after `mov`-ing a 32-bit-quantity from memory). That basically means reversing the bytes.

⟨ define the littleend-function 143 ⟩ ≡

```
def littleend(self,val):
  num=0;
  k=len(val)-1;
  while k>=0:
    num*=256;
    num+=ord(val[k]);
    k=k-1;

  fmtst='0x%%%dx' % (len(val)*2);
  hex=fmtst % (num);

  return string.replace(hex,' ','0');◇
```

Macro referenced in 139.

That's quite simple. All we have to do is add up the ascii-values of the characters, as we traverse the string backwards, and multiply the value with 256 (that is shift left by 1 byte) before each iteration. Then we use python's formatting capabilities to return a hexadecimal string, instead of the number.

6.3.4 Examining the Signal

Reading a signal from memory is actually quite trivial. Depending on the maximum signal-length we're going to handle in this state, we either `movl`, `mov` or

`movb` the next signal into the `edx`-register (respectively `dx` or `dl`). Note that it takes the Intel 80486, and newer i386-processors, exactly one tact's time, to do any of these operations, so it would make no sense to first only `movb`, the first byte, then try to match that, then `movb`, the second byte, in case we need that.

Note what this means for the input buffer: it must be three bytes longer, than the actual string-length of the input, since it might happen, that we read 4 bytes from memory, because this state has transitions for 4-byte-signals, while the current one is the last byte available in the input-buffer.

⟨ read the next signal from memory 144 ⟩ ≡

```
if len(trns[4])>0 or len(trns[3])>0:
  print 'q%d: movl (%%eax),%%edx' % (curstate);
elif len(trns[2])>0:
  print 'q%d: mov (%%eax),%%dx' % (curstate);
elif len(trns[1])>0:
  print 'q%d: movb (%%eax),%%dl' % (curstate);
elif len(trns[0])>0:
  print 'q%d:' % (curstate);◇
```

Macro referenced in 142b.

Now we have the data ready in the `edx`-register. We already mentioned that i386 uses little-endian memory-operations, that means, we have the first byte of the signal at the least significant byte in the register (that would be `al`), and the last byte in the most significant byte of the register.

⟨ handle signals of length one 145a ⟩ ≡

```
if len(trns[1])>0:
  prev=1;
  print 'incl %eax';

  for trans in trns[1]:
    print 'cmpb $%s,%%dl' % (self.littleend(trans[1]));
    self.dotransition(trans);◇
```

Macro referenced in 142b.

First thing we do is we `incl %eax`. The policy is that at any time in further processing we have to be able, to simply jump to any state, and leave `eax` pointing to the next signal we have to examine. Then, for each transition in that state, we simply compare the signal, and call the function `dotransition` to print assembly-code handling the transition, if the compare was successful (equal, that is).

⟨ handle signals of length two 145b ⟩ ≡

```
if len(trns[2])>0:
  if prev==0:
    print 'addl $2, %eax';
  elif prev==1:
    print 'incl %eax';
  prev=2;

for trans in trns[2]:
  print 'cmp $%s,%%dx' % (self.littleend(trans[1]));
  self.dotransition(trans);◇
```

Macro referenced in 142b.

Note that we maintain the `prev`-pointer, giving information about the signal length, we examined last, because that information is necessary to appropriately

set `eax`. Apart from that this code-snippet is quite the same as the above one.

⟨ handle signals of length four 146 ⟩ ≡

```
if len(trns[4])>0:
  if prev==0:
    print 'addl $4,%eax';
  elif prev==1:
    print 'addl $3,%eax';
  elif prev==2:
    print 'addl $2,%eax';
  prev=3;

for trans in trns[4]:
  print ' cmpl $%s,%%edx' % (self.littleend(trans[1]));
  self.dotransition(trans);◇
```

Macro referenced in 142b.

And the same thing again for signal-length 4. Note that we handle length 4 before length 3, because comparing a signal of length 3 requires us to destroy the data in the most significant byte, since there is no way to directly compare a 3-byte-quantity.

⟨ handle signals of length three 147 ⟩ ≡

```
if len(trns[3])>0:
  if prev==0:
    print 'addl $3,%eax';
  elif prev==1:
    print 'addl $2,%eax';
  elif prev==2:
    print 'incl %eax';
  elif prev==3:
    print 'decl %eax';
  print 'andl $0x00FFFFFF,%edx';
  prev=4;

for trans in trns[3]:
  print 'cmpl $%s,%%edx' % (self.littleend(trans[1]+'\0'));
  self.dotransition(trans);◇
```

Macro referenced in 142b.

And this is how it works: all we basically have to do is to `and` it against a mask, setting the first byte to zero, and all other bits to `1`, which leaves the most significant byte zero, and all others untouched. Then we append zero to the signal, we want to compare it to, convert it to littleendian notation, and compare the whole double-word.

⟨ handle the default transition 148 ⟩ ≡

```
if len(trns[0])==0:
  print 'jmp reject';
else:
  if prev==1:
    print 'decl %eax';
  elif prev==2:
    print 'subl $2,%eax';
  elif prev==3:
    print 'subl $4,%eax';
  elif prev==4:
    print 'subl $3,%eax';

  self.dotransition(trns[0][0]);◇
```

Macro referenced in 142b.

6.3.5 Handling Transistions

Up to this point we simply called a `dotransition`-function, to handle the transition, in case we leave the `equal` flag set.

This is what this function looks like:

⟨ define the dotransition-function 149 ⟩ ≡

```
def dotransition(self, targ):
  if targ[0]==-1:
    target='match';
  else:
    target='q%d' % targ[0];

  # if we are not doing any output:
  if targ[2]==0:
    if targ[1]!='&':
      print 'je %s' % (target);
    else:
      print 'jmp %s' % (target);

  # if we are doing output:
  else:
    ⟨ handle a transition with output 150a ⟩◇
```

Macro referenced in 139.

The approach is quite straightforward: recode any reference to the final state `-1`, to `match`, and call all other targets `q<index>`. Then we can simply `je` to our target. If the transition we are currently handling is the default-transition we don't need the conditional jump, printing an unconditional one instead.

But we can only handle transitions, that are not doing any output this way, otherwise it gets a bit trickier: First thing we do is jump to the end of the block, we are talking about, in case the `equal`-flag isn't set. If we pass that point, or if we are handling the default-transition we can savely do the output by simply `movl`-ing it to the memory-region pointed to by `ebx` (recall that we read the second argument to our function into that register, earlier), add `4` to the register, to leave it pointing to the next memory-cell that's ready for writing output to,

and `jmp` to the target.

⟨ handle a transition with output 150a ⟩ ≡

```
if targ[1]!='&':
  print 'jne i%d' % (self.curl);
print 'movl $%s,(%%ebx)' % (targ[2]);
print 'addl $4,%ebx';
print 'jmp %s' % (target);
if targ[1]!='&':
  print 'i%d:' % (self.curl);
self.curl=self.curl+1;◇
```

Macro referenced in 149.

6.3.6 Finalizing the Transducer

All we have to do is

⟨ finalize the transducer 150b ⟩ ≡

```
print """⟨ final chunk of assembly-code 151 ⟩"""◇
```

Macro referenced in 139.

And this is what it looks like:

⟨ final chunk of assembly-code 151 ⟩ ≡

```
reject: xorl %eax,%eax
        jmp done

match:  movb (%eax),%dl
        cmpb $0,%dl
        jne reject

accept: subl %ecx,%ebx
        shr $2,%ebx
        movl %ebx,%eax

done:   pop %edx
        pop %ecx
        pop %ebx

        movl %ebp, %esp
        popl %ebp
        ret◇
```

Macro referenced in 150b.

That basically just defines the standard-states `match`, `reject` and `accept`. `match`, first has a look at the next signal. If it is not equal to zero, we reject the input, else we accept it. (We do that because, we want strings to be zero-terminated.) `reject` leaves the `eax` register `0` (which means the function has return code `FALSE`), and the `accept`-state computes the number of output double-words, by subtracting `ecx` from `ebx` and dividing by `4`. (Note: When entering the procedure we remembered the initial value of `ebx` in `ecx`). Last thing we do is return from the function, that means reset the registers and tear down the stack-frame.

6.4 The XML-Loader

The XML-Loader is one of the central components, that are visible to the fstcompiler-main-program. It is used to read an XML-File, and all external FSTs that are referenced by it, and integrate everything into one transducer, leaving it ready for further processing by other classes of this toolkit.

I already mentioned, that I chose XML as a representation for FSTs, because it can easily be handled, using standard-libraries. The standard-library I am using in this prototype is python's SAX-API. It is an event-oriented framework that automates the parsing, and leaves the processing up to the programmer.

This is what the class we are talking about looks like:

"xmlloader.py" 153 ≡

```
import sys;
import string;
from xml.sax import ContentHandler;
from xml.sax import saxutils;
from xml.sax import make_parser;
from xml.sax.handler import feature_namespaces;

from misc import loader;
from txtloader import txtloader;
from opunit import opunit;

class xmlloader(saxutils.DefaultHandler,loader):
  def __init__(self,fst):
    self.fst=fst;
    self.fst.remove({});

  def load(self, filename):
    f=open(filename);
    parser = make_parser();
    parser.setFeature(feature_namespaces,0);
    parser.setContentHandler(self);
    parser.parse(f);
    f.close();

  def startElement(self, name, attrs):
    if name=='transducer':
      ⟨start the transducer 154a⟩
    if name=='state':
      ⟨start the state 154b⟩
    if name in ['transition', 'transduction']:
      ⟨start the transition or transduction 155b⟩

  def endElement(self, name):
    if name=='state':
      ⟨finish the state 155a⟩
```

This code doesn't really do anything, it's merely a skeleton, that we will fill, in this section. The `load`-funciton registers the xmlloader class as content-handler for the XML-file, and calls the `parse`-function, that starts the parsing process.

Whenever the SAX-parser for example comes across an opening `transducer`-tag, it calls `startElement`, and sets the `name`-parameter to `transducer`. When it finds the closing `transducer`-tag, it calls `endElement`, and sets `name` to `transducer`.

6.4.1 Initializing the Transducer

⟨ start the transducer 154a ⟩ ≡

```
self.statenos={};
self.nextstateno=0;◇
```

Macro referenced in 153.

All we have to do, to initialize the transducer, is to initialize the `statenos`-dictionary, that maps the XML-tag's IDs, to indices of the `self.fst`-array, and set `self.nextstateno` to 0. This is where we keep track of the index, that the next state, we come across will get.

6.4.2 Handling States

This is how we initialize a new state:

⟨ start the state 154b ⟩ ≡

```
self.curst={};
self.statenos[attrs.get('id').encode('ascii','ignore')]=self.nextstateno;
self.nextstateno=self.nextstateno+1;◇
```

Macro referenced in 153.

First we initialize the `self.curst`-dictionary, where we will put together the current state, so that we can simply append it to `self.fst`, when we're done. Then we add the current ID, and stateno to the `statenos`-dictionary.

⟨ finish the state 155a ⟩ ≡

```
self.fst.append(self.curst);◇
```

Macro referenced in 153.

6.4.3 Handling Transitions and Transductions

This is quite simple. The following outline shouldn't need any further explanation.

⟨ start the transition or transduction 155b ⟩ ≡

```
⟨ read the output into op 155c ⟩
⟨ read the target into trg 155d ⟩
⟨ read the signal into sig 156a ⟩
self.curst[sig]=[trg,op];◇
```

Macro referenced in 153.

We assume that `op=0`. If the key `output` is specified, we set `op` to its value.

⟨ read the output into op 155c ⟩ ≡

```
op=0;
if attrs.has_key('output'):
  op=string.atoi(attrs.get('output','0').encode('ascii','ignore'));◇
```

Macro referenced in 155b.

We set `target` to the `target`-attribute's value. Note that this attribute is specified as mandatory in the DTD, so we don't have to handle the default-case in which no such attribute is specified.

⟨ read the target into trg 155d ⟩ ≡

```
trg=attrs.get('target').encode('ascii','ignore');◇
```

Macro referenced in 155b.

If we are handling a transition, we simply set `sig` to the `signal`-attribute's value. If we are handling a transduction we insert a dummy-signal, that will get replaced

later. This dummy signal takes the form `#T#<filename>`, if we are talking about a text-file as external FST. (That's the only external resource, that is implemented so far).

⟨ read the signal into sig 156a ⟩ ≡

```
if name=='transition':
  sig=attrs.get('signal').encode('ascii','ignore');
else:
  sig='#T#'+attrs.get('txt').encode('ascii','ignore');◇
```

Macro referenced in 155b.

6.4.4 Putting Together the Transducer

What we have at this point in processing is a data-structure, that could almost be a transducer, with two differences: signals can be pseudo-signals like `#T#testdta1.txt`, marking a spot, were we have to insert an external transducer, and targets aren't indices in the array, but strings with the XML-id's of the targets.

⟨ finish the transducer 156b ⟩ ≡

⟨ replace xml-ids of targets by array-indices 157a ⟩
⟨ insert external transducers replacing the pseudo-signals 157b ⟩◇

Macro referenced in 153.

We can simply replace the XML-id's by array-indices using the data we collected in the `statenos`-dictionary.

⟨ replace xml-ids of targets by array-indices 157a ⟩ ≡

```
k=0;
while k<len(self.fst):
  for key in self.fst[k].keys():
    if self.fst[k][key][0]!='fin':
      newt=self.statenos[self.fst[k][key][0]];
    else:
      newt=-1;
    self.fst[k][key][0]=newt;
  k=k+1;
◇
```

Macro referenced in 156b.

Handling external transducers isn't very complicated either, given that the actual algorithmic work behind this is done by the `opunit`.

⟨ insert external transducers replacing the pseudo-signals 157b ⟩ ≡

```
k=0;
while k<len(self.fst):
  for key in self.fst[k].keys():
    if key[:3]=='#T#':
      fs2=[{}];
      txtloader(fs2).load(key[3:]);
      opunit(self.fst).join(k,fs2,self.fst[k][key][0]);
      del self.fst[k][key];
  k=k+1;◇
```

Macro referenced in 156b.

If we come across a pseudo-signal, we simply load it using the `txtloader`, and join it into the current transducer.

6.5 The TXT-Loader

It is the `txtloader`'s job to open a text-file such as the `testdta1.txt`-file at the beginning of this chapter, and convert it to an FST.

Let's have a look at the class's basic outline:

"txtloader.py" 158 ≡

```
from misc import loader;
import string;

class txtloader(loader):
  def __init__(self,fst):
    self.fst=fst;

  def _inserttrans(self,signal,target,output):
    ⟨ insert a single transition into the current fst 164 ⟩

  def _insertpath(self,entr,output):
    ⟨ insert transition into the current fst making up to a path described by entr 163 ⟩

  def _addstring(self,entry,output):
    ⟨ insert the path described by entry to the current fst 161a ⟩

  def _resetall(self,node):
    ⟨ reset output-signals along a path to 0 166b ⟩

  def _resetwherepossible(self,node):
    ⟨ call _resetall, for every path, where it is needed 166a ⟩

  def load(self,filename):
    ⟨ main function 159a ⟩◇
```

Because python's object-oriented capabilities are actually quite poor, but still sufficient for our needs, we will have to introduce a new convention: Every function, that is not intended to be called by an external caller, begins with an underscore.

I absolutely admit that the above code-snippet may look cryptic, and do nothing but confuse the reader. However I hope that the subsequent sections, that reveal the details of these functions, will also make this clear.

Let us begin by looking at the main function:

⟨ main function 159a ⟩ ≡

```
f=open(filename);
line=f.readline();
while line!='':
  line=string.replace(line,'\n','');
  ⟨ process the line 159b ⟩
  line=f.readline();


self._resetwherepossible(0);
f.close();◇
```

Macro referenced in 158.

This is no big deal: Iterate through a textfile, and call `_resetwherepossible(0)` when finished. Details of the resetting are handled in section 6.5.2.

We will hear about the resetting later. Let's first concentrate on the processing:

⟨ process the line 159b ⟩ ≡

```
spl=string.split(line);
word=spl[1];
no=string.atol(spl[0]);
word=word+'&';
self._addstring(word,no);◇
```

Macro referenced in 159a.

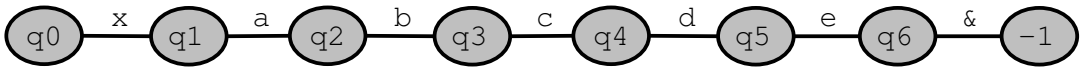


Figure 6.1: A part of an FST containing the string "xabcde"

There's still nothing revolutionary about that: split up each line by the default whitespace-delimiter into a word, and a number this word will get mapped to. Before adding the string to the current FST, we append the `&`-special-signal, to make sure our transducer terminates properly. Consider the string `hand` and `handy`. Once we have `hand` in the FST, if we want to add `handy`, we will need a state such as `{'&':(-1,0),'y':(42,0)}`. Instead of implementing this as a special-case in the inserting-procedure, we just define every word to have a final `&`.

6.5.1 Adding a String

Now how do we go about adding a string into the current transducer?

Consider an example, taken from our `testdta1.txt`-file again, where the string `xabcde` is already in the transducer. That situation is depicted in figure 6.1. The word we want to add to this transducer is the word `xabfgh`. The first thing we would have to do is find out what portion of the current FST could be reused for the new word. Such an algorithm, let's call it A_1, would find out that we would have to add the path `fgh` to the node q_4.

After that all we would have to do is add a state q_7 with a transition $\& \rightarrow -1$, a state q_8 with a transition $h \rightarrow q_7$ and state q_9 with the transition $g \rightarrow q_8$, so we could simply add the transition $f \rightarrow q_9$ to the state q_4, leaving the FST like depicted in figure 6.2, which is exactly what we wanted to achieve. Let's call that algorithm A_2

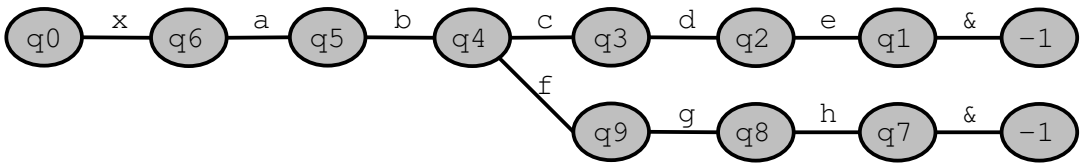


Figure 6.2: A part of an FST containing the string "xabcde" and "xabfgh"

⟨ insert the path described by entry to the current fst 161a ⟩ ≡

⟨ do A_1 161b ⟩

⟨ do A_2 162 ⟩◇

Macro referenced in 158.

Now let's go back one step, and have a more detailled look at A_1.

⟨ do A_1 161b ⟩ ≡

```
curst=0;
cure=0;
while cure<len(entry) and self.fst[curst].has_key(entry[cure]):
  ⟨ output-signal handling for FST-traversal when inserting 165 ⟩
  curst=self.fst[curst][entry[cure]][0];
  cure=cure+1;◇
```

Macro referenced in 161a.

The principle is quite straightforward: Iterate through the characters of the string we should add, and try to traverse the FST, while doing so. We do the traversal by setting the `curst`-variable to the state pointed to by a transition from the current state, which has as a signal, the character we are currently at. We terminate, when no such transition is in the current state, or we run out of characters from the input string.

Note that there is another condition, that makes our loop terminate, that isn't explicitly mentioned in the while-condition. The case I'm talking about is, when we run out of input-states from the FST, or, putting it differently, that the FST terminates, while we traverse it. We never actually reach the state `-1`, because

that would require us to follow the terminating transition from `&` to `-1`. This will obviously never happen, since the symbol `&` isn't (conceptually speaking) defined in the input string.

In each case we leave the `curst`-variable with the index, where we have to add the new path, and the `cure`-variable at the first index of the string, that should be part of the new path (since we weren't able to follow that particular signal in the FST). That means the new path should be `entry[cure:]`, to stay in python-terminology.

Note that we haven't mentioned the output-signals yet. We will talk about them in section 6.5.2, which is why we leave a placeholder in the code for output-signal-handling.

⟨ do A_2 162 ⟩ ≡

```
if (cure+1)<len(entry):
  stno=self._insertpath(entry[(cure+1):],output);
  self.fst[curst][entry[cure]]=[stno,output];
else:
  self.fst[curst][entry[cure]]=[-1,output];◇
```

Macro referenced in 161a.

The details of the A_2-algorithm reveal two cases we handle seperately.

The one from the `if`-block is the standard-case. For example, when we found that we have to add the `fgh` path to the q_4-state. The first thing we do is let the `_insertpath`-function do its work, and insert the path `gh`. Then we simply add the transition, leading to that path, to the current state. Note that at this point in processing we simply add the output-signal to every transition. Again the reader is referred to section 6.5.2 for the details.

The `else`-block is entered when the string we are inserting is in fact a substring of one that is already in the FST. In this case `entry[(cure+1):]` is undefined because the index `cure+1` is already out of bounds (that is what we check in the

if-condition). Or, to put it differently, `entry[cure]` is already the `&`-signal. In this case we don't call the `_insertpath`-function, and set the target to `-1`.

Adding a Path

In the previous section we blindly called a function named `_insertpath` that inserts a path to the FST, and returns the index of the first state in this transducer of the newly created path.

⟨ insert transition into the current fst making up to a path described by entr 163 ⟩ ≡

```
if len(entr)==1:
  return self._inserttrans(entr[0],-1,output);
else:
  stno=self._insertpath(entr[1:],output);
  return self._inserttrans(entr[0],stno,output);
```

◇

Macro referenced in 158.

We already mentioned some of the behavioral details of this routine, in the previous section. Although it could easily be formulated as an iterative routine, I chose a recursive one, which makes it more readable, but the algorithm is trivial anyway.

If the path's length is only one character, we simply insert a transition leading to the `-1`-state, by calling the `_inserttrans`-function.

If it is longer than one character, we insert the path `entr[1:]` (which is the input-path with the first character stripped off), by recursively calling ourselves, and insert a transition to the path that was just inserted. The signal that fires that transition is, of course, the character we just stripped off.

Inserting a single transition is also trivial:

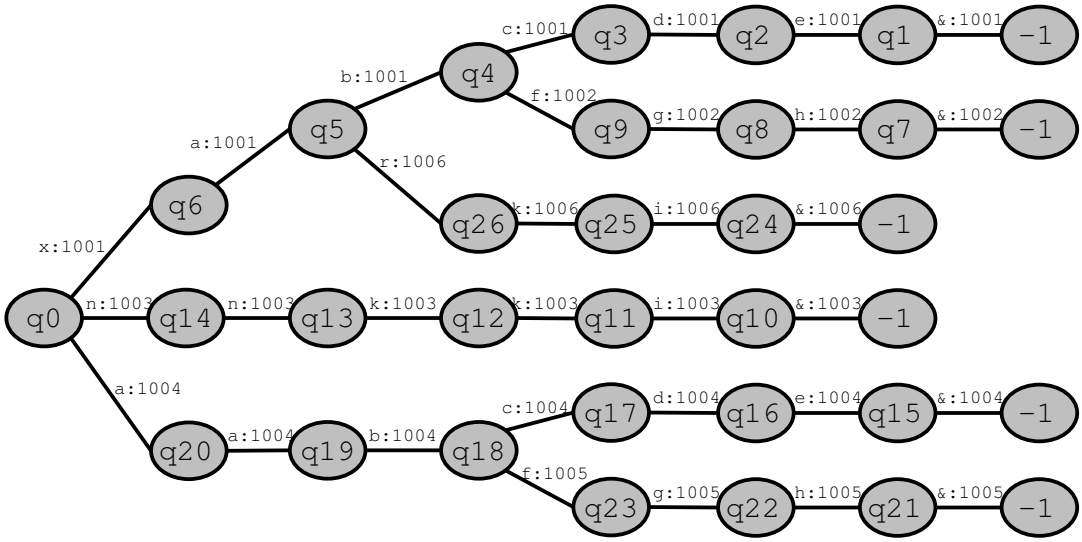


Figure 6.3: What our FST would look like without resetting output-signals

⟨ insert a single transition into the current fst 164 ⟩ ≡

```
newp={signal:[target,output]};
self.fst.append(newp);
return len(self.fst)-1;◇
```

Macro referenced in 158.

6.5.2 Output

Figure 6.3 shows what our FST would look like, if we would run our program like this. The structure is already completed, also the signals are already as intended, the only thing that is a little bit confusing in figure 6.3 are the output-signals (that can be found following the colon in the arc's captions).

Until now we just set the output-field of each transition to the output of the path that this transition was originally created for. Of course we only want to do output once.

Obviously we cannot create output on any transition, that directly or indirectly leads to a decision-state. (That is a state with more than one transition), because, when we enter such a transition, we never know which path we will take,

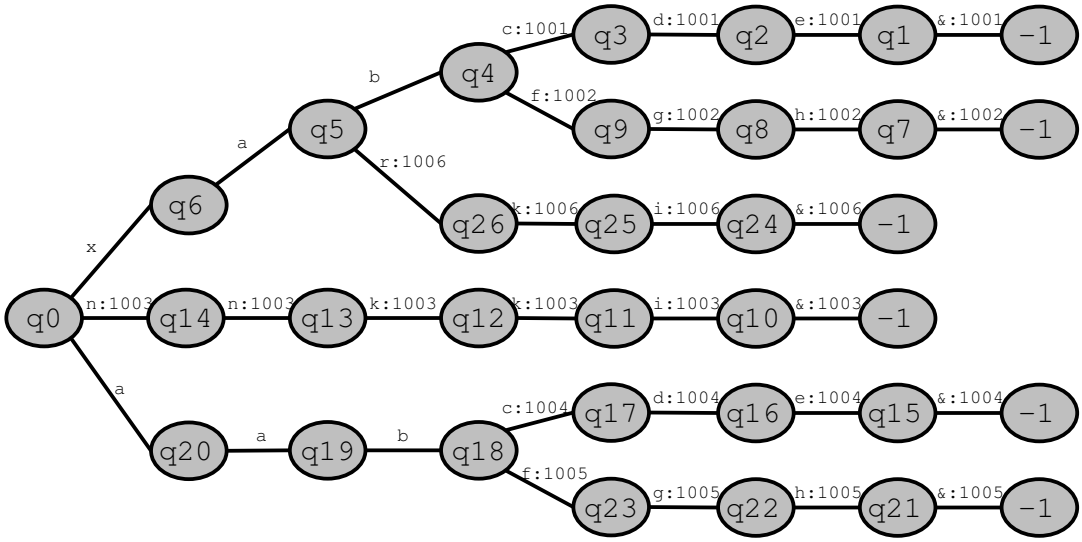


Figure 6.4: What the additional line of code does to our transducer.

once the decision node is reached.

Therefore we can add the following line of code to our program:

⟨ output-signal handling for FST-traversal when inserting 165 ⟩ ≡

```
self.fst[curst][entry[cure]][1]=0;◇
```

Macro referenced in 161b.

.

This makes sure, that when we traverse the FST, in order to find out where to append a new path, when we add a string (what was previously called A_1) we set all output-signals of transitions along that way to `0`. Figure 6.4 shows how the FST looks, after adding this line of code.

Note that it's not possible to "mistakenly" reset ALL output-signals, since at least the `&`-transition, will never be touched by this routine.

The second thing we have to make sure is that, after we've done output once, we don't do it again. This is what the `_resetwherepossible`-routine does.

⟨ call _resetall, for every path, where it is needed 166a ⟩ ≡

```
if node==-1:
  return;


for key in self.fst[node].keys():
  if self.fst[node][key][1]==0:
    self._resetwherepossible(self.fst[node][key][0]);
  else:
    self._resetall(self.fst[node][key][0]);◇
```

Macro referenced in 158.

We will see code that looks like this a lot more often in the program. This is how we recursively traverse the FST. For every transition with an output-signal we call the _resetall-function for the subtree of the FST, that has as root-element the target of the transition. For every transition that doesn't have an output-signal we call ourselves.

⟨ reset output-signals along a path to 0 166b ⟩ ≡

```
if node==-1:
  return;


for key in self.fst[node].keys():
  self.fst[node][key][1]=0;
  self._resetall(self.fst[node][key][0]);
◇
```

Macro referenced in 158.

This is also quite simply a recursive transition of the FST, that sets every output-signal along its way to 0.

After that we have the final product of this class, that is shown in figure 6.5.

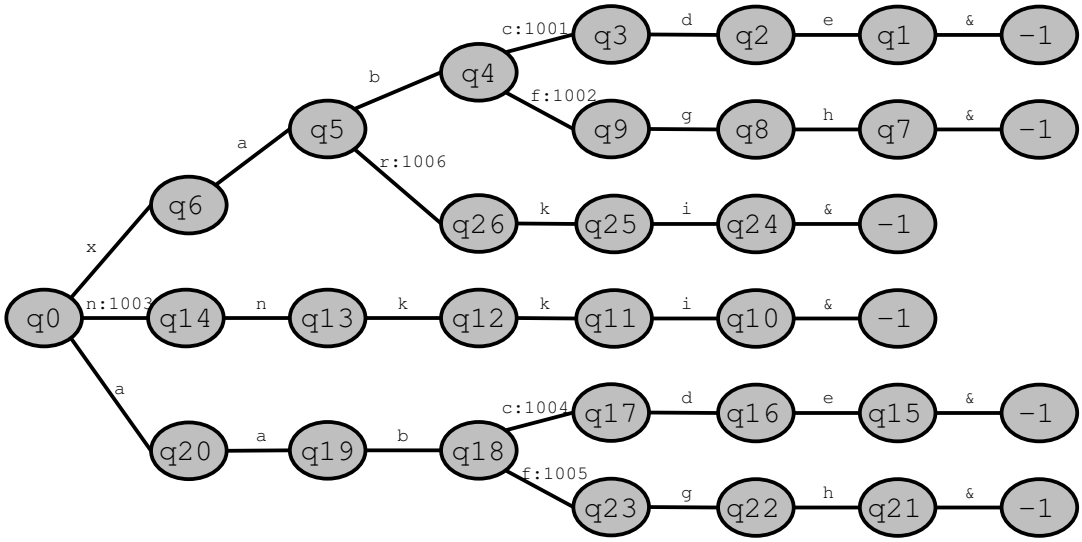


Figure 6.5: Product of the TXT-Loader.

6.6 FST-Operations

After these little warm-up rounds, we will have to discuss the core-component of the FST-toolkit: The `opunit` (As I already mentioned short for OPerational UNIT). The opunit is capable of doing two things: appending an FST to an existing one, and joining two FSTs "in parallel".

Let's have a look at the class:

"opunit.py" 168 ≡

```
import string;

class opunit:
  def __init__(self,fst):
    self.fst=fst;

  def _appendfst(self, add, anode, target):
    ⟨append a complete FST to the current one 169⟩

  def appendfst(self, add):
    self._appendfst(add, 0, -1);

  def _join(self, bnode, bop, add, anode, aop, target):
    ⟨join an FST to the current one 170a⟩

  def join(self, bnode, add, target):
    self._join(bnode, 0, add, 0, 0, target);
```

◇

6.6.1 Appending an FST

Appending an FST to the current one requires us to read every state from the new transducer, append it to the `self.fst`-array, and recode every target-reference to an index of the new array.

⟨ append a complete FST to the current one 169 ⟩ ≡

```
if anode==-1:
  return target;

newdic={};

for key in add[anode].keys():
  n=self._appendfst(add,add[anode][key][0],target);
  newdic[key]=[n,add[anode][key][1]];

self.fst.append(newdic);
return len(self.fst)-1;◇
```

Macro referenced in 168.

Again, the basic structure of the code is a recursive traversal of the tree, that we want to add. We compile each state, that we want to add to the new FST by first recursively appending the subtree pointed to by each transition in the current state, and adding a transition to that newly added subtree to the new state, leaving the input- and output-signal untouched, and recoding the target to the return-value of the recursive call. Then we append the newly created state to the `self.fst`-array, and return the index that this state just got in that array.

6.6.2 Joining an FST

First of all, we have to define, what exactly we mean by joining: Suppose we have a transducer T_1, accepting the set of strings S_1, and a second transducer T_2, accepting the set of strings S_2. Then joining would produce a transducer T_3, accepting the set of strings $S_3 = S_1 \cup S_2$.

In our case the join-function receives a parameter `bnode`, which is the root-node of that subtree of `self.fst`, that will be joined. The tree that we will join

to that is the add-FST, and anode points to the root-node of the subtree of add, that will be joined. target, is the target-state, that all paths from the add-FST will ultimately lead to. (That is, the -1-signal will be recoded to that value in the add-FST). aop and bop are part of the recursive algorithm that will be described in greater detail later.

⟨ join an FST to the current one 170a ⟩ ≡

```
for key in add[anode].keys():
  ⟨ set aope and bope to the output "carried over" from the add or base-FST 170b ⟩
  ⟨ do the joining 171 ⟩
◇
```

Macro referenced in 168.

One of the more complex problems in this context, is to handle output-signals. In some cases output-signals have to be "shifted" along a path when new decision-nodes enter the FST. That is what we need the aop and bop-parameters for. We use aop to "carry over" output-signals from the add-FST, and bop to do the same thing with the base-FST (that is self.fst).

⟨ set aope and bope to the output "carried over" from the add or base-FST 170b ⟩ ≡

```
aope=aop;
if aope==0:
  aope=add[anode][key][1];
bope=bop;
if bope==0 and self.fst[bnode].has_key(key):
  bope=self.fst[bnode][key][1];
  self.fst[bnode][key][1]=0;◇
```

Macro referenced in 170a.

The principle is quite simple: Recode aop to the current output-signal in the add-FST, and bop to the current output-signal in the self.fst-FST, if they are zero. (Later we will pass these in the recursive call).

If we "pick up" an output-signal from the base-FST for carrying it over to a later state, we also have to set the current output-signal to `0`.

⟨ do the joining 171 ⟩ ≡

```
if self.fst[bnode].has_key(key):
  self._join(self.fst[bnode][key][0],bope,add,add[anode][key][0],aope,target);
else:
  if len(self.fst[bnode].keys())==1:
    self.fst[bnode][self.fst[bnode].keys()[0]][1]=bope;
  self.fst[bnode][key]=[self._appendfst(add,add[anode][key][0],target),aope];◇
```

Macro referenced in 170a.

If it is possible to follow a transition in the base-FST's current state, having the same signal as the current transition of the current state in the `add`-FST, we recursively call ourselves, for the target-states.

Otherwise, that is, in case we found the state where we have to add the new subtree, we use `_appendfst` to add it to `self.fst`, linking it to the target we got as a parameter. The output is of course the `aop`-value we carried over.

If by doing so the state would become a decision state, we also have to set the output-signal to the `bop`-value, we carried over.

6.7 The Optimizer

"optimizer.py" 172 ≡

```
import string;

class optimizer:
  def __init__(self,fst):
    self.fst=fst;

  def _joinpossible(self, compare, modify):
    ⟨join a node to each joinable node 174b⟩

  def _joinnodes(self,st):
    ⟨call _joinpossible for each node 174a⟩

  def _concatsubtree(self,curst):
    ⟨concatenate linear paths through the FST, leaving longer signals 177⟩

  def _notorphaned(self,orph,curst):
    ⟨find out whether orph is not an orphan of self.fst's subtree curst 179a⟩

  def _resetorphs(self):
    ⟨reset all orphaned states in the FST 179b⟩

  def optimize(self):
    self._joinnodes(0);
    self._concatsubtree(0);
    self._resetorphs();
```

◇

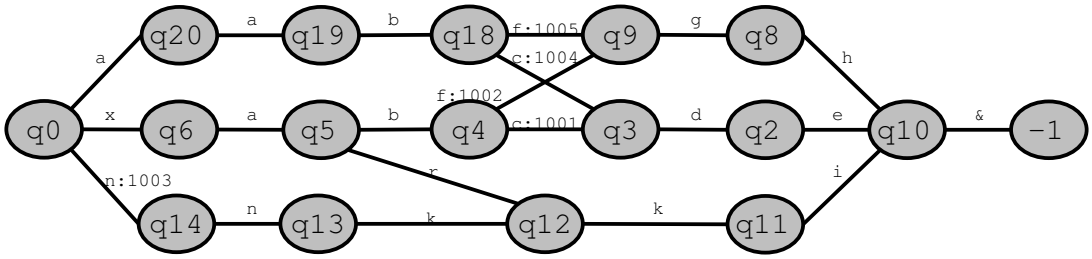


Figure 6.6: An FST after joining.

6.7.1 Joining nodes

In order to write a routine "optimizing" a transducer, we have do define, what exactly we mean by "optimal". Figure 6.6 shows a "more optimal" version of the same FST, shown in figure 6.5. Now what exactly happened to it from figure 6.5 to figure 6.6?

We quite inaccurately refer to that process as "joining", because we already used this terminus for "parallely cascading" two transducers, in the previous section. Note that this has nothing to do with what we are doing here.

It is apparent in figure 6.5, that the paths `de&`, `gh&` and `ki&` are all redundant, and even within these redundant paths there is a redundancy, namely the states q_1, q_7, q_{24}, q_{10}, q_{15}, q_{21}, which are basically all the same. Our process of "joining", defines a means to get rid of these redundancies.

The basic iteration-pattern I chose was straightforward, and I suppose it's not quite optimal, but simple, readable and easily understandable.

⟨ call `_joinpossible` for each node 174a ⟩ ≡

```
if st==-1:
  return;

for sig in self.fst[st].keys():
  self._joinnodes(self.fst[st][sig][0]);

self._joinpossible(st,0);◇
```

Macro referenced in 172.

I guess there's nothing to say about that. It's a simple depth-first, left-to-right, postprocessing tree-traversal, the "processing" being abstracted by a procedure-call to `_joinpossible`.

⟨ join a node to each joinable node 174b ⟩ ≡

```
if modify==compare:
  return 0;
if modify==-1:
  return 0;
```

⟨ find out whether the `compare`-node is joinable to any of the subtrees of `modify` 175 ⟩

```
if not poss:
  return poss;

return self.fst[compare]==self.fst[modify];◇
```

Macro referenced in 172.

We process each node by again traversing the FST, "comparing" each node in the tree, to each other node. (We will see that the term "compare" is not quite correct, but for the time being it will do). Each time this procedure is entered, we are actually dealing with two nodes, `compare` and `modify`.

The `compare`-node is that node this procedure was originally called for from `_joinnodes`. It never gets touched. The `modify`-node is the one we traverse in this recursive procedure. This is the one, that will get modified if possible, because `_joinpossible` not only returns whether a join is possible, it also carries out the joining if possible.

If these two nodes are actually the same, we can, of course, not join them. If the node, we are attempting to modify is already the `-1`-node it is also not possible. If not all the subtrees allow joining, we also can't do it. If for all these reasons we are still in the procedure, the joining is possible if, and only if, the states are completely equal.

One might think that conceptual "equality" is actually different from the `==`-operator, because python would check whether each entry in the dictionary has exactly the same signal mapping to exactly the same `(target,output)`-tuple. In figure 6.5 for example the subtrees from q_3 and q_{17} are actually equal, but the python objects {`'d':(2,0)`} and {`'d':(16,0)`} are different. The trick, that makes the `==` work correctly in this case, is, that we have already done the joining of the subtrees, at this point in processing. That means that the subtree 16, and the subtree 2 will already be joined, therefore leaving the states as {`'d':(2,0)`} and {`'d':(2,0)`}, which are obviously equal.

⟨ find out whether the `compare`-node is joinable to any of the subtrees of `modify` 175 ⟩ ≡

```
poss=1;

for sig in self.fst[modify].keys():
  if self._joinpossible(compare, self.fst[modify][sig][0]):
    self.fst[self.fst[modify][sig][0]]={};
    self.fst[modify][sig][0]=compare;
    poss=0;◇
```

Macro referenced in 174b.

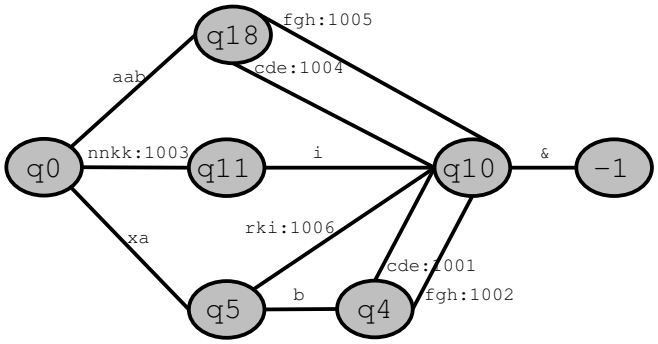


Figure 6.7: An FST after concatenating paths.

In order to find out whether all subtrees allow joining we first assume, that they do. If we find, that one of the subtrees of the current `modify`-node is joinable to the current `compare` node, then the `modify`-node itself cannot be joinable to it, and we therefore set `poss=0` in this case.

If we find out that one of the subtrees of the `modify`-node is joinable to the `compare`-node, then we can modify the `modify`-node, by setting the target of the transition, that we found to be joinable to the `compare`-state to `compare` itself, and the old state to {}.

6.7.2 Concatenating Linear Paths

Figure 6.7 shows a version of our sample-FST, which is even more "optimal". The background of this idea is purely technical. The FST will be handled by an Intel 80468 or higher, and as I already mentioned, this processor can compare a 32-bit-quantity as quickly as an 8-bit-quantity. In order to make use of that capability this optimization-technique, is used, to create longer signals (of up to 32 bits) by concatenating signals, connecting non-decision-states.

⟨ concatenate linear paths through the FST, leaving longer signals 177 ⟩ ≡

```
if curst<0:
  return;

for thiskey in self.fst[curst].keys():
  thisst=self.fst[curst][thiskey][0];
  nextkey='';
  if thisst!=-1:
    nextkey=self.fst[thisst].keys()[0];

  if nextkey!='&':
    if thisst<0 or len(self.fst[thisst])>1 or (len(thiskey)+len(nextkey))>4:
      self._concatsubtree(thisst);

    elif len(self.fst[thisst])==1:
      ⟨ concatenate the current and the next state 178 ⟩
      return;◇
```

Macro referenced in 172.

Yet another recursive traversal of the FST. For each transition in each state we do the following: First we find out about the current key (`thiskey`), the current state `thisst`, the key of the transition we might want to concatenate to the current one `nextkey`.

If we are at the `-1`-state, at a decision-state or if we are about to produce a signal with a length greater than 4, we go on to the next state and can not do any further optimization. Otherwise, we can concatenate the current and the next state.

⟨ concatenate the current and the next state 178 ⟩ ≡

```
nextst=self.fst[thisst][nextkey][0];
self.fst[curst][thiskey+nextkey]=[nextst,self.fst[curst][thiskey][1]];
del self.fst[curst][thiskey];
self._concatsubtree(curst);◇
```

Macro referenced in 177.

This routine finds out about the state, we have to link the current one to, adds the new transition for the signal `thiskey+nextkey`, deletes the old one, and calls the procedure for the current state again, to see what else we can optimize in the current state, before breaking off the currently running function. This algorithm may also seem a bit naive, but for our purposes it will do.

6.7.3 Removing Orphaned States

While concatenating the paths we often leave states as "orphans", that means when recursively tranversing the FST, we never come across these states, yet they are still in the array. For easier processing by subsequent routines that operate on these FSTs it would be fine, to set all the states in the array that are completely irrelevant to the FST to a defined value, say {}.

We divide this algorithm into two parts. Finding out whether a specific state is not an orphan, that means it can be reached through recursive traversal, is one routine, and another routine goes iteratively through the array, checking for each state, whether it is not an orphan. If it is not not an orphan, it can safely be reset to {}.

⟨ find out whether orph is not an orphan of `self.fst`'s subtree curst 179a ⟩ ≡

```
if curst==-1:
  return 0;

if orph==curst:
  return 1;

for thiskey in self.fst[curst].keys():
  if self._notorphaned(orph,self.fst[curst][thiskey][0]):
    return 1;

return 0;◇
```

Macro referenced in 172.

The `-1` state is not an orphan, if the current state, is the state, in question of being an orphan, it is definitely not an orphan, because we would never have reached it otherwise.

Then we simply assume, that it is not not an orphan, and check in each subtree, whether we can find the state there. If we find any subtree, where that state is not an orphan, it is also not an orphan of the current subtree.

⟨ reset all orphaned states in the FST 179b ⟩ ≡

```
k=0;
while k<len(self.fst):
  if not self._notorphaned(k,0):
    self.fst[k]={};
  k=k+1;◇
```

Macro referenced in 172.

This is, for a change, not a recursive traversal of the FST, but a simple linear iteration through it. While doing that iteration we simply set every state, that

is found not to be not an orphan to {}.

6.8 Future Directions of this Toolkit

This section aims to give an overview of future directions of this toolkits. The implementation, that was presented in the previous sections is basically a "quick 'n dirty"-hack leaving (a lot of) room for further optimization.

6.8.1 Probabilistic Data

This toolkit handles states with transitions, that are not in any way ordered. If we are trying to model a dictionary for a morphological parser for natural language processing, using this software, we have some very valuable statistic data available, making statements about how likely a certain transition in a state is to be followed. That makes it possible to order transitions by their likeliness, potentially shortening the number of transitions that have to be handled, before the correct one is found.

6.8.2 Optimizations

Most of the optimization-algorithms that were used in this implementation aren't actually finding any "optimum", they rather make it "a little bit more optimal". For example when deciding which transition should do the output I followed a pure intuition, when deciding that this is always the transition leading away from the last decision-state for the specific path. When concatening the linear paths it was a pure convenience for the programmer, to always concatenate the signals in an order that naturally arises from the traversal of the FST, although in some cases this might not be optimal.

6.8.3 External FSTs

It might be desireable to be able to insert a transduction reading an external FST from another XML-file, so that FSTs could easily be modularized.

6.8.4 Dirty States

When the `opunit` does certain operations it sometimes leaves behind dirty states such as {`'&': (42,0)`}, of course such a transition could be completely avoided.

6.8.5 Parameterizability

At the moment this code works only in a quite limited amount of operating environments. The framework should extend its parameterizability, providing a means of easily porting it, for example to other processors or other assemblers and calling-conventions.

Chapter 7

Parser

7.1 General Considerations

As we have quite extensively elaborated on the subject of parsing in the first part of this paper, it is not neccessary to provide a more detailled account of what parsing is all about, and what exactly we expect our parser to do, in this section.

The parser we want to construct for the LISA-project is an Earley-parser designed mainly according to considerations regarding performance and simplicity.

Although whereever possible we try to make use of general interfaces, so the parser can easily be extended (most notably the event-oriented parseforest-traverser, we will hear about later), we keep focus on a simple task: Reading a file of plain ASCII-text, using a morphological analyzer and building up an effective representation of a parse-forest, that is subsequently translated into PROLOG.

This time the language of choice was C (plain ANSI/ISO-C, not C++), rather than Python, because the much lower level that C operates on seemed to fit not only out of performance-reasons, but also for explaining the data- and runtime-structures that are used, when building a parser. To me it seemed inappropriate to use a language like Python or make extensive use of standard-libraries (like MFC) providing high-level data-structures, because I wanted to

make these data-structures transparent to the reader, rather than follow the "never mind what's in the black box"-zeitgeist software-engineering seems to be following nowadays.

7.1.1 Representing the Text

The text we want our parser to operate on is represented in a way, as if we inteded it to be read by humans. The test-files looks like this:

"test.txt" 183 ≡

```
Kommissar Klug is on the verge of solving one of his
most complicated cases. He knows that at least two
of the suspects Peter, Frank, Richard and Steve are involved. If he
can prove Richard's guilt, he will also know that Frank and
Peter are involved. Proving Steve's participation he can
confirm Peter's innocence. "Allright", he mumbles
under his breath, "The case isn't closed yet, but
we can already arrest someone".

Who is Kommissar Klug talking about?
◇
```

It is definitely not what one would call machine-readable, but after all, understanding this kind of input is what this paper is all about.

7.1.2 Representing the Grammar

An Earley-Parser can be thought of as having two "inputs". The text it should parse, and the grammar it should use for parsing. Unlike techniques like LR-parsing (and its generalization for ambiguous grammars, described by Tomita (1987)) that use precomputed LR-tables, an Earley-Parser directly operates on a CFG.

I chose to represent the CFG in C as a three-dimensional array of bytes.

⟨ define the data-type of the grammar 184 ⟩ ≡

```
#define GRATYPE(id) const char id[NUM_NONTERM][12][MAX_RULLEN + 1]
#define MAX_RULLEN 3◇
```

Macro referenced in 190b.

That the Earley-parser directly operates on CFGs has the big advantage that it is possible for an Earley-parser to read the CFG at runtime. However, for simplicity's sake, I chose not to make use of this, but rather to hardcode the grammar in the initialization of a global variable named `grammar`.

⟨ define the grammar 185 ⟩ ≡

```
GRATYPE(grammar)={
  {
    {2, NTGS_CL, TGS_EOS},
    {2, NTGS_WHQ, TGS_EOS},
    GRA_STOP
  }, {
    {2, TGS_DET, NTGS_NP},
    {2, TGS_ADJ, NTGS_NP},
    {2, TGS_FUNC, NTGS_NP},
    {2, TGS_FUNC, NTGS_CL},
    {2, TGS_N, NTGS_NP},
    {2, TGS_NAME, NTGS_NP},
    {2, NTGS_NP, NTGS_PP},
    {2, NTGS_PP, NTGS_NP},
    {3, TGS_ADJ, TGS_P, NTGS_NP},
    {1, TGS_N},
    {1, NTGS_NL},
    GRA_STOP
  }, {
    {2, NTGS_VCL, NTGS_NP},
    {2, NTGS_VP, TGS_ADV},
    {2, NTGS_VP, NTGS_PP},
    {1, NTGS_VCL},
    {2, NTGS_VCL, TGS_ADJ},
    {2, NTGS_VP, TGS_FUNC},
    GRA_STOP
  }, {
    {2, TGS_P, NTGS_NP},
    {2, TGS_P, NTGS_VP},
    GRA_STOP
  }, {
```

◇

Macro defined by 185, 186.
Macro referenced in 191a.

⟨ define the grammar 186 ⟩ ≡

```
      {1, TGS_NAME},
      {2, TGS_NAME, NTGS_NL},
      {2, TGS_FUNC, NTGS_NL},
      GRA_STOP
    }, {
      {2, NTGS_NP, NTGS_VP},
      {1, TGS_EX},
      {3, TGS_FUNC, NTGS_CL, NTGS_CL},
      {2, NTGS_VP, NTGS_CL},
      {2, NTGS_QCL, NTGS_CL},
      {2, NTGS_CL, NTGS_QCL},
      {3, NTGS_CL, TGS_FUNC, NTGS_CL},
      {2, NTGS_PP, NTGS_VP},
      GRA_STOP
    }, {
      {3, TGS_FUNC, NTGS_VP, NTGS_NP},
      {2, TGS_FUNC, NTGS_VP},
      {3, TGS_FUNC, NTGS_VP, NTGS_VP},
      GRA_STOP
    }, {
      {1, TGS_V},
      {2, TGS_V, TGS_V},
      {2, TGS_V, TGS_P},
      GRA_STOP
    }, {
      {3, TGS_QM, NTGS_CL, TGS_QM},
      GRA_STOP
    }
  };◇
```

Macro defined by 185, 186.
Macro referenced in 191a.

This initialization-string already provides us with a full insight of this data-structure. The first dimension (the one dereferenced with `grammar[SYM]`) wouldn't be neccessary, but later in this paper we will see why it is more efficient. It refers to a two-dimensional array, that holds all rules of the form $SYM \rightarrow \alpha$. We need the second and third dimension, because there is presumably more than one such rule, and in each of the rules α can have more than one symbol. Therefore `grammar[1][2][3]`) would dereference the third symbol of α in the second rule that takes the form $S \rightarrow \alpha$, given that S is assigned the numeric value `1`.

Boundaries for the first array-dimension are given via `NUM_NONTERM`, the count of nonterminal symbols. Throughout this program, we will try to keep consistent with the convention that data-structures of dynamic size are terminated by a special symbol. This is why boundaries of the second array-dimension are given only implicitly with the `GRA_STOP` symbol. When iterating through the rules for `grammar[SYM]` this symbol should be read as "no more rules available for `SYM`". `GRA_STOP` is defined as

⟨ define `GRA_STOP` 187 ⟩ ≡

```
#define GRA_STOP        {0}◇
```

Macro referenced in 190b.

The only exception to this is the data-structure representing a single CFG-rule, the third dimension of our grammar-array. Instead of terminating a rule with a special symbol, the zeroth element of the rule carries the number of symbols this rule has, similar to the way PASCAL represents strings. We make this exception, because providing the symbol-count makes testing a state for completion a lot more efficient. (All one would have to do then is compare the position of the • and the number of symbols in the rule, instead of having to iterate over the rule and count the symbols).

To make code more readable the numeric value of each symbol is defined using C's precompiler.

⟨ define numeric values for non-terminal symbols 188 ⟩ ≡

```
#define NTGS_S          0
#define NTGS_NP         1
#define NTGS_VP         2
#define NTGS_PP         3
#define NTGS_NL         4
#define NTGS_CL         5
#define NTGS_WHQ        6
#define NTGS_VCL        7
#define NTGS_QCL        8

#define NUM_NONTERM  9◇
```

Macro referenced in 190b.

The prefix `NTGS` stands for "non-terminal grammar symbol", while the prefix `TGS` stands for "terminal grammar symbol". The `NUM`-macros provide the count of terminal, respectively non-terminal symbols, and are very useful for bounding arrays and for iterating over them.

⟨ define numeric values for terminal symbols 189a ⟩ ≡

```
#define TGS_EOS         128
#define TGS_ADJ         129
#define TGS_ADV         130
#define TGS_DET         131
#define TGS_EX          132
#define TGS_N           133
#define TGS_NAME        134
#define TGS_P           135
#define TGS_V           136
#define TGS_FUNC        137
#define TGS_QM          138

#define NUM_TERM  11◇
```

Macro referenced in 190b.

Note that the first non-terminal is assigned the numeric value 0, while the first terminal is assigned the numeric value 128. This way the parser can easily distinguish terminals and non-terminals, by testing the most significant bit, and it can easily map numeric-values to values used for indexing arrays, by erasing the most significant bit of a terminal symbol's numeric value.

Again for readability's sake these operations are defined as precompiler-macros

⟨ define macros for distinguishing terminals and nonterminals 189b ⟩ ≡

```
#define IS_POS(sym) (sym&0x80)
#define AS_POS(sym) (sym&0x7F)◇
```

Macro referenced in 190b.

For forming human-readable output it is also desirable to provide a way of mapping numeric-values back to strings (called debug-symbols, therefore `debsym` for short). This is done using two globally available string-arrays:

⟨ define arrays mapping numeric values back to strings 190a ⟩ ≡

```
const char debsym_pos[NUM_TERM][5]={
  "EOS", "Adj", "Adv", "Det", "Ex", "N", "Name", "P", "V", "Func", "QM"};
const char debsym_nt[NUM_NONTERM][4]={
  "S", "NP", "VP", "PP", "NL", "CL", "WHQ", "VCL", "QCL"};◇
```

Macro referenced in 191a.

Now we have everything we need to handle the grammar. This is what the grammar-module looks like:

"grammar.h" 190b ≡

```
#ifndef HAVE_GRAMMAR_H
#define HAVE_GRAMMAR_H

⟨ define the data-type of the grammar 184 ⟩

⟨ define numeric values for non-terminal symbols 188 ⟩
⟨ define numeric values for terminal symbols 189a ⟩
⟨ define GRA_STOP 187 ⟩
⟨ define macros for distinguishing terminals and nonterminals 189b ⟩

extern GRATYPE (grammar);
extern const char debsym_pos[NUM_TERM][5];
extern const char debsym_nt[NUM_NONTERM][4];

#endif
◇
```

"grammar.c" 191a ≡

```
#include "grammar.h"
```

⟨ define the grammar 185, . . . ⟩

⟨ define arrays mapping numeric values back to strings 190a ⟩

◇

7.2 A Simple Earley-Recognizer

In this section we will build a small Earley-recognizer that is declared as:

⟨ declaration for the main routine 191b ⟩ ≡

```
int parse (FILE * f, struct chart **top, struct state **succ);◇
```

Macro never referenced.

Its overall structure looks like this:

⟨ implementation for the main routine 192a ⟩ ≡

```
int
parse (FILE * f, struct chart **top, struct state **succ)
{
  ⟨ declarations for the parse-routine 195a, ... ⟩

  if (feof (f))
    return FALSE;

  ⟨ create the chart (192b *top ) 198a ⟩

  ⟨ create the initial state 195b ⟩

  ⟨ work off the agenda 197a ⟩

  return TRUE;
}◇
```

Macro referenced in 228.

The parse-routine returns a boolean value, corresponding to whether it could successfully read from the input-file *f. The return-value has nothing to do with whether the parse itself was successful or a syntax-error. The routine leaves the address of the initial state where **top points to, and the address of the final state in case of a successful parse or NULL in case of a syntax-error where **succ points to.

The caller needs these two states in order to inspect the parse-forest we build up during parsing, but in this section we shouldn't be concerned with the parse-forest, yet. All we want to do in this section is recognize whether the input is syntactically correct or incorrect.

Note that we use the word chart here also for a single entry in what is commonly called the chart. To be precise this is the data-structure that holds all the states corresponding to a specific position in the input, but this shouldn't cause any confusion, since the context usually makes it clear.

7.2.1 Basic data-structure

The top-level data-structure we will be operating on is a list of charts.

⟨ data-structure for the chart 193 ⟩ ≡

```
struct chart
{
  struct statelist *itemtop;
  struct statelist *itemtail;

  ⟨ indices for the states in the chart 202a, ... ⟩

  struct chart *next;

  struct wordref *word;

  ⟨ numeric index for debugging-purposes 214b ⟩
  ⟨ general purpose pointer for the chart 227b ⟩
};◇
```

Macro never referenced.

Each item in the list holds two pointers referring to the top and the tail of a `statelist` and a pointer referring to a `wordref`, because every word corresponds to exactly one position in the input, and every chart corresponds to exactly one position in the input.

`struct statelist` is a wrapper-structure for listing `state`s in a linked list.

⟨ data-structure for listing states 194a ⟩ ≡

```
struct statelist
{
  struct state *st;
  struct statelist *next;
};◇
```

Macro never referenced.

`struct wordref` is a wrapper-structure that represents a word.

For now we keep focus on building a simple Earley-recognizer, and we are now at the most central data-structure of the Earley-algorithm: the state.

⟨ data-structure for states 194b ⟩ ≡

```
struct state
{
  char rule_left;
  char *rule_right;

  char bul_pos;
  struct chart *glb_pos;
  struct chart *rul_pos;

  ⟨ pointers for the parse-forest 221, ... ⟩
  ⟨ general purpose pointer for the state 227c ⟩
};◇
```

Macro never referenced.

Let's have a look at the prominent initial state $\lambda \rightarrow \bullet S[0,0]$ and how we load it into a `struct state` declared as

⟨ declarations for the `parse`-routine 195a ⟩ ≡

```
struct state *ini;
char inirule[] = { 1, NTGS_S };
```

◇

Macro defined by 195a, 196, 199, 215a.
Macro referenced in 192a.

⟨ create the initial state 195b ⟩ ≡

```
ini = (struct state *) immalloc (sizeof (struct state));
ini->rule_left = NTGS_LAMBDA;
ini->rule_right = inirule;
ini->bul_pos = 0;
ini->glb_pos = *top;
ini->rul_pos = *top;
```

⟨ initialize the data-structures needed for the parse-forest (195c `ini`) 222a, ... ⟩◇

Macro referenced in 192a.

First we allocate memory for this state using `immalloc`, which can be thought of as doing the same as `malloc`.

The rest is rather self-explaining: `rule_left` is the symbol on the left-hand-side of the →. `rule_right` is the rest of the grammar-rule. It is implemented as a pointer to the grammar. `bul_pos` indicates the position of the • in the state, with the index starting at 0, which indicates that the bullet is at the far left. When the bullet is at the far right then `bul_pos` equals `rule_right[0]`. (Recall that this is where we keep the length of a grammar-rule).

`glb_pos` and `rul_pos` are what we've called the "global position" and the "rule-position" in the first part of this paper. In the examples we represented these as numeric values used for indexing the list that contains the charts. Of course this wouldn't lead us anywhere in this case, since we've decided to implement the chart-list as a linked-list. That's why `glb_pos` and `rul_pos` are

implemented as pointers to the charts corresponding to the input-positions, that would otherwise be indicated as numeric values.

7.2.2 The Agenda

Recall that the Earley-algorithm is able to dynamically manipulate its own runtime structure by working off the chart while adding new states to the chart. The agenda and the chart is actually the same. I use the word agenda for chart only when I want to emphasize the chart's role as a "guide" through the runtime of the parser.

First the parser gives itself something to do by enqueuing the initial state to the agenda. Then it simply iterates over the agenda, interpreting each state "along the way" until it finds an item in the agenda that tells it to stop doing so.

We simply use a pointer like

⟨ declarations for the `parse`-routine 196 ⟩ ≡

```
struct chart *cur;
```

◇

Macro defined by 195a, 196, 199, 215a.
Macro referenced in 192a.

to iterate through the agenda. The iteration looks like this:

⟨ work off the agenda 197a ⟩ ≡

```
enqueue (ini, *top);
(*top)->word = NULL;

cur=(*top);

for (;;)
  {
    struct wordref *curwr;

    ⟨ assign an index for debugging-mode 215b ⟩

    ⟨ create the chart (197b cur->next ) 198a ⟩
    ⟨ get the next word from the input and save it in curwr 198b ⟩
    cur->next->word = curwr;

    ⟨ print the current word for debugging 215c ⟩

    ⟨ handle all states in the current chart 200 ⟩

    ⟨ do debug-output for the finished chart 219b ⟩

    if (cur->next->itemtop == NULL)
      {
        ⟨ handle the syntax-error 201b ⟩
      }

    cur = cur->next;
  }
end:◇
```

Macro referenced in 192a.

The infinite loop and the `end`-label might seem a bit awkward, but artificially formulating a while-condition would even be worse (at least to my taste of "coding aesthetics", but de gustibus non est disputandum). This way we can simply terminate the recognizer by jumping to `end` whenever we find we are done with the job. Note that `break` would also artificially complicate things since we would have to `break` two loops at once.

⟨ create the chart 198a ⟩ ≡

```
@1 = (struct chart *) immalloc (sizeof (struct chart));
bzero (@1, sizeof (struct chart));◇
```

Macro referenced in 192a, 197a.

Creating a new chart requires `bzero`ing it first, because some routines depend on `NULL`-values indicating empty lists.

After creating a new chart we have to assign it to a word read from the input (except for the top chart, which corresponds to that position in the input, where nothing has been read so far).

⟨ get the next word from the input and save it in `curwr` 198b ⟩ ≡

```
curwr = NULL;
if ((cur->word == NULL) || (cur->word->morph & 0xF0000000))
  {
    curwr = getnextword (f);
    if (curwr == NULL)
      return FALSE;
  }◇
```

Macro referenced in 197a.

If `cur == NULL` (that is, if we are in the top chart), or if the POS of the word, assigned to the current chart is not an end-of-sentence-character we can read from the input. We don't do any reading after having discovered the end of the

sentence, because reading beyond the bounds of the current sentence would "read away" a word, that will be needed in the next parse.

If the result of `getnextword(f)` is `NULL` we return `FALSE` because in that case the premorpher found an `EOF`. This situation is considered a reading-error, and not a syntax-error. Otherwise the result is saved in `curwr`.

The interpretation of the states in a chart is probably the most prominent part of the Earley-algorithm.

Again we use a pointer-variable

⟨ declarations for the `parse`-routine 199 ⟩ ≡

```
struct statelist *curit;
```

◇

Macro defined by 195a, 196, 199, 215a.
Macro referenced in 192a.

to iterate through the states in a chart. This iteration looks like this:

⟨ handle all states in the current chart 200 ⟩ ≡

```
for (curit = cur->itemtop; curit != NULL; curit = curit->next)
  {
    ⟨ do debug-output of the current state 216 ⟩

    if ((!(curit->st->rule_right == NULL))
         && (curit->st->rule_right[0] > curit->st->bul_pos))
      {
        if (IS_POS (curit->st->rule_right[curit->st->bul_pos + 1]))
          scanner (curit->st);
        else
          predictor (curit->st);
      }
    else
      {
        if (curit->st->rule_left == NTGS_LAMBDA)
          {
            ⟨ handle the successfullly finished parse 201a ⟩
          }
        else
          completer (curit->st);
      }
  }◇
```

Macro referenced in 197a.

In the first part of this paper we already explained the working of the three basic parts of the Earley-algorithm, the PREDICTOR, the SCANNER and the COMPLETER. Recall that the PREDICTOR is run for every state that has a nonterminal to its right, to "expand" rules that could possibly lead to a solution, the SCANNER is called for every state that has a terminal to its right, to scan the

input, and the COMPLETER is called for every complete state (that is a state with the • at the far right) in order to advance rules that are looking for the symbol we just proved the input to be interpretable as. Make sure you've completely understood that section in the first part, before going on.

If the parse we were working on was successful, we simply save the state that gave us this idea in `*succ`, and jump to the `end`.

⟨ handle the successfullly finished parse 201a ⟩ ≡

```
cur->next = NULL;
(*succ) = curit->st;
goto end;◇
```

Macro referenced in 200.

If the parse was not successful, we do the same, except for `*succ` which has to take the value `NULL` in this case, as we've mentioned before.

⟨ handle the syntax-error 201b ⟩ ≡

```
⟨ do debug-output for the syntax-error 217 ⟩
cur->next = NULL;
(*succ) = NULL;
goto end;◇
```

Macro referenced in 197a.

7.2.3 Enqueuing and Indexing

Now that we've seen how we can work off states in the chart, let's consider the next question: How do the states get there? Traditional implementations of the Earley-algorithm have a routine called ENQUEUE, that takes care of this. This routine is called from the PREDICTOR, SCANNER and COMPLETER functions to enqueue a newly created state to a chart.

The most important feature of this routine is that it checks whether a given state is already in the chart, before it actually adds the state to the chart. Optimizing this part is essential for achieving high performance, because the parser

spends most of its running time doing this check. (Profiler-results showed, that the parser spent 54.74% of its total running time doing this check, when run against our sample-file and sample-grammar!)

This is why I've tried to make that lookup as fast as possible, using a separate index-structure, instead of iterating over all states in the chart, and checking whether we "incidentally" come across one that equals the one we are looking for.

⟨ indices for the states in the chart 202a ⟩ ≡

```
struct stateindex *idx;
```

◇

Macro defined by 202a, 211a.
Macro referenced in 193.

⟨ definition of the `struct stateindex` 202b ⟩ ≡

```
struct stateindex
{
  struct state *st;
  struct stateindex *less;
  struct stateindex *greater;
};◇
```

Macro never referenced.

The `struct stateindex` is a node in a heap. It refers to a state and two subtrees. The subtree pointed to by `*less` contains all the states that are smaller, and the subtree pointed to by `*greater` contains all the states that are greater than the state pointed to by `*st`.

We decide whether a state is greater or smaller than another one using a function called `compare`, where return values are interpreted anolagous to those of `strcmp`. If $s1 < s2$ the return-value is negative, if $s1 = s2$ it is zero, if $s1 > s2$ it is positive. It doesn't matter how this comparison is actually done, as long as

half of the comparisons return a negative, and half of the comparisons return a positive value.

I implemented it like this:

⟨ compare-implementation 203 ⟩ ≡

```
int
compare (struct state *s1, struct state *s2)
{
  int rc;

  if ((rc = (s1->rule_right - s2->rule_right)))
    return rc;
  if ((rc = (s1->bul_pos - s2->bul_pos)))
    return rc;
  if ((rc = (s1->glb_pos - s2->glb_pos)))
    return rc;
  if ((rc = (s1->rul_pos - s2->rul_pos)))
    return rc;

  ⟨ check data-structures for the parse-forest 226a ⟩

  return 0;
}◇
```

Macro never referenced.

Unfortunately experimental results using a debugger showed that this function doesn't exhibit that feature. Most of the comparisons are positive. Creating a better `compare`-function might be subject to further work on this parser, since the parser spent 22.6% of its running time in this function. Providing a way to compare two states with return-values of a better statistical distribution would also dramatically speed up the `find`-routine, which uses 32.2% of the total running-

time.

Given a data-structure for indexing states in a heap, and a way of comparing two states we can now implement `find` as standard heap-lookup.

⟨ `find`-implementation 204 ⟩ ≡

```
struct stateindex **
find (struct stateindex **idx, struct state *st)
{
  if ((*idx) == NULL)
    return idx;
  else
    {
      int rc = compare (st, (*idx)->st);

      if (rc == 0)
        return idx;

      if (rc < 0)
        return find (&((*idx)->less), st);
      else
        return find (&((*idx)->greater), st);
    }
}◇
```

Macro never referenced.

The only thing that might seem unfamiliar, is why we are working with pointers to a pointer to an index (`**idx`), instead of using simple pointers to an index (`*idx`). The reason is that we want to give the caller the possibility to modify such a pointer, in order to append a state at the right node of the index.

⟨ enqueue-implementation 205 ⟩ ≡

```
struct state *
enqueue (struct state *news, struct chart *chart)
{
  struct stateindex **si;

  si = find (&(chart->idx), news);

  if ((*si) == NULL)
    {
      ⟨ add the state to the chart 206, ... ⟩

      return news;
    }
  else
    return (*si)->st;
}◇
```

Macro referenced in 228.

Provided with these indexing-facilities enqueuing is quite straightforward. All we have to do is see if we can find a given state in a given chart. If we can't find it, we add it and return the state we have just added. If we can find it, we return the state that is equal to the one we were supposed to add to the chart instead.

⟨ add the state to the chart 206 ⟩ ≡

```
struct stateindex *newsi;


add_to_listtail (&(chart->itemtop), &(chart->itemtail), news);


newsi = (struct stateindex *) immalloc (sizeof (struct stateindex));
newsi->less = NULL;
newsi->greater = NULL;
newsi->st = news;
(*si) = newsi;
```

◇

Macro defined by 206, 211b.
Macro referenced in 205.

All we have to do, to actually add a new state to a chart, is append it to the linked list we use for traversing the agenda using `append_to_list` (which is an implementation of the well-known algorithm used for adding an element to the tail of a linked list) and create a new index-item, that we add to that node of the index the `find`-routine returned.

7.2.4 The Predictor

Recall that the PREDICTOR is that part of the Earley-algorithm that takes care of expanding states which have a nonterminal to the right of their •, and is therefore responsible for the top-down nature of the parser.

⟨ the PREDICTOR 207 ⟩ ≡

```
int
predictor (struct state *st)
{
  char sym;
  int i;

  ⟨ do debug-output for the predictor 220a ⟩

  sym = st->rule_right[(unsigned int) st->bul_pos + 1];

  for (i = 0; grammar[(unsigned int) sym][i][0] != 0; i++)
    {
      ⟨ add the current rule as a new state 208a ⟩
    }

  return 0;
}◇
```

Macro referenced in 228.

First the predictor finds out which symbol *SYM* is at the right of the • in the state it was called for (`*st`). Then it iterates over all rules in the grammar that have the form `SYM` $\rightarrow \alpha$. This is why we have partitioned the grammar in such a way, because instead of having to iterate over all of the rules, and check the symbol to the left of the • for equality with the symbol we want to expand, we can simply iterate over all the rules in `grammar[`*SYM*`]`.

⟨ add the current rule as a new state 208a ⟩ ≡

```
struct state *newst;

newst = (struct state *) immalloc (sizeof (struct state));
newst->rule_left = sym;
newst->rule_right = (char *) grammar[(unsigned int) sym][i];

newst->bul_pos = 0;
newst->glb_pos = st->glb_pos;
newst->rul_pos = st->glb_pos;
```

⟨ initialize the data-structures needed for the parse-forest (208b `newst`) 222a, ... ⟩

```
enqueue (newst, st->glb_pos);◇
```

Macro referenced in 207.

We initialize the fields `bul_pos`, `glb_pos` and `rul_pos` according to the standard Earley-algorithm.

The • in a predicted state is, of course, at the far left, since we haven't read any input for it, or completed it so far. Therefore `bul_pos` must be zero.

The state we predict shouldn't only find the correct symbol (that is the symbol to the right of the • in the state we were called for), it should also look for it at the correct position in the input (that is the global position of the bullet in the symbol we were called for). Therefore we initialize the new `rul_pos` to that `glb_pos`. If the • is at the far left of the new state, then its `rul_pos` must, of course, match its `glb_pos`, which is why the new `glb_pos` is the same as the old one.

7.2.5 The Scanner

The SCANNER takes care of adding states to the chart, corresponding to the input. If a state has a nonterminal symbol to the right of its •, then the SCANNER is called to see if the symbol this state expects can really be found in the input at that position. If so, it adds a new state that can then be used by the COMPLETER to advance the state that is "looking for" that nonterminal.

⟨ the SCANNER 209 ⟩ ≡

```
int
scanner (struct state *st)
{
  char cursym;

  ⟨ do debug-output for the scanner 220b ⟩

  cursym = st->rule_right[st->bul_pos + 1];

  if ((st->glb_pos->next->word != NULL)
      && ((st->glb_pos->next->word->morph >> 28) == (cursym & 0x7F)))
    {
      ⟨ add the current input as a new state 210a ⟩
    }

  return 0;
}◇
```

Macro referenced in 228.

⟨ add the current input as a new state 210a ⟩ ≡

```
struct state *newst;

newst = (struct state *) immalloc (sizeof (struct state));
newst->rule_left = cursym;
newst->rule_right = NULL;

newst->bul_pos = 1;
newst->glb_pos = st->glb_pos->next;
newst->rul_pos = st->glb_pos;
```

⟨ initialize the data-structures needed for the parse-forest (210b `newst`) 222a, ... ⟩

⟨ manipulate the data-structures needed for the parse-forest according to the scan 227a ⟩

```
enqueue (newst, st->glb_pos->next);◇
```

Macro referenced in 209.

In this case the position of the • is the far right end (which is position `1`, given that states added by the SCANNER take the form $POS \rightarrow WORD\bullet$). Therefore the rule position must be the same as in the state we were called for (`st->glb_pos`) and given that the • is one word further in the input, the global position of the new state must be the one following the global position of the state we were called for (`st->glb_pos->next`).

7.2.6 The Completer

The COMPLETER connects the top-down predictions from the bottom up, anchored by scanned states. It is called for every complete state ($SYM \rightarrow \gamma\bullet$) and looks through all the states "looking for" the newly completed symbol SYM ($A \rightarrow \alpha \bullet SYM \beta$), and adds a new state, where the • is advanced over the symbol

SYM $(A \rightarrow \alpha SYM \bullet \beta)$.

"Looking through" all the states of the form $A \rightarrow \alpha \bullet SYM\beta$ would be quite time-consuming, if we implemented it as an iteration over all the states in the chart matching them against this form. That's why we use another index, a linked list pointing us to all such states.

⟨ indices for the states in the chart 211a ⟩ ≡

```
struct statelist *idx_ntsym_rod[NUM_NONTERM];
struct statelist *idx_tsym_rod[NUM_TERM];◇
```

Macro defined by 202a, 211a.
Macro referenced in 193.

Of course, somehow, we have to create this list, and the right place to do so is the ENQUEUE-routine.

⟨ add the state to the chart 211b ⟩ ≡

```
if ((news->rule_right != NULL)
    && (news->rule_right[0] > news->bul_pos))
  {
    char sym = news->rule_right[news->bul_pos + 1];
    if (IS_POS (sym))
      add_to_listhead (&(chart->idx_tsym_rod[AS_POS (sym)]), news);
    else
      add_to_listhead (&(chart->idx_ntsym_rod[(unsigned int) sym]), news);
  }◇
```

Macro defined by 206, 211b.
Macro referenced in 205.

This way the COMPLETER can iterate through all the states that have the form $A \rightarrow \alpha \bullet SYM\beta$ in the chart, by simply iterating through the list headed by `chart->idx_tsym_rod[SYM]` for terminal symbols and `chart->idx_ntsym_rod[SYM]` for non-terminal symbols.

⟨ the COMPLETER 212 ⟩ ≡

```
int
completer (struct state *st)
{
  char sym = st->rule_left;
  struct statelist *cur;

  ⟨ do debug-output for the completer 220c ⟩

  if (IS_POS (sym))
    cur = st->rul_pos->idx_tsym_rod[AS_POS (sym)];
  else
    cur = st->rul_pos->idx_ntsym_rod[(unsigned int) sym];

  for (; cur != NULL; cur = cur->next)
    if (cur->st->glb_pos == st->rul_pos)
      {
        ⟨ complete the state 213 ⟩
      }

  return 0;
}◇
```

Macro referenced in 228.

Additionally to the basic iteration-pattern this runtime-structure features a check, that validates whether the complete state and the state to be completed match with respect to their positions in the input.

⟨ complete the state 213 ⟩ ≡

```
struct state *newst;

newst = (struct state *) immalloc (sizeof (struct state));

newst->rule_left = cur->st->rule_left;
newst->rule_right = cur->st->rule_right;

newst->bul_pos = cur->st->bul_pos + 1;
newst->glb_pos = st->glb_pos;
newst->rul_pos = cur->st->rul_pos;
```

⟨ prepare the data-structures needed for the parse-forest for completion 222b, ... ⟩

```
newst = enqueue (newst, st->glb_pos);
```

⟨ manipulate the data-structures needed for the parse-forest according to the completion 223 ⟩◇

Macro referenced in 212.

To advance the • in a state, the rule is simply carried over to the new state, and the rule-position also stays the same. The position of the • is incremented, and the new position in the input, corresponding to that new position of the • (its `glb_pos`) is the same as in the complete state (the state we were called for).

7.2.7 User-Output

Usually we wouldn't want our parser to operate completely silently. Doing human-readable output at important places in the code makes debugging significantly easier.

Doing this output is very important for the parser to be useful, but algorithmically trivial. The reader not interested in this kind of detail should feel free

to skip this section. It provides no information essential to the understanding of the rest of the program.

In order not to confront the user with too much trash-output, the parser provides two compile-time-options:

⟨ compile-time-options for debugging 214a ⟩ ≡

```
#define DEBUG_MODE
#define ONLY_LAST◇
```

Macro never referenced.

Here DEBUG_MODE specified that we want our program to do user-output. If we didn't define DEBUG_MODE, the parser would operate completely silently. There are two versions of debug-mode. In plain debug-mode (when DEBUG_MODE is defined and ONLY_LAST isn't) the parser does output for every state in the chart. This is usually too much senseless output, unless we are really looking for a bug in the code. That's why the option ONLY_LAST can be used, to tell the parser to do output for every state in a chart only for the last chart before it discovers a syntax-error. This output is very useful when doing grammar-engineering. As long as there is no syntax-error, words are printed on the screen as they are read.

The first problem, for forming human-readable output, is the "pointer-jungle" that makes up the data-structures for parsing. Pointers allow for very efficient processing, but they are unreadable by humans. That's why each chart is assigned a numeric index. (This index is used *only* for output-purposes).

⟨ numeric index for debugging-purposes 214b ⟩ ≡

```
#ifdef DEBUG_MODE
  char debidx;
#endif◇
```

Macro referenced in 193.

Of course we also have to assign a value to this index. We do that in the main loop by using a counter-variable

⟨ declarations for the `parse`-routine 215a ⟩ ≡

```
#ifdef DEBUG_MODE
  int w = 0;
#endif◇
```

Macro defined by 195a, 196, 199, 215a.
Macro referenced in 192a.

and assigning the chart its value

⟨ assign an index for debugging-mode 215b ⟩ ≡

```
#ifdef DEBUG_MODE
  cur->debidx = w++;
#endif◇
```

Macro referenced in 197a.

After reading a word we also print it on the screen if we are in the only-last-mode.

⟨ print the current word for debugging 215c ⟩ ≡

```
#ifdef DEBUG_MODE
#ifdef ONLY_LAST
  if (cur->word != NULL)
    printf("%s ", cur->word->cha);
#endif
#endif◇
```

Macro referenced in 197a.

Then, for each state in the chart, we call the `debug_out`-routine if we're in plain debug-mode.

⟨ do debug-output of the current state 216 ⟩ ≡

```
#ifdef DEBUG_MODE
#ifndef ONLY_LAST
  debug_out (curit->st);
#endif
#endif◇
```

Macro referenced in 200.

We also call this routine for every state in a chart, where we discovered a syntax-error. We do this only if we are in only-last-mode. (Otherwise we would already have printed them "along the way"). In this case, we have to start a new iteration over the states in the current chart, and we indent the output, marking states with a `"*"` that would be handled by the SCANNER. This makes grammar-engineering a lot easier, because it gives information about what POS the parser would have expected at this position in the input.

⟨ do debug-output for the syntax-error 217 ⟩ ≡

```
#ifdef DEBUG_MODE
#ifdef ONLY_LAST

printf ("\n");
for (curit = cur->itemtop; curit != NULL; curit = curit->next)
  {
    if (((!(curit->st->rule_right == NULL))
        && (curit->st->rule_right[0] > curit->st->bul_pos))
   && (IS_POS (curit->st->rule_right[curit->st->bul_pos + 1]))
      )
      printf ("* ");
    else
      printf ("  ");

    debug_out (curit->st);
  }

#endif
#endif◇
```

Macro referenced in 201b.

The implementation of `debug_out` looks like this:

⟨ define the routine for printing a state 218 ⟩ ≡

```
#ifdef DEBUG_MODE
void
debug_out (struct state *st)
{
  int i;

  symout (st->rule_left, stdout);
  printf ("-> ");

  if (st->rule_right != NULL)
    for (i = 1; i <= st->rule_right[0]; i++)
      {
        symout (st->rule_right[i], stdout);
        printf (" ");
      }
  else
    printf ("%s ", st->glb_pos->word->cha);

  printf ("[%d,%d,%d]\n", st->rul_pos->debidx, st->glb_pos->debidx,
          st->bul_pos);
}
#endif◇
```

Macro never referenced.

This routine is quite simple. We call `sym_out` for each symbol, throwing in the string `"->"` to indicate the →, and print the debug-indices of rule-position and global position (the `rul_pos` and `glb_pos`) as well as the position of the • (`bul_pos`)

⟨ define the routine for printing a symbol 219a ⟩ ≡

```
void
symout (char sym, FILE * f)
{
  if (sym == NTGS_LAMBDA)
    fprintf (f, "\\lambda ");
  else if (IS_POS (sym))
    fprintf (f, "%s", debsym_pos[AS_POS (sym)]);
  else
    fprintf (f, "%s", debsym_nt[(unsigned int) sym]);
}◇
```

Macro never referenced.

The implementation is, again, quite simple. Note that this routine is defined, also when we are not in debug-mode, and has a parameter for the output-file. This way output-modules (such as the module that does the conversion to TEX, and the one that does the conversion to PROLOG) can use it.

The last piece of "output-cosmetics" is the string `"--"` we print when we are finished with a chart, so the user sees where a chart ends and a new one begins and a message indicating which part of the program was used to interpret a state:

⟨ do debug-output for the finished chart 219b ⟩ ≡

```
#ifdef DEBUG_MODE
#ifndef ONLY_LAST
  printf ("--\n");
#endif
#endif◇
```

Macro referenced in 197a.

⟨ do debug-output for the predictor 220a ⟩ ≡

```
#ifdef DEBUG_MODE
#ifndef ONLY_LAST
  printf ("CALLED PREDICTOR\n");
#endif
#endif◇
```

Macro referenced in 207.

⟨ do debug-output for the scanner 220b ⟩ ≡

```
#ifdef DEBUG_MODE
#ifndef ONLY_LAST
  printf ("CALLED SCANNER\n");
#endif
#endif◇
```

Macro referenced in 209.

⟨ do debug-output for the completer 220c ⟩ ≡

```
#ifdef DEBUG_MODE
#ifndef ONLY_LAST
  printf ("CALLED COMPLETER\n");
#endif
#endif◇
```

Macro referenced in 212.

7.3 Building a Parse-Forest

The program we've been considering so far is not yet a parser, but rather a recognizer. The runtime-structure of the recognizer allows interpretation of the input, while recognizing it. The kind of interpretation we want to do in this program is build up a data-structure similar to what Tomita (1987) calls a "packed forest".

Recall that the biggest problem in interpreting natural language is that of

ambiguity. A parser dealing with an ambiguous grammar has to build up a data-structure similar to a parse-tree, but effectively dealing with concurrent interpretations of a single node.

We already introduced the notion of the backpointer in the first part, when we were explaining the augmentations neccessary to provide a way for extracting a parse-forest from the chart. The basic idea behind it was to add a pointer corresponding to a symbol in a state pointing to the state, that caused the • to be advanced over the symbol.

7.3.1 Backpointers

⟨ pointers for the parse-forest 221 ⟩ ≡

```
struct statelist *backptr[MAX_RULLEN];
```

◇

Macro defined by 221, 225.
Macro referenced in 194b.

This array is indexed, according to symbol-positions in the rules. A state for the rule $S \rightarrow NP\ VP$ would therefore carry the backpointer indicating the state the NP was resolved by in `st->backptr[0]` and the backpointer indicating the state the VP was resolved by in `st->backptr[1]`.

The type of this pointer is `struct statelist *` instead of `struct state *` because there can be more than one state, that could cause a symbol to be resolved. Therefore we use this wrapper-structure for building up a linked list of states.

This also means, that we have to initialize the elements to `NULL`, so we can use standard linked-list-algorithms.

⟨ initialize the data-structures needed for the parse-forest 222a ⟩ ≡

```
bzero (@1->backptr, sizeof (@1->backptr));
```

◇

Macro defined by 222a, 226b.
Macro referenced in 195b, 208a, 210a.

Note that this is where ambiguity enters our data-structure and makes it a parse-forest, rather than a parse-tree, because these lists are used to enlist alternative interpretations for a single symbol.

7.3.2 Providing Backpointers during Completion

It's the COMPLETER's job to provide backpointers, since the completer is the only routine that "knows" which state causes which other state's • to be advanced over a symbol.

Recall that `cur->st` is the state which is looking for a symbol the COMPLETER was called for. Of course the backpointers of the state, created by the COMPLETER (`newst`) first have to be copied over from the old state.

⟨ prepare the data-structures needed for the parse-forest for completion 222b ⟩ ≡

```
memcpy (newst->backptr, cur->st->backptr, sizeof (newst->backptr));
```

◇

Macro defined by 222b, 226c.
Macro referenced in 213.

Then we enqueue the newly created state to the chart, if it isn't already there. We call the `enqueue`-routine in such a way that `newst` either stays untouched, in case it gets enqueued to the chart, or `newst` is set to the state that is found to be equal, but already in the chart. In each case it is the `newst`-state where we have to add a new backpointer, and this can be done with a simple call to `add_to_listhead`, which is an implementation of the well-known algorithm adding an element to the head of a linked list.

⟨ manipulate the data-structures needed for the parse-forest according to the completion 223 ⟩ ≡

```
add_to_listhead (&(newst->backptr[(unsigned int) cur->st->bul_pos]), st);◇
```

Macro referenced in 213.

7.3.3 Member-Span

Augmenting the recognizer in such a way also makes a change in the routine neccessary that checks two states for equality, when deciding whether a new state should be entered into the chart.

Consider a state s like $S \rightarrow \bullet NPVP$ that should be advanced. Let's assume that there are two states s_1 and s_2 of the form $NP \rightarrow \alpha\bullet$, such that s_2's global position is further in the input, than s_1's (say s_1's global position is 2, and s_2's is 3, when speaking in terms of numeric global-positions, as in the first part of this paper), and they both have the same rule-position as the global position of s. Completing s with s_1 would lead to a state s_3, and completing s with s_2 would lead to a different state s_4. These are, of course, two distinct states because their global positions don't match, but both of the states s_3 and s_4 would have the form $S \rightarrow NP \bullet VP$.

Now let's further assume there are two states s_5 and s_6, that both have the form $VP \rightarrow \beta\bullet$, such that s_6's rule position matches s_2's global position (and therefore also s_4's), and s_5's rule position matches s_1's global position (and therefore also s_3's) and s_5's and s_6's global positions are equal. (This situation might seem quite abstract, but it is in fact a very common member of the "family" of ambiguities, the most prominent member of which is known as "dangling else").

Let's have a look at how our parser would deal with this situation. It would come across s_3, and complete it with s_5. Let's call the resulting state s_7. This state would now be added to the chart, given that there is not already such a state. Then it would come across s_4, and would complete it with s_6, resulting in the state s_8.

We might now easily agree on the fact that s_7 and s_8 must be two distinct nodes in the resulting parse-forest, since they "split up" their rule $S \rightarrow NP \mid VP$ at different places in the input, and since they have completely different back-pointers.

However the parser we've constructed so far wouldn't agree: s_7 and s_8 both have the same rule, the same rule position, the same global position and the same bullet position. That's why s_8 wouldn't be added to the chart, since it would be considered to be equal to the state s_7, which is already in the chart.

This goes along with the intuition that one might have, that when adding new information to the data-structure making up a "state" (in our case the lists of backpointers) this new information must also be considered a distinguishing feature in the equality-check.

It's important to understand what this means for our recognizer. Building up a parse-tree is no longer a task that can go "along the way" of recognition, but it does, in fact, influence the runtime-structure of the recognition. States that wouldn't make it into the chart, if it wasn't for building up the parse-tree, have to be added. At least this is how our prototype works.

Considering the backpointer-lists as distinguishing features in the equality-check is, again, a challenge if we want to do it in a performant way. Checking two lists of backpointers is a rather time-consuming operation. Therefore we make use of the fact that a backpointer-list can be uniquely identified given the symbol it resolves, the global position of the spot immediately to the left of the symbol and the global position immediately to the right of the symbol. The check for the position each of these symbols "starts at" must be implemented using a new data-structure, which we call "member span", the other checks are then given implicitly.

⟨ pointers for the parse-forest 225 ⟩ ≡

```
struct chart *memb_span[MAX_RULLEN];◇
```

Macro defined by 221, 225.
Macro referenced in 194b.

In this array we keep the rule positions common to all backpointers in the list at the corresponding index. (Therefore, if we have a state with the rule $S \rightarrow NP\ VP$, we would find the backpointers for resolving the VP at `backptr[1]`, according to the rule we've already mentioned. The VP is the second symbol, and indexing begins with 0. In `memb_span[1]` we would find the rule-position common to all the states in `backptr[1]`.

I'd like to elaborate a little bit on this subject, since it is one of the trickier parts in this program. Why is checking this array of start-positions sufficient for knowing that all backpointer-lists are equal? As I've already mentioned, a backpointer-list is uniquely identified by the symbol it resolves, and the positions where the states resolving the symbol start, and where they end. This seems logical (since we can't use two different interpretations that expand different symbols for interpreting one symbol in the original state and two different interpretations for one symbol must begin at the same place in the input, and end at the same place, since otherwise we could skip symbols, or multiply use the same symbols), but how do we know that our program now exhibits this feature?

The check for the symbol is the simplest. It is implicitly stated in the check for equality of the grammar-rules. If two grammar-rules are equal, then all the symbols in the state, and their order, must also be equal.

The check for the start positions is the only one we have to do explicitly, by checking the start positions saved in the `memb_span`-array. The check for the end positions is then given implicitly, since one symbol ends where the next one begins (and we would have to check all the start-positions anyway), leaving only the symbol the • is advanced over, without a check for the end position, and this

check is implicitly given by checking the global position of the new state.

Ergo, the only augmentation neccessary in the `compare`-routine is

⟨ check data-structures for the parse-forest 226a ⟩ ≡

```
if ((rc = memcmp (s1->memb_span, s2->memb_span, sizeof (s1->memb_span))))
  return rc;◇
```

Macro referenced in 203.

The use of `memcmp` might seem a bit oversimplified here, and a more effective implementation would indeed be possible, because actually we have to check only the symbols to the left of the •, but since the maximum length of a rule is 3 in our sample-grammar, and usually not much bigger, this is neglectable.

Of course we have to initialize the array, by `bzero`ing it, so the undefined values don't cause the comparison to fail.

⟨ initialize the data-structures needed for the parse-forest 226b ⟩ ≡

```
bzero (@1->memb_span, sizeof (@1->memb_span));◇
```

Macro defined by 222a, 226b.
Macro referenced in 195b, 208a, 210a.

7.3.4 Providing Member-Span-Data during Completion

This works similarly to the way we update backpointers. First `memb_span` is copied over, then the new backpointer is written to the array.

⟨ prepare the data-structures needed for the parse-forest for completion 226c ⟩ ≡

```
memcpy (newst->memb_span, cur->st->memb_span,
        sizeof (newst->memb_span));
newst->memb_span[newst->bul_pos - 1] = newst->glb_pos;◇
```

Macro defined by 222b, 226c.
Macro referenced in 213.

This time we have to apply our augmentation also to the SCANNER:

⟨ manipulate the data-structures needed for the parse-forest according to the scan 227a ⟩ ≡

```
newst->memb_span[newst->bul_pos - 1] = newst->glb_pos;◇
```

Macro referenced in 210a.

⟨ general purpose pointer for the chart 227b ⟩ ≡

```
void *info;◇
```

Macro referenced in 193.

⟨ general purpose pointer for the state 227c ⟩ ≡

```
void *info;◇
```

Macro referenced in 194b.

"parse.c" 228 ≡

```
#include <stdio.h>
#include <stdlib.h>

#include <string.h>
#include <strings.h>

#include "config.h"

#include "grammar.h"
#include "parse.h"
#include "misc.h"

#include "debug.h"

#include "premorph.h"
```

⟨ enqueue-implementation 205 ⟩

⟨ the Completer 212 ⟩

⟨ the Predictor 207 ⟩

⟨ the Scanner 209 ⟩

⟨ implementation for the main routine 192a ⟩

◇

Discussion

In grammar-school our headmaster used to promote his Ancient-Greek-course by telling students how generally appliable linguistic structures were, when it came to everyday problem-solving. Today I understand what he meant, but when I first set out to build this natural language understander, I didn't quite realize that I was actually trying to build the almost mythical "general problem solver". Needless to say I did not succeed in building the HAL-like computer that would revolutionize artificial intelligence.

One of the greatest challenges in authoring this kind of work was to steer a middle-course between generality and speciality. Generality, on one side of the didactic spectrum, could provide the reader with a sophisticated understanding of all the techniques used and how they could work together to make up any NLP-system he has in mind, but would never provide information detailed enough to allow him to sit down and hack it into a computer. Speciality, on the other side of the spectrum, would leave the reader with a very detailed understanding of a prototype that executes a very narrowly defined task, but would never provide information neccessary to take these concepts and apply them to any other but this narrowly defined task. The danger of overwhelming the reader with tons of very specific but loosely related facts, is opposed to that of getting lost in vaguely formulated ideas and concepts that seem to have no more practical impact.

The compromise that I chose was to stay on the didactic, more general and, hopefully, more interesting side of this spectrum, but to pick out the two tech-

nically most challenging aspects of creating this prototype and to present them in full detail down to the sourcecode-level. This is why topics discussed in this paper span the whole spectrum from philosophical considerations about the true nature of sense and meaning to the question of whether a double-word is written to memory in AT&T-Assembler with the `mov`- or the `movl`-mnemonic.

Of course this strategy has some negative impacts on the prototype. While the prototype is suited very well in terms of didactic considerations applied to a system that was never meant to be used for anything else but for didactic purposes and laboratory conditions, it does have some flaws, in terms of design-considerations applied to a more pragmatic, business-like approach to software-engineering.

The parser and the FST-Tools, the parts that were described here in greater detail are very general, very flexible and highly performant, but none of this is actually made use of in the whole system, since these parts are connected mainly by "glue code", which simply serves the purpose of integrating everything into a system carrying out a real-world task that makes everything transparent and shows *that* (rather than *how*) these core-components can be used for building practical systems.

In fact, the prototype is far away from a practical system, since this "glue code" limits the problem domain and the linguistic data too much as to actually make it useful under any but laboratory-conditions, which is the only thing I regret now.

These are some of the major limitations:

- The dictionary contains exactly the words that are really in the corpus, and the grammar contains only the rules that are really used by sentences in the corpus.
- There is no way the system can learn new words or grammar rules.
- Input is not at all disambiguated. All interpretations are considered equally

probable, no statistical methods are used. There is not even a way for the system to choose one of the interpretations as a means of disambiguation. If a given fact is derivable from any one of the interpretations, it can be used for semantic processing.

- The grammar works only on parts of speech. Our parser cannot use grammars dealing with features or semantic attachments.
- The syntactic aspects of the grammar and the syntactic parser is completely decoupled from the semantic aspects of the grammar and semantic processing.
- The "glue code" bridging the semantic and syntactic layers doesn't use syntax-driven semantic analysis, but rather something like "information retrieval in a parse-forest".
- Morphological analysis of word-forms is "hardcoded". It doesn't use linguistic rules as a basis for morphological analysis, since I couldn't get hold of suited linguistic material.

Most of these flaws in the prototype are due to a major misunderstanding I had when I started programming it. I assumed that I could use linguistic data, such as the lexical database WordNet or the English Resource Grammar, as a basis for this project's linguistic knowledge-base, which turned out to be too difficult.

Given more time, I would have been able to use more of the great potential of these core-components. Building up a more complete dictionary and grammar and modelling a wider problem-domain would have made it a lot more "spectacular". It could then be a system talking to a human and answering questions about the conversation, it could be a robot controlled by natural-language or a talking car fighting for justice. Just about anything science fiction has ever dreamed of . . . Perhaps.

Bibliography

Backus, J. W. (1959), The syntax and semantics of the proposed international algebraic language of the Zurch ACM-GAMM Conference, *in* 'Information Processing: Proceedings of the International Conference on Information Processing, Paris', UNESCO, pp. 125–132.

Beckwith, R., Miller, G. A. & Tengi, R. (n.d.), 'Design and implementation of the WordNet lexical database and searching software', `http://www.cogsci.princeton.edu/~wn/5papers.ps`.

Chomsky, N. (1956), 'Three models for the description of language', *IRI Transactions on Information Theory* **2**(3), 113–124.

Copestake, A. (2002), *Implementing Typed Feature Structure Grammars*, CSLI Publications.

Earley, J. (1970), 'An efficient context-free parsing algorithm', *Communications of the ACM* **6**(8), 451–455.

Fellbaum, C. (n.d.), 'English verbs as a semantic net', `http://www.cogsci.princeton.edu/~wn/5papers.ps`.

Fellbaum, C., Gross, D. & Miller, K. (1993), 'Adjectives in WordNet', `http://www.cogsci.princeton.edu/~wn/5papers.ps`.

Jackson, P. C. (1985), *An Introduction to Artificial Intelligence*, Dover Publications, Incorporated.

Jurafsky, D. & Martin, J. H. (2000), *Speech and Language Processing*, Prentice Hall.

Lenat, D. B. (1995), Cyc: A large-scale investment in knowledge infrastructure, *in* 'Communications of the ACM', number 11 *in* '38', ACM.

Marcus, M. P., Santorini, B. & Marcinkiewicz, M. A. (1993), 'Building a large annotated corpus of English: The Penn treebank', *Computational Linguistics* **19**(2), 313–330.

McCarthy, J. (1958), Programs with common sense, *in* 'Teddington Conference on the Mechanization of Thought Processes'.

McCarthy, J. (1977), Epistemological problems of Artificial Intelligence, *in* 'International Joint Conference on Artificial Intelligence'.

McCarthy, J. (1987), 'Generality in Artificial Intelligence'.

McCarthy, J. (1989), Artificial Intelligence, Logic and formalizing common sense, *in* R. Thomason, ed., 'Philosophical Logic and Artificial Intelligence', Dordrecht; Kluwer Academic.

McCarthy, J. (1990), *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz*, Ablex.

McCarthy, J. & Hayes, P. J. (1969), 'Some philosophical problems from the standpoint of Artificial Intelligence', *Machine Intelligence*.

Miller, G. A. (1993), 'Nouns in WordNet: A lexical inheritance system', `http://www.cogsci.princeton.edu/~wn/5papers.ps`.

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D. & Miller, K. (1993), 'Introduction to WordNet: An on-line lexical database', `http://www.cogsci.princeton.edu/~wn/5papers.ps`.

Schank, R. (1971), *Intention, memory, and computer understanding*, Stanford: Stanford Art. Intell. Proj.

Sterling, L. & Shapiro, E. (1994), *The Art of Prolog*, Series in Logic Programming, second edition edn, MIT Press.

Tomita, M. (1987), 'An efficient augmented-context-free parsing algorithm', *Computational Linguistics* **13**(1-2), 31–46.

Weisler, S. E. & Milekic, S. (2000), *Theory of Language*, MIT Press.

Winograd, T. (1971), Procedures as a representation for data in a computer program for understanding natural language, Technical report, MIT.

List of Figures