Error Correction using DOP

Menno van Zaanen

Supervisor: Dick Grune
Vrije Universiteit
Amsterdam
1996–1997

vrije Universiteit amsterdam

Preface

This thesis is part of my graduation project at the department of Computer Science of the Vrije Universiteit in Amsterdam. I worked on this project from January 1997 until October 1997.

I would like to thank Dick Grune who was my graduation coordinator, Rens Bod, the writer of the DOP system, Ceriel Jacobs for explaining the details of LLgen, Remko Scha who told me about using DOP in a compiler and Koen Langendoen for proofreading this thesis. Also I want to thank Mila Groot with whom I had several discussions about the project. Finally, I would like to thank my parents for their years of support.

Abstract

A compiler is a program that translates computer code into something the computer is able to understand. It will try to structure the computer code. The result of this can be seen as a tree structure on top of the code.

The systems described in this thesis break these tree structures into pieces. These pieces are then sorted and counted. When some new code is offered for compilation, the systems try to build a new tree structure using the pieces of the previously encountered code. The pieces that have a higher count are more likely to be used in the structure. The compiler tries to build the new tree structure with pieces that have a high count.

When the code which is being compiled contains an error, the compiler does not know what to do, because the pieces that are used to build the structure of the code are pieces of code that do not contain errors. The systems described here use certain techniques to correct the code in a way that the pieces with the highest count can be used to build the structure.

Four systems will be described. The first system (DOP 2) tries to correct the code by changing certain symbols into other symbols after an error has been detected. The second system (DOPPER 1) corrects the code by inserting or deleting one symbol at a time after an error has been detected. The third system (DOPPER 2) corrects the code by inserting or deleting several symbols at a time after an error has been detected. The fourth system (DOPPER 3) starts building a structure all over again after an error has been found. This system corrects the errors by inserting and deleting several symbols at a time like in DOPPER 2.

DOP 2 does not do very well in correcting errors. DOP 2 corrects errors by replacement, whereas most of the errors found in the code are best corrected by inserting or deleting symbols. DOPPER 1 is able to correct some pieces of code very well, but since it only corrects one symbol at a time, it sometimes corrects errors in a strange way. DOPPER 2 does a much better job. It corrects almost all errors in a way that is satisfactory. DOPPER 3 does the best job. The major disadvantage (at this moment) is the big overhead in time and space.

This thesis is divided into two parts. The first part, *Basics*, describes the basics needed to understand the second part, *Implementation*, which explains the different systems.

The first part contains information about the ideas and techniques that are already described elsewhere in the literature, but are very important in the rest of the thesis.

The first part starts with explaining the basics of parsing. After that, the Data Oriented Parsing (DOP) framework is sketched. The next section briefly describes how compilers work. That section is followed by a section about different error handling techniques. Finally a conclusion about these sections is given.

These sections are the basics of the thesis, so it could be that the reader is already familiar with one or more parts. These parts can then be skipped. For example, people already familiar with the basics of parsing can skip section 2, Parsing, whereas someone from the computational linguistics field who is already familiar with the DOP framework can safely skip section 3, Data Oriented Parsing.

The second part contains information about the actual functioning of the several systems. Everything in this part of the thesis is new, not present in other literature.

The second part starts out with giving a detailed description of the goal of the project, followed by sections 8...10 containing information about the implementation. These sections are followed by a section explaining the way the errors are corrected. Next, there is a section describing several possible implementations of the error correcting algorithms. Last, some conclusions are given.

If the reader is interested in the computer implementation, sections 8 up to and including section 10 are important. Readers interested in the actual error correcting algorithms should especially read section 11 and section 12.

Contents

I Basics 1

1 Introduction 3

2 Parsing 5

2.1 Introduction to the basics . . . 5

2.2 Kinds of parsers . . . 8

2.2.1 Top-down versus bottom-up parsers . . . 8

2.2.2 Tabular parsers . . . 10

3 Data Oriented Parsing 15

3.1 Trees . . . 16

3.2 DOP framework . . . 17

3.2.1 Sentence analyses . . . 19

3.2.2 Sub-analyses . . . 19

3.2.3 Combination operations . . . 19

3.2.4 Combination probabilities . . . 19

3.3 Parsing using DOP . . . 20

3.4 Predecessors of DOP . . . 22

4 Compilers 25

4.1 LLgen . . . 26

5 Error handling 29

5.1 Introduction to syntax error handling . . . 29

5.2 Different types of error handling methods . . . 30

5.2.1 Error detection . . . 30

5.2.2 Ad hoc error handling . . . 31

5.2.3 Local error handling . . . 31

5.2.4 Regional error handling . . . 31

5.2.5 Global error handling . . . 32

6 Conclusion 33

6.1 Parser . . . 33

6.2 DOP . . . 33

6.3 Compilers 33
6.4 Error handling 34

II Implementation **35**

7 Goal **37**

8 Programming languages **41**
8.1 Compiler 41
8.2 Implementation 41

9 Data Oriented Parsing **43**
9.1 Parsing 43
9.1.1 `subtree` class 43
9.1.2 `itemtree` class and `st_iterator` class 44
9.1.3 `edge` class 44
9.1.4 `chartstate` class 44
9.1.5 `chart` class 44
9.2 Corpus 45
9.2.1 Format of the corpus 45
9.2.2 Filling the corpus 46
9.3 Disambiguation 50
9.3.1 Monte Carlo disambiguation 52

10 LLgen **55**
10.1 `lexical` routine 56
10.2 `LLmessage` routine 58
10.3 `%onerror` routine 58

11 Error correction **61**
11.1 Handling errors in DOP 62
11.1.1 Handling an inserted symbol 62
11.1.2 Handling a deleted symbol 63
11.2 Correcting the input 63
11.3 Overview 65

12 Different implementations **67**
12.1 A naive way: DOP 2 67
12.1.1 Correcting the input: Replacement-only correction . . 67
12.1.2 Results 68
12.2 A good way: DOPPER 1 68
12.2.1 Correcting the input: Simple insert and delete 69
12.2.2 Results 69
12.3 A better way: DOPPER 2 70

12.3.1 Correcting the input: Multiple insert and delete . . . 70
12.3.2 Results 71
12.4 The best way: DOPPER 3 71
12.4.1 Correcting the input: Correct and restart 71
12.4.2 Results 72
12.5 Results 72
12.5.1 Examples 72
12.5.2 Practical results 76
12.6 Insertion and deletion probabilities 76

13 Conclusion 79
13.1 Goal 79
13.2 Error-correction 79
13.3 Selecting the most probable corrections 80
13.4 Different implementations 80
13.5 Comparison 81

A Computer implementation 83
A.1 C-compiler 83
A.2 `-e` flag 83
A.3 `-f` flag 84
A.4 `-c` flag 84
A.5 Class structure 84

B Example parse 87
B.1 Program code 87
B.2 Statesets 87
B.3 List of corrections 90
B.4 Most probable parse 90
B.5 Corrected program code 92

References 93

Part I

Basics

Chapter 1

Introduction

This part describes the notions and techniques necessary to follow the rest of the thesis.

First, we will start out explaining the basics of parsing, including what parsing is, how it works and what different types of parsers exist. Next, an explanation of Data Oriented Parsing (DOP) will be given, because this framework will be used in the rest of the thesis. We will describe some of the ideas behind DOP and show how it functions. Following DOP we will give a short description of what compilers are and we will describe the parts that make up a compiler. Then error handling in parsers will be treated. We will describe different types of error handling and their advantages and disadvantages.

Chapter 2

Parsing

2.1 Introduction to the basics

According to [GJ95] parsing is:

> the process of structuring a linear representation in accordance with a given grammar.

On the one hand, this definition may contain some concepts that are not completely clear. On the other hand, this definition is too general to be of use for us. So we will explain and refine parts of this definition.

A lot of things are linear representations. For example, a sentence is a linear representation: a list of words; music can be seen as a linear representation: a list of notes; DNA can be seen as a linear representation: a list of characters describing the DNA. Although here we will concern ourselves with computer code as the linear representation, most conclusions also apply to other kinds of linear representations.

The linear representation is often subject to restrictions. Just any list of words does not have to form a correct sentence. An instance of a linear representation is built using certain rules. These rules are defined by the grammar.

To make this more precise, we define a grammar as follows:

Definition 2.1 (Grammar) *A grammar* G *is defined as a quadruple*

$$\mathsf{G} = (V_N, V_T, \mathsf{S}, P)$$

where

V_N *is a non-empty, finite set of* non-terminal *symbols,*

V_T *is a non-empty, finite set of* terminal *symbols,*

$\mathsf{S} \in V_N$ *is a special symbol called the* start *symbol,*

P is a non-empty, finite set of productions.

Terminal symbols are symbols that are used in the input. We use lowercase words, characters in the beginning of the alphabet or punctuation marks to denote them, for example: string, a or ';'. A sequence of terminal symbols is denoted by lowercase characters at the end of the alphabet, for example: w or z. A special symbol is ε, which denotes the empty sequence (zero terminals).

Non-terminal symbols do not occur in the input, but are used to structure the input. We use uppercase words or characters in the beginning of the alphabet to denote them, for example: PROGRAM, A or EXPRESSION. Greek characters at the begin of the alphabet are used to denote a terminal or non-terminal, for example: α or β. Greek characters at the end of the alphabet are used to denote a possibly empty sequence of terminals or non-terminals, for example: ϕ or ψ.

Productions are the rules of the grammar. The productions define what sentences can be generated by the grammar.

Definition 2.2 (Productions) *A* production *can be seen as a pair (*A*,* ϕ*). The productions used here are always in the form:* A$\to\phi$

Definition 2.3 (Left-hand side and Right-hand side) *A production* A$\to\phi$ *has* A *as its* left-hand side *and* ϕ *as its* right-hand side.

These grammars have the power of a context-free grammar, or a type 2 grammar in the Chomsky hierarchy.

The start symbol is a special, distinguished non-terminal.

Several operations are defined on terminals and non-terminals.

Definition 2.4 (Concatenation) $\alpha\beta$ *denotes the concatenation of* α *and* β.

Definition 2.5 (Repetition) α^* *denotes zero or more repetitions of* α.
α^+ *denotes one or more repetitions of* α.

Definition 2.6 (Length) *The length of a sentence* w*, written* |w|*, is defined as follows:*
$|\varepsilon| = 0$
$|\mathsf{a}| = 1$
$|\mathsf{ax}| = 1 + |\mathsf{x}|$

Definition 2.7 (Derivation and Sentential form) *Consider the production* $\mathsf{A}\rightarrow\phi$*. This means we may replace* A *with* ϕ*. We say that* A derives ϕ *or* ϕ *is* derived *from* A*. We write this as* $\mathsf{A}\Rightarrow\phi$*. If* $\mathsf{S}\Rightarrow\phi_1\Rightarrow\phi_2\Rightarrow\cdots\Rightarrow\phi_\mathsf{n}\Rightarrow\mathsf{w}$ *then* $\mathsf{S}\Rightarrow\phi_1\Rightarrow\phi_2\Rightarrow\cdots\Rightarrow\phi_\mathsf{n}\Rightarrow\mathsf{w}$ *is called a* derivation *of* w*. The strings* S*,* ϕ_1*,* ϕ_2*, ...,* ϕ_n*, which may contain non-terminals as well as terminals, are called* sentential forms *of the derivation.*

Definition 2.8 (Sentence) *A* sentence *is a linear representation* $\mathsf{w} \in V_T{}^*$ *if there exists a derivation based on grammar* G *that derives* w*.*

Definition 2.9 (Language) *The language generated by a grammar* G*,* $\mathcal{L}(\mathsf{G})$*, is the set of all sentences that can be generated by grammar* G*.*

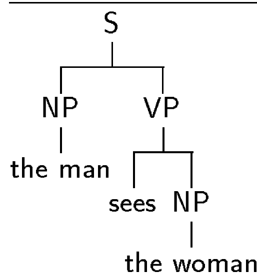
A sentence can be structured into a tree structure using the grammar. The start symbol is the root of a tree structure and the leaves of the tree are the terminal symbols in the same order as in the sentence. For example, if we have the grammar:

$$\mathsf{G} = (\{\mathsf{S}, \mathsf{VP}, \mathsf{NP}\}, \{\mathsf{the}, \mathsf{man}, \mathsf{sees}, \mathsf{woman}\}, \mathsf{S}, P)$$

where P is:
S→NP VP
NP→the man
NP→the woman
VP→sees NP
VP→walks
we can construct the structure of the sentence "the man sees the woman".

The structure of the sentence "the man sees the woman" can be visualized as the tree structure depicted in figure 2.1.

Figure 2.1 Tree structure of "the man sees the woman"



In this case it was easy to create the structure of the sentence, but this is not always the case. Sometimes no structure can be found, sometimes there is more than one.

Definition 2.10 (Ambiguous sentence) *A sentence* w *is ambiguous with regard to grammar* G *if there exist different tree structures for it using grammar* G*.*

Every sentence in the language described by the grammar has *at least one* corresponding tree structure. Ambiguous input sentences have multiple tree structures, input sentences not in the grammar have none.

2.2 Kinds of parsers

The previous section describes the parts necessary for parsing, but not *how* it should be done. This section provides a short introduction into parsing techniques, including a description of the parser we are going to use for error correction.

2.2.1 Top-down versus bottom-up parsers

The purpose of parsers is to generate a *tree* whose structure corresponds to the grammar and whose leaves match the input given.

Definition 2.11 (Tree structure of a production) *A production* $\mathsf{A}\rightarrow\phi$ *has a corresponding tree structure, with* A *as its head and* ϕ *as its leaves.*

There are several ways to get from the input sentence to a corresponding tree structure. These can be roughly divided into two categories, top-down methods and bottom-up methods. We will describe both of them here briefly.

Top-down methods

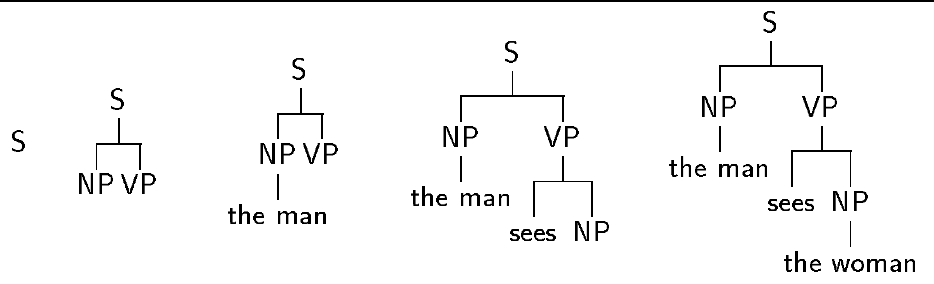
When trying to find a tree structure of some input, it is possible to start at the root of the tree[1]. This method is called *top-down parsing.*

The idea of top-down parsing is, that if we want to have a tree structure of an input sentence, the root of the tree has to be the start symbol. The parser selects the "correct" grammar rule, with the start symbol as its left hand side. We now have a part of the "final" tree structure, but large parts are still missing. The leaves of the tree at that point form exactly a sentential form as defined by definition 2.7. These missing parts need to be filled in, so we try to rewrite non-terminals until all parts are filled in and the leaves of the tree match the non-terminals in the input sentence. See for an example figure 2.2.

The method just sketched is non-deterministic, since the algorithm itself needs to choose the "correct" grammar rule when rewriting a non-terminal. In order to implement a top down method, the non-deterministic part has to be replaced by a deterministic version. What is important, is that the algorithm starts with the start symbol and parses *from the top, downward* to the input sentence.

[1] For some vague reason, in computer science the root of a tree structure is always at the top of the structure.

Figure 2.2 A top-down parse

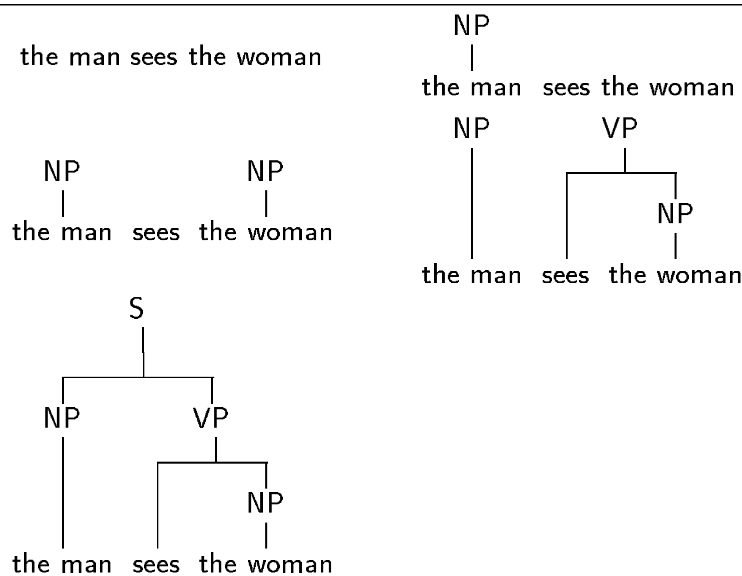


Bottom-up methods

The bottom-up methods work almost the same as the top-down methods, only the parsing starts from the input sentence. The idea behind the bottom-up methods is that we know the input and just need to construct a tree structure on top of that.

The bottom-up methods select grammar rules, such that a part of the input matches the right-hand side of a production rule P. A tree can then be constructed on top of this part with P as the top node. One says that part of the input *is reduced to* the root node of P. This rewriting "changes" the input, so that another grammar rule may be used. The algorithm is finished when a tree structure has been built with the start symbol at the root and the input sentence at the leaves. See for an example figure 2.3.

Figure 2.3 A bottom-up parse



The method just described is again non-deterministic and the non-deterministic part again involves the selection of the "correct" grammar rule.

However, this method differs from the top-down methods in that it starts with the input sentence and tries to match grammar rules with the input. This way the tree structure is built from bottom (the input sentence) to the top (the start symbol).

2.2.2 Tabular parsers

Most parsers try to find *one* tree structure given the input sentence. When we need *all* tree structures of a sentence, we need a parser that generates all possible tree structures. There exist parsers that generate all possible tree structures given an input sentence and put them in a convenient, compact representation. A compact representation is required, since an input sentence may have exponentially many or even infinitely many tree structures. It is then convenient to have these tree structures in a compact representation.

We call these parsers tabular parsers, because the compact representation can be viewed as a table. Examples of this kind of parser are the "chart parser", "Earley parser", "Tomita's parser" and "CYK parser". Some of these are similar methods, but use different compact representations, which can then be transformed into the other.

The parser we are going to describe here is the Earley parser. It is this parser which we will be using for the error correction.

The Earley parser

The Earley parser is a top-down restricted bottom-up parser. The parser tries to rewrite the start symbol by trying all possible grammar rules that rewrite it. It then tries to rewrite all of these grammar rules, selectively guided by the input. This is done until the input is matched with the leaves of the grammar rules. When the input is parsed, a parse can be generated, from the bottom working up.

The generation of the compact representation of the Earley parser is top-down, but the computation of a parse from that compact representation is done bottom-up, hence the name top-down restricted bottom-up.

To keep track of what part of a grammar rule is already rewritten, Earley in [Ear70] introduces the notion of an item. This is a grammar rule with a dot between two adjacent tokens in its right-hand side. Everything left of the dot has already been recognized and everything right of the dot still needs to be recognized. If we have for example the grammar rule:

A→aB

and we have not read any non-terminals yet, we represent this situation as:

A→•aB

and after reading the terminal a we have:

A→a•B

Earley now introduces states, a state is an item and a pointer[2], notation [item@*pointer*]. These states are in (numbered) statesets, and the pointer points to a stateset. The compact representation is the list of statesets which are created while parsing.

Now that we know the concepts and notation, we give the parsing algorithm. The algorithm is given in algorithm 2.1.

Algorithm 2.1 Earley's parsing algorithm

Input
Context-free grammar $\mathsf{G} = (V_N, V_T, \mathsf{S}, P)$ and an input string $\mathsf{w} = \mathsf{a_1 a_2 \ldots a_n}$ in $V_T{}^*$.
Output
The list of statesets $I_0, I_1, \ldots, I_n$.
Method
First, we construct I_0 as follows:

1. Set I_0 to empty.

2. If $\mathsf{S} \rightarrow \phi$ is a production in P, add $[\mathsf{S} \rightarrow \bullet \phi @ 0]$ to I_0.

3. Execute the routine $\mathbf{R}(0)$.

We now construct I_j, having constructed $I_0, I_1, \ldots, I_{j-1}$.

4. Set I_j to empty.

5. For each $[\mathsf{B} \rightarrow \phi \bullet \mathsf{a} \psi @ i]$ in I_{j-1} such that $\mathsf{a} = \mathsf{a_j}$, add $[\mathsf{B} \rightarrow \phi \mathsf{a} \bullet \psi @ i]$ to I_j.

6. Execute the routine $\mathbf{R}(j)$.

The algorithm, then, is to construct I_j for $0 \leq j \leq n$.
The routine $\mathbf{R}$ is defined as follows:
Routine $\mathbf{R}(j)$: Perform the following steps until no new items can be added.

1. Let $[\mathsf{A} \rightarrow \tau \bullet @ i]$ be an item in I_j. Examine I_i for items of the form $[\mathsf{B} \rightarrow \phi \bullet \mathsf{A} \psi @ k]$. For each one found, we add $[\mathsf{B} \rightarrow \phi \mathsf{A} \bullet \psi @ k]$ to I_j.

2. Let $[\mathsf{A} \rightarrow \phi \bullet \mathsf{B} \psi @ i]$ be an item in I_j. For all $\mathsf{B} \rightarrow \tau$ in P, we add $[\mathsf{B} \rightarrow \bullet \tau @ j]$ to I_j.

---

[2]Earley uses lookahead, but because we use no lookahead, the states and the algorithm are a bit simpler.

To give an idea of how this algorithm works, figure 2.4 contains a small example.

Figure 2.4 A parse with an Earley parser

S→NP VP	DET→the
NP→DET N	N→man
VP→V NP	N→woman
DET→a	V→sees

Input sentence “The man sees a woman”

stateset 0, input ‘the’	stateset 1, input ‘man’	stateset 2, input ‘sees’
[S→•NP VP@*0*]	[DET→the•@*0*]	[N→man•@*1*]
[NP→•DET N@*0*]	[NP→DET•N@*0*]	[NP→DET N•@*0*]
[DET→•a@*0*]	[N→•man@*1*]	[S→NP•VP@*0*]
[DET→•the@*0*]	[N→•woman@*1*]	[VP→•V NP@*2*]
		[V→•sees@*2*]

stateset 3, input ‘a’	stateset 4, input ‘woman’	stateset 5
[V→sees•@*2*]	[DET→a•@*3*]	[N→woman•@*4*]
[VP→V•NP@*2*]	[NP→DET•N@*3*]	[NP→DET N•@*3*]
[NP→•DET N@*3*]	[N→•man@*4*]	[VP→V NP•@*2*]
[DET→•a@*3*]	[N→•woman@*4*]	[S→NP VP•@*0*]
[DET→•the@*3*]		

When we have a list of statesets, we would like to construct a tree structure from it. The algorithm which constructs a list of production numbers from a list of statesets is given in algorithm 2.2. This algorithm produces information that can be used to generate a single, arbitrary parse from the chart.

Algorithm 2.3 converts the list of production numbers into a parse in the form of a tree structure. The algorithm uses tree structures derived from a production.

See figure 2.5 for an example of an application of algorithm 2.3, based on the list of statesets generated in figure 2.4.

In figure 2.5 a list of productions was generated from figure 2.4. We have converted the production numbers to the corresponding productions, so it is clear what productions will be used to generate the tree structure. Finally, we show each step in the process of converting the productions (from the production numbers) to the tree structure.

Algorithm 2.2 Generate a parse in the form of a list of production numbers

Input

A (cycle-free) context-free grammar $\mathsf{G} = (V_N, V_T, \mathsf{S}, P)$ with the productions in P numbered from 1 to p, an input string $\mathsf{w} = \mathsf{a}_1\mathsf{a}_2 \ldots \mathsf{a}_n$ in $V_T{}^*$ and the list of statesets $I_0, I_1, \ldots, I_n$ for w.

Output

π, a parse for w, or an "error" message.

Method

We assume that π is a global variable and is initially empty. If no item of the form $[\mathsf{S}\rightarrow\phi\bullet@0]$ is in I_n, then w is not in $\mathcal{L}(\mathsf{G})$, so emit "error" and halt. Otherwise, execute the routine $\mathbf{R}([\mathsf{S}\rightarrow\phi\bullet@0], n)$ where the routine $\mathbf{R}$ is defined as follows:

Routine $\mathbf{R}([\mathsf{A}\rightarrow\psi\bullet@i], j)$:

1. Let π be the previous value of π followed by h, where h is the number of production $\mathsf{A}\rightarrow\psi$.
2. If $\psi = \mathsf{X}_1\mathsf{X}_2 \ldots \mathsf{X}_\mathsf{m}$, set $k = m$ and $l = j$.
3.
   - If $\mathsf{X}_\mathsf{k} \in V_T$, subtract 1 from both k and l.
   - If $\mathsf{X}_\mathsf{k} \in V_N$, find an item $[\mathsf{X}_\mathsf{k}\rightarrow\tau\bullet@r]$ in I_l for some r such that $[\mathsf{A}\rightarrow\mathsf{X}_1\mathsf{X}_2 \ldots \mathsf{X}_{\mathsf{k}-1}\bullet\mathsf{X}_\mathsf{k} \ldots \mathsf{X}_\mathsf{m}@i]$ is in I_r.
     Then execute $\mathbf{R}([\mathsf{X}_\mathsf{k}\rightarrow\tau\bullet@r], l)$. Subtract 1 from k and set $l = r$.
4. Repeat step (3) until $k = 0$. Return.

Algorithm 2.3 Convert a list of production numbers into a parse tree

Input

π, a list of production numbers as generated by algorithm 2.2.

Output

A parse tree `T`, derived from the list of production numbers π.

Method

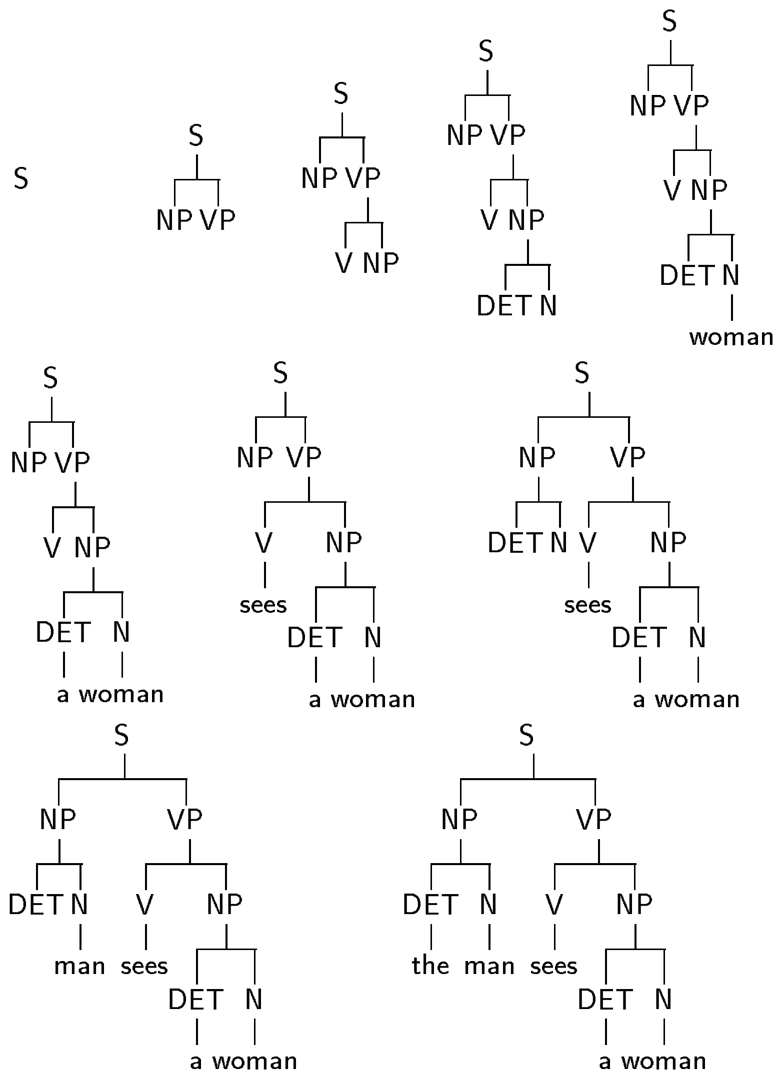
1. Set `T` to the start symbol, S.
2. Let $\pi = h\rho$, h is a number of a production.
3. Find in tree `T` the non-terminal leaf node A, that is the same as the head of the tree of production h, by searching from right to left. Replace A in tree `T` by the tree of production h.
4. If ρ is non-empty, goto (2) with $\pi = \rho$.

Figure 2.5 Generating a parse tree from a list of statesets

Generating list of production numbers (π)

S→NP VP
VP→V NP
NP→DET N
N→woman
DET→a
V→sees
NP→DET N
N→man
DET→the

Generating a parse tree



Chapter 3

Data Oriented Parsing

Parsing techniques can be used in many different types of fields. Examples are: computer science (computer programs), linguistics (sentences), biology (DNA patterns). The parsing method *Data Oriented Parsing* or *DOP* originates from the computational linguistics field.

Before explaining how the method works, we want to explain the ideas behind it. When explaining the ideas, we use the word *utterance* for a sentence in a natural language, as opposed to a sentence as we have defined before.

Ambiguity When we try to understand an utterance, normally we "see" only one or two different interpretations. Most linguistic theories, however, generate many more possible interpretations. So a useful natural language parser should choose exactly that interpretation (out of all which have been generated) the natural language user chooses as well. By definition an interpretation is the most probable interpretation when the natural language user considers it to be *the* most probable interpretation.

History According to Bod in [Bod95]:

1. People register frequencies and differences in frequencies of previously perceived analyses.
2. People prefer analyses that have been experienced before: "old before new".
3. The preference for "old before new" is influenced by the frequency of occurrence of analyses: "more frequent before less frequent".

and he summarizes:

> The frequencies of previously perceived analyses bias the analysis of a new input.

So each language user has a "memory" of past language experience, a set of previously perceived analyses or a *corpus* of analyses. This corpus is a multi-set of sentences together with their interpretations. Based on this corpus, the language user selects the most probable interpretation out of all possible interpretations. It should be noted that the other interpretations are not *wrong*, their likelihood is just smaller than that of the most likely interpretation.

Statistics As we are already speaking about likelihood, or the more usual term probability, it is straightforward[1]to use statistics to choose the most probable interpretation. Using the frequency of previously perceived analyses, probabilities of interpretations of the utterance to be perceived are calculated, so the most probable interpretation can be chosen.

3.1 Trees

Although the ideas are general, Bod uses tree structures in the implementation of DOP. Before we describe the implementation, more information about tree structures and the operations on them is necessary.

Definition 3.1 (Tree structure and Root of a tree) *A* tree[2] *is a directed, acyclic graph which has a distinct node, the* root.
There is exactly one path from the root to every other node, so a root does not have any incoming edges.
The nodes are labelled with symbols, elements of $V_N \cup V_T$.

Definition 3.2 (Leaf) Leaves *are nodes in a tree that do not have outgoing edges.*

Definition 3.3 (Parent and Child) *If there is an edge from node* v_i *to* v_j*, then* v_i *is said to be the* parent *of* v_j *and* v_j *the* child *of* v_i.

Now we know what a tree is, we can define the notion of *elementary subtree* or *subtree*[3]. Figure 3.1 shows an example of elementary subtrees of a tree structure.

Definition 3.4 (Elementary subtree) *An* elementary subtree *of a tree* `T` *is a connected subgraph* `U` *of* `T` *such that*

[1]Most other natural language parsing systems do not use statistics. Most generative grammarians did not use statistics. See most of Chomsky's work.

[2]Actually, we want a tree to be an ordered tree, in which the nodes are ordered at each level.

[3]Note that this definition of subtree is not the usual definition of subtree.

1. *every node in* U *has either zero child nodes or all the child nodes of the corresponding node in* T, *and,*

2. U *consists of more than one node.*

Definition 3.5 (Composition and Rightmost substitution) *The* composition *or* rightmost substitution[4] *of subtrees* t *and* u, t∘u, *yields a copy of* t *in which its rightmost non-terminal leaf node that is labelled with the same label as the root node of tree* u, *has been replaced with tree* u.
We write (t ∘ u) ∘ v *as* t∘u∘v, *although the rightmost substitution operator is not associative.*

See figure 3.2 for an example of the composition operator.
DOP also uses the notion *corpus*.

Definition 3.6 (Corpus) *A* corpus *is a multi-set (or bag) of sentences together with their structures. This is usually a set of utterances together with their tree structures.*

We will also use the notion of *structured corpus*.

Definition 3.7 (Structured corpus) *A* structured corpus *is a quadruple*

$$C = (V_N, V_T, \mathsf{S}, P)$$

where

V_N *is a non-empty, finite set of non-terminal symbols,*

V_T *is a non-empty, finite set of terminal symbols,*

$\mathsf{S} \in V_T$ *is a special symbol, called the start symbol,*

P *is a non-empty, finite multi-set of tree structures.*

3.2 DOP framework

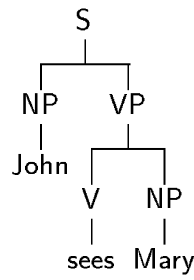
The DOP framework as described in [Bod95] is very general. We need to choose some parameters to make an actual implementation. The implementation described here is the DOP 1 system as described in [Bod95].

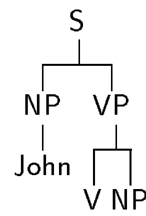
The different parameters are:

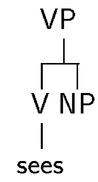
[4]Note that [Bod95] uses leftmost substitution. We will use rightmost substitution, because the Earley parser generates a list of productions numbers that need to be combined by rightmost substitution. The Earley parser generates a rightmost derivation. [Bod95] uses a chart parser that generates a leftmost derivation.

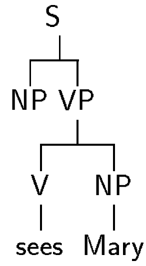
Figure 3.1 Elementary subtrees

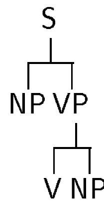
"main tree"

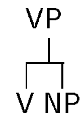


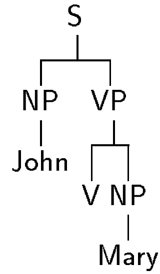


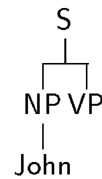


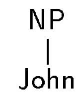


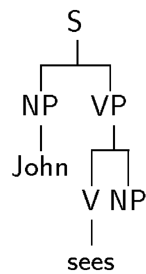


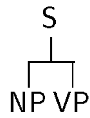


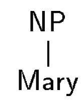


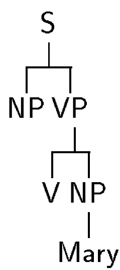


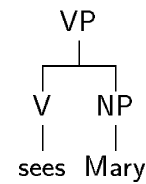


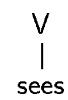


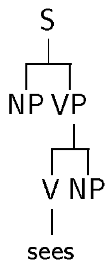


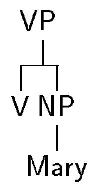












Sentence analyses The sentence analyses parameter describes the notation used for describing a sentence and its (most probable) interpretation. It is used as notation of the elements of the corpus.

Sub-analyses The sub-analyses are parts of total analyses. With these partial analyses a new analysis can be obtained. Sub-analyses describe the units in the analyses.

Combination operations Combination operations allow us to create a complete analysis from sub-analyses. It is the "glue" that fits sub-analyses together to create an interpretation.

Combination probability The combination probability defines the probability of a combination of sub-analyses. When the probabilities of the sub-analyses are known, the probability of the combination of the sub-analyses can be calculated using the combination probability.

We will now describe the parameters for the DOP 1 system.

3.2.1 Sentence analyses

In DOP 1 the sentence analyses are syntactically labeled phrase structure trees. These trees are in the form as if they were generated by a "standard" parser, based on a context-free grammar. Because it is based on a context-free grammar, DOP 1 can be used in all fields, not only in computational linguistics.

3.2.2 Sub-analyses

The sub-analyses used in DOP 1 are elementary trees. Elementary trees are parts that can be used to build complete sentence analyses.

3.2.3 Combination operations

In DOP 1 the only combination operation is rightmost substitution. The rightmost substitution operator "glues" the sub-analyses, the elementary trees, together to form a sentence analysis. See figure 3.2 for an example of the rightmost substitution operator.

3.2.4 Combination probabilities

In DOP 1 combination probabilities are defined as "the probability of selecting a subtree among all corpus-subtrees that could be substituted on a certain non-terminal leaf node". The chance of selecting subtree `t` from the structured corpus is

$$p(\mathtt{t}) = \frac{\#\{\mathtt{t} \in \mathrm{Corpus}\}}{\#\{\mathtt{v} \in \mathrm{Corpus} | \mathrm{root}(\mathtt{t}) = \mathrm{root}(\mathtt{v})\}}$$

This is the ratio between the number of occurrences of a subtree and the total number of occurrences of subtrees with the same root label. (This implies that the probabilities of the subtrees with the same root label sum up to one). To calculate the probability of two combined subtrees, we need the probabilities of both subtrees (we get these from the structured corpus). The probability of the combination of the subtrees is then the product of the probabilities of both the subtrees.

In order to calculate the probability of a subtree, we need to consider the number of times the subtree occurs in the structured corpus and all subtrees with the same root label. Because of this, calculating the probability of a subtree on the fly would generate considerable overhead. Fortunately, a corpus usually does not change, so the probabilities of the subtrees can be calculated in advance.

3.3 Parsing using DOP

Now that we have filled in the DOP framework, we can give the complete algorithm. The main algorithm is given in algorithm 3.1.

The Earley parser needs some modifications to allow it to operate with tree structures from the structured corpus instead of a context-free grammar. To reuse the "standard" Earley parser, we map the tree structures to their underlying context-free grammar rules. This is done by viewing the head of the tree structure as the left-hand side of the context-free grammar rule and the leaves of the tree structure as the right-hand side of the context-free grammar rule.

Algorithm 3.1 Parsing using DOP

Input

A sentence $\mathsf{w} = \mathsf{a}_1\mathsf{a}_2 \ldots \mathsf{a}_\mathsf{n}$ in $V_T{}^*$, and a structured corpus C.

Output

The most probable interpretation of sentence w, based on corpus C.

Method

1. A list of statesets for sentence w is computed using an Earley parser as described in algorithm 2.1. The only difference is that the parser does not work on context-free grammar rules, but on tree structures.

2. Now we have a list of statesets from which we can get the possible interpretations. The simplest way is to generate all possible interpretations (in the form of trees) out of the list of statesets by using algorithms 2.2 and 2.3, calculate their probabilities and select the most probable parse, the derivation with the highest probability.

The result is then the most probable parse, derived from the list of statesets.

If we use the approach described in algorithm 3.1, we will encounter two problems.

1. When parsing with context-free grammars, each tree structure can only be constructed in one way (if we assume for example rightmost derivations). This immediately becomes clear when we consider trees consisting of levels. With context-free grammar rules, we have exactly one choice on each level. The context-free grammar rules can be seen as trees with only two levels, the level of the root node and the level of the child nodes. Each non-terminal in a tree structure generated by context-free grammar rules is a "glue"-point. On each non-terminal another tree structure (in the form of a context-free grammar rule) is added. Because DOP uses larger structures as grammar rules, not every non-terminal in the tree structure has to be a "glue"-point. When other "glue"-points are chosen, it is possible to build the same tree structure in several ways. An example of this is given in figure 3.2. The "glue"-points are the non-terminals on which the composition takes place. This property marks the difference between a derivation and a parse; figure 3.2 contains two derivations for one parse.

2. The number of derivations can be exponential with regard to the length of the input sentence. So generating *all* derivations may take too long. A description of a method to calculate (with a certain error) the most probable parse follows in section 9.3.

Definition 3.8 (Parse) *A* parse *is a tree structure, with the start symbol as its root node and terminals as its leaves. (It is the result of a run of the parser.) The probability of a parse is the sum of the probabilities of its derivations.*
Note that the combination probability is defined in a way such that the probability of an elementary subtree `t`

$$0 < p(\texttt{t}) \leq 1$$

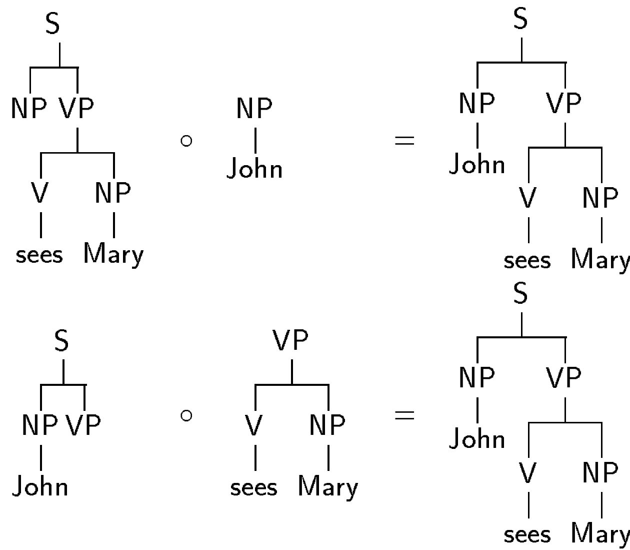
and

$$\sum_{\texttt{t}|root(\texttt{t})=\mathsf{A}} p(\texttt{t}) = 1$$

In order to compute the most probable parse, we need to select the parse with the highest probability.

Note that the DOP system uses an Earley parser, but Earley items are normally defined as context-free grammar rules with a • describing how far parsing has progressed. With DOP we use tree structures as if they were context-free grammar rules. Actually these tree structures can be seen as context-free grammar rules with some additional hierarchical information.

Figure 3.2 Multiple ways to build the same tree structure



Each tree structure can be transformed into a context-free grammar rule, losing only hierarchical information. This is done by taking the root of the tree as the left hand side of the context-free grammar rule and the leaves of the root as the right hand side of the grammar rule. The languages produced by the tree structures and the transformed grammar rules are the same.

3.4 Predecessors of DOP

There are pre-DOP statistical parsing methods which have been used to disambiguate the possible interpretations of sentences. All of these methods use context-free grammar rules, so there is no difference between a derivation and a parse, as is the case in DOP. While DOP can describe a statistical dependency between two parts of the input sentence that are not part of the same grammar rule, these pre-DOP parsing methods cannot do this, because the only statistical dependencies they can express are dependencies between parts of a context-free grammar rule.

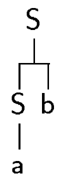
Each tree structure has one or more underlying grammar rules, but when we use these underlying grammar rules, we lose hierarchical information. A corpus and a grammar with the underlying grammar rules of the corpus will generate (and parse) the same language, but the structure of the parsed sentences might differ.

If we try to capture the hierarchical information as well, the underlying grammar rules (the root of the tree is the left hand side and the leaves of the tree are the left hand side) are not sufficient. Another approach needs to be taken. In order to capture the hierarchical information, it is necessary to

create context free grammar rules in which the left hand side of a grammar rule is the root of a (sub)tree structure and the right hand side is made up of the direct leaves of the root. This approach is different from the one which uses underlying grammar rules, because this time grammar rules are generated for each level in the tree structure. When this approach is taken, some corpora generate a different language than their corresponding grammar rules. See figure 3.3 for an example of a corpus that generates another language than its grammar rules that hold the hierarchical information.

Figure 3.3 Tree structure with no equivalent context-free grammar

Corpus:



Corresponding grammar rules capturing the hierarchical information:

S→Sb

S→a

The corpus only generates the sentence “a b”, but the corresponding grammar rules generate all sentences starting with an “a” and followed by any number of “b”s.

Although these pre-DOP statistical parsing methods are considered to be very good in disambiguating possible interpretations, the DOP system has a much higher accuracy; see [Bod95].

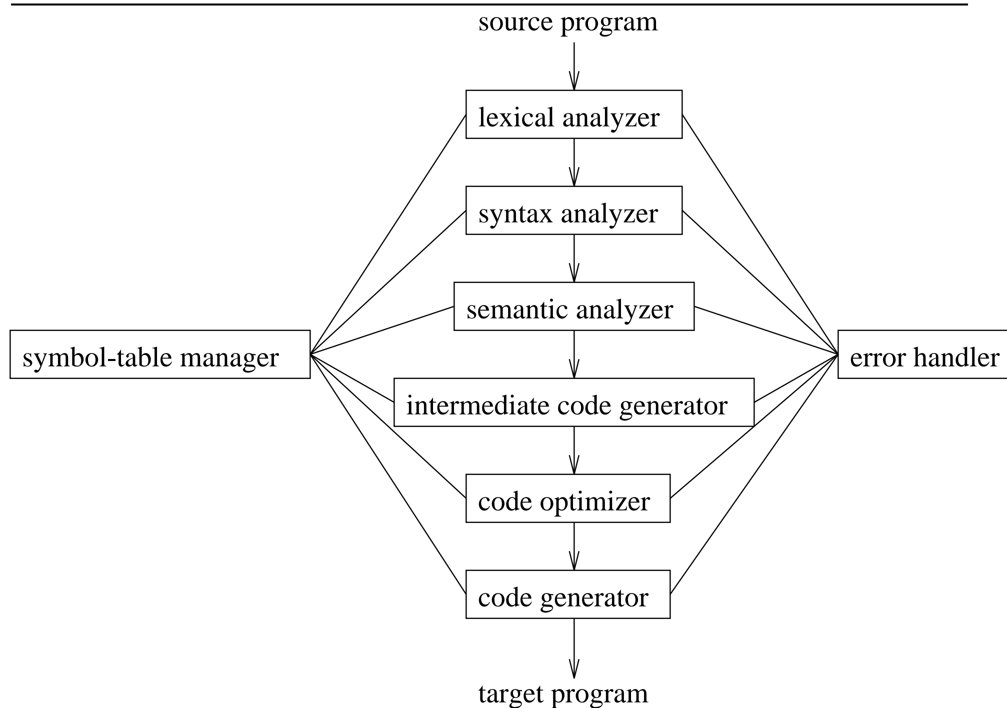
Chapter 4

Compilers

Compilers are programs that translate program code into machine executable code. A parser is one part of a compiler. The parser determines the structure of the program to be compiled, so the program can be translated.

A compiler usually contains several parts, each having its own function. See figure 4.1 for an overview of the parts.

Figure 4.1 The various parts of a compiler



Lexical analyzer This part of the compiler reads the input and chops it

into parts, called tokens. These tokens can be for example integers, variable names, etc.

Syntax analyzer The syntax analyzer reads the tokens generated by the lexical analyzer and parses these tokens into a structure of the compiled program.

Semantic analyzer This part of the compiler checks if the program is a semantically correct one. This means that for example it checks if all variables have been declared or if all called procedures are known.

Intermediate code generator The structure generated by the syntax analyzer and the semantic analyzer is then translated into a special language, the intermediate code. This is a language that is not yet machine executable, but it is a lot easier to translate into the machine executable code than the source language of the compiled program.

Code optimizer The code optimizer tries to optimize the intermediate code generated by the intermediate code generator.

Code generator This part of the compiler actually translates the intermediate code into machine executable language.

Symbol-table manager The symbol-table manager is a part of the compiler that is used throughout all compiling phases. The symbol-table contains information about parts of the compiled program. If any part of the compiler wants information about for example a certain variable, it asks the symbol-table manager.

Error handler The error handler takes care of all the errors that can occur in each phase of the compiler. Depending on the kind of error handling (see section 5), this part of the compiler can be complex or very simple.

Generally compilers are written partly by hand, but large parts of the compiler are generated automatically. Especially the parsers (part of the lexical analyzer and the syntax analyzer) may be generated by a parser generator. This is a program, that when given a grammar (in a certain form), generates a parser for it. The compiler can then use the automaticly generated parser.

4.1 LLgen

According to Jacobs, see [Jac94]:

> *LLgen* provides a tool for generating an efficient recursive descent parser with no backtrack from an Extended Context Free

> syntax. The *LLgen* user specifies the syntax, together with code describing actions associated with the parsing process. *LLgen* turns this specification into a number of subroutines that handle the parsing process.

LLgen is a parser generator, which means that LLgen is a program that, given a grammar and some extra information, can build a parser. Extra code can be added to the grammar rules to allow LLgen to build a complete compiler from the grammar.

LLgen needs the following parts to generate a parser:

Grammar The grammar describes the language from which sentences can be parsed.

Lexical analyzer The lexical analyzer tokenizes the input and hands the tokens to the main parse routine generated by LLgen.

Error message handler This routine generates error messages for the user.

Error handler This is an optional routine, which allows the way LLgen normally copes with errors in the input to be changed.

Chapter 5

Error handling

5.1 Introduction to syntax error handling

So far, we have only discussed the workings of a parser if all input is correct. In reality, it is not uncommon for the input to contain errors. In our case, if a program contains an error, the input is not a sentence described by the grammar.

Error handling can be divided into error detection, error recovery, and error correction. Error detection only tries to find an error in the input. When it has found one, it cannot continue, so it stops and in the best case gives us information about the error. Then there is error recovery: this technique not only tries to find an error in the input, but when it has found one, it also gives information about it, skips some input and tries to go on parsing (looking for more errors in the input). Finally, the method of error correction tries to find an error in the input and attempts to correct it, so that further errors in the rest of the input may be detected more effectively and no correct input is treated as an error.

Most parsers can be (slightly) modified to at least detect errors, but we would also like the parser to give us an indication of the exact place the error occurred. This is almost impossible. Some parsers have the *correct-prefix property*, but the place that these parsers indicate as the point of the error, does not have to be the "real" error. The "real" error could have been before that point, but it may not have been a syntax error at the time it was analyzed.

Definition 5.1 (Prefix and Correct-prefix property) *If* $\mathsf{w} = \mathsf{vu}$ *is a sentence, then* v *is a* prefix *of* w.
A parser has the correct-prefix property *when it can detect an error at the first symbol in the input that results in a string that is not a prefix of a sentence of the language.*

When a parser with correct-prefix property is going to try to parse an

erroneous symbol, it reads the symbol, examines it, and tries to process it, but fails. It would be better if the parser, while examining the symbol, immediately recognizes it as erroneous. This is called the *immediate error detection property.*

Definition 5.2 (Immediate error detection) *A parser has the* immediate error detection property *when it can detect an error when the erroneous symbol is first examined.*

This is a somewhat stronger property, because the parser will not even try processing the erroneous symbol; so a parser with immediate error detection property provides more context of the parse up to that symbol and gives more extensive information about the error.

When an error is detected and we want to continue parsing, the easiest thing to do is just skip symbols until we see a symbol on which we can continue parsing. This method has advantages and disadvantages; the advantage is that the parser can continue, as opposed to the error detection methods. The disadvantages are that a parser might skip too many tokens and thus skip possible errors and generate too few error messages. On the other hand, when some symbols are skipped, the parser might be in a wrong state to continue parsing from that position, so the parser might generate spurious error messages.

Even though the research in error recovery and correction in compilers started in the early sixties, for example with [Iro63], and continued in the seventies and eighties, for example [AP72], [Lév74], [Lyo74], [Tho76], [LF77] and [Pai80], the "perfect" error recovery system still does not exist.

5.2 Different types of error handling methods

Error handling methods can be divided into several different types. Some methods are of a hybrid kind; some even switch from one kind to another when appropriate.

5.2.1 Error detection

This is the simplest kind of error handling. It does not do any error recovery or correction at all. When an error is encountered it just gives up. This is the only error *detection* method. Although some methods generate more information than others, this is how all of them work.

This method has its advantages as well as its disadvantages. The greatest advantage is that it is easy to implement and it is very quick. There is no need to try and find a way to correct the error. The major disadvantage is that it does not do any error recovery or correction, (hopefully) some implementations says where the first error is and what kind it is. Because

it skips the rest of the program, it does not say anything about the part of the program after the error.

5.2.2 Ad hoc error handling

Ad hoc error handling methods are called *ad hoc*, because they cannot be generated from the grammar. It is the parser writer which takes care of the error recovery or correction part of the parser.

An example of an ad hoc error handling method is the method using error productions. Error productions are special grammar rules, added by the parser writer, so certain errors are no longer syntax errors. Attached to these error productions are error routines, which can give very specific information about the error. The main advantage is that it is (often) easy to implement, because the error productions are just grammar rules with some special semantics attached to them, so very specific error messages can be given. The greatest disadvantage is that only anticipated errors can be handled this way.

5.2.3 Local error handling

Local error handling methods are methods that only use information available at the moment an error occurs, so no context of where the error occurred is used. This makes these kinds of method often easy to implement and quick, but the parser cannot make the best decision about how to recover from the error.

All these methods start skipping tokens when an error is encountered, until the input contains a token that is in the so called *acceptable-set*. When such a token is found, the parser continues.

Local error handling methods differ in the construction of their acceptable-set. The easiest method is called panic mode. For example when an error is encountered in natural language texts, the parser skips tokens until the '.' token is read. The parser then knows a sentence has been completed, so it can start parsing another sentence. In this case the acceptable-set is {'.'}.

These methods are often quite easy to implement, but when an error is found, sometimes large parts of the input are skipped, so no error can be found in that part of the input.

5.2.4 Regional error handling

Regional error handling methods use some context around the error. This way they can make a better choice of how to continue parsing. This way of error handling is most often used in bottom-up parsers, where the error and some context is reduced to a left-hand side of a grammar rule.

The advantage is that a better choice can be made how to recover from the error and where to continue parsing, and little input is skipped. A

disadvantage is that the methods are often quite complex and difficult to implement.

5.2.5 Global error handling

Global error handling techniques consider all of the input when they encounter an error. These techniques are most often used in general parsers, like the Unger and CYK parsers.

The best known global error handling method is called the least-error correction method. This method (described in [AP72]) computes the least number of corrections needed to change the (erroneous) input to a correct one.

Definition 5.3 (Correction) *A* correction *is an operation which transforms an input sentence into another input sentence. The most common corrections are:*

- *the insertion of a symbol,*
- *the deletion of a symbol.*

These are the two basic corrections; all other corrections can be produced from these two corrections. Some other common corrections are:

- *the replacement of one symbol by another,*
- *the transposition (the changing of places) of two adjacent symbols.*

The advantage of these methods is that since the method uses all the input, a large amount of information is available, and a good error correction choice can be made. The disadvantage is that because these methods use all of the input. They are (often) quite complex; they are also not very efficient in time and space.

Chapter 6

Conclusion

6.1 Parser

Parsing is actually the structuring of information. It can be used in many different fields of science. Most important is the grammar and the parsing method. The grammar tells us in what structure the information should be, while the parsing method tells us how to get from the information to the structure as described by the grammar. The grammar depends on the specific field, while parsing methods are very general and can be used in various subject fields.

6.2 DOP

There are a lot of different parsing methods. Our primary interest went to the DOP system. This is one of the several parsers which incorporate the ambiguity of the grammar directly into the parser. Internally it uses an Earley parser, and after parsing (and generating a representation of all possible interpretations) it chooses the most probable parse based on the corpus of tree structures.

6.3 Compilers

Compilers are programs that translate program code into a machine executable language. They consist of several parts, including (at least one) parser.

The parsers are often generated by a parser generator. The parser generator is given a grammar and it generates a parser for that grammar. LLgen is such a parser generator.

6.4 Error handling

Although lots of work has gone into the research of error handling, no "perfect" error handling routine exists. Error handling can be divided into:

- Error detection
- Error recovery
- Error correction

Error detection is the most easy, while error correction is the most difficult. Error detection and error recovery methods are often quite efficient but very simple, while error correction methods are not efficient in time and space and often very complex.

In the next part, an error correcting method will be described which (in its present form) is not very efficient, but makes an attempt to calculate the most probable parse and derives from it the most probable error in the input. Most likely, this method corrects the error in the best way possible.

Part II

Implementation

Chapter 7

Goal

Part I discussed parsing techniques that can be used in various subject fields. In this part, the focus will be primarily on parsers in compilers. Although some of the ideas presented here can also be used in general parsing techniques, they were designed to work in compilers.

The goal of this project was to devise and implement a system that would do "perfect" error-handling in a compiler. According to Pai and Kieburtz in [Pai80] a "perfect" error handling scheme should meet the following goals:

1. It should never fail to return control to the parser, and when it does, parsing should be able to continue. The interval of text that escapes analysis in the course of attempting recovery from an error should be minimal.

2. It should not introduce spurious errors into an otherwise syntactically correct text fragment as a consequence of its actions in recovering from a single error.

3. It should not have higher space or time complexity than the parser itself.

4. It should not degrade the performance of the parser on error-free text segments.

ad (1) It is obvious that the error-handling routine should return control to the parser, otherwise the program would never get compiled. It is obvious, too, that when the error-handling routine returns, the parser should be able to continue parsing, since that is exactly what the error-handling routine is supposed to do. More interesting is the size of interval of text that escapes analysis in the course of attempting recovery from an error. Of course it should be minimal, because in that interval no errors can be detected and we want to find all of the errors in the input. Therefor our system does not skip input at all.

We try to continue parsing, but because the input contains an error, we have to make corrections on it.

Note that even when we delete a token in the input it is not skipped; every token is considered when parsing or correcting. An token that erroneously got inserted in the input, will be deleted, but not skipped (the token is still analyzed). The "correct" input is the input without that token.

ad (2) Several systems may be devised to make sure there will be no spurious errors into an otherwise syntactically correct text fragment as a consequence of recovering from a single error. For example, the syntax-directed least-error analysis [Lyo74] is a system that computes the minimal number of corrections needed to make incorrect input correct. To achieve this goal, global or at least regional error handling is necessary. Trying to achieve this with local error-handling techniques requires some choice to be made, without knowing the rest of the input. Insufficient information about the rest of the input may result in a wrong choice. Even regional error-handling techniques have this problem, namely if a cluster of errors occur. Assume we have a regional error-handling method that uses a context of n symbols and we have a cluster of $n + 1$ errors. Then the decision of correcting the error is based on incorrect information. Regional error-handling techniques can be good when the context is large enough. The system described here uses global error handling, although the different implementations use different levels of global error handling.

ad (3) Although we want the time and space complexity of the error handling routine the same as the "normal" parsing method, this property is very hard to achieve. Of course there are lots of error-handling methods that have this property; for example panic mode error handling and some ad hoc error-handling methods have this property, but all of them have bad (if any) error correcting properties. The system described here has bad space and (especially) time complexity, but variants of our system exist that use the same ideas as the system described here but replace the inefficient parts. The disadvantage of these systems is that they are more complex and difficult to implement, while the system described here is far easier to understand.

ad (4) When parsing a correct program, the compiler should be able to compile as fast with one error-handling routine as with another one. This means the error-handling routine should be called only when an error is encountered. Although this is not completely true for our system because information about the correct part of the input is stored even with correct programs, the overhead is very small and almost neglectable. The system described in section 12.4 does not

have this overhead, so the time and space complexity of parsing a correct program is not affected at all.

Some of these properties are contradictory. For example, item (2) and item (3) are contradictory, because to computing a correction that does not introduce spurious errors, the error-handling routine should know (at least a bit) more of input to come. If we assume that the parser of the compiler itself is (almost) linear, the error-handling routine is more than linear. If we only drop item (3) for now, we can create a good error-handling routine.

Chapter 8

Programming languages

8.1 Compiler

The compiler used in this project is the ANSI-C compiler, developed at the Vrije Universiteit in Amsterdam.

We have chosen for a C compiler because:

- A C compiler was available including its source files.
- The C compiler is generated using LLgen, a compiler generator which allows us to attach our own error-handling routines without having to change the rest of the compiler.
- The C programming language is very common, so it should not be too difficult to find test programs and to fill the corpus. A corpus in this context is a multi-set of (parts of) C programs with their structure. This corpus is used in the DOP part of the error handling.

8.2 Implementation

We started by writing the DOP part of the error handling of the C compiler. This part is written in C++. There are several reasons why we chose the C++ programming language for this part.

Modules The DOP part of the error-handling module can be split into small parts, each having its own data structure, semantics and functionality.

Re-implementation One of the greatest advantages of the C++ programming language is that parts can be easily replaced by other parts, if the new parts provide the same functionality. For example, our current implementation is very inefficient regarding disambiguation after parsing. More efficient implementations exist, so they can be

easily incorporated into the existing error-handling module. The only thing that needs to be done is to create a new class with the same interface and the same functionality, but with a higher efficiency.

Extendibility Because the DOP error-handling module started out as a "normal" DOP parser, it needed to be extended so input containing errors could be handled as well. This extension was easy to incorporate into the existing system because C++ had been used.

Interface Because the existing C compiler (which we wanted to give a new error-handling method) was written in C, we needed to use either C itself or a programming language that could be linked easily with C code. With C++, linking is easy indeed.

Chapter 9

Data Oriented Parsing

The Data Oriented Parsing part of the error-handling module started out as a DOP parser. This parser was later changed to parse incorrect input and to calculate the most probable parse including the corrections made to the input.

In this chapter we will discuss the DOP parser and in section 11.1 we will describe how we handle the errors in the DOP parser.

The class structure of the program (because the DOP part of the error-handling module is written in C++, the class structure is mostly about the DOP part) can be found in appendix A.5.

9.1 Parsing

The first thing we need in order to implement a DOP system, is a parser. As explained before, we chose to use an Earley parser.

9.1.1 `subtree` class

In order to implement an Earley parser that uses tree structures rather than linear items, we started out to implement a tree structure. The code for the tree structure is found in the `subtree` class. This is a very simple way of implementing a tree structure; each node in the tree is an instantiation of the class `subtree`. The class `subtree` consists of an `elem`, which is the data type of the contents of the tree (in our case `ints`) and a (possibly empty) list of pointers to `subtrees`. We chose to use pointers because this avoids copying `subtree` instantiations.

For the list we used a datatype from *STL*. STL, Standard Template Library, described [SL95] by Stepanov and Lee, provides a set of well structured generic C++ components that work together seamlessly.

9.1.2 `itemtree` class and `st_iterator` class

Now that we can store a tree structure, we need to add the • to the tree. This is done by defining a new class structure, `itemtree`. `itemtree` is based on `subtree`. Everything that can be done with a `subtree` can also be done with an `itemtree`.

We had to add something like a pointer to the class `itemtree` that indicates where the • is in the `itemtree`. This was done by defining a new class structure `st_iterator`, which is an `iterator` as defined in STL. According to Stepanov and Lee:

> Iterators are a generalization of pointers that allow a programmer to work with different data structures (containers) in a uniform manner.

So an `iterator` is a special kind of pointer. We defined one on the class `subtree`, so the pointer always points to a leaf in the tree or to the non-existing *past-the-end* leaf, which is the imaginary leaf past the last leaf in the tree. Note that the `st_iterator` is not just a pointer. We *made* the `iterator` follow the leaves if the `iterator` is increased.

So the `itemtree` is just a `subtree`, with a `st_iterator` pointing to where the • should be.

9.1.3 `edge` class

Now that we have the `itemtree` class, we can define an `edge`. The `edge` class is what we called a *state* in section 2.2.2. This is an item together with a pointer pointing to the stateset where the item started.

The `edge` is a class with an `itemtree` together with an `int` pointing to the stateset (declared as `chartstate`, see section 9.1.4) in which the item was introduced.

9.1.4 `chartstate` class

When parsing, each terminal introduces a new stateset (see section 2.2.2). This stateset is called `chartstate` in our implementation. It is a container of states, or `edges`.

9.1.5 `chart` class

A `chart` is a list of `chartstates`. The `chart` is the list of statesets in the original algorithm.

We have implemented the `chart` as having all of the functionality. It is possible to send a message to (call a member function of) the `chart` class to make it parse a sentence, for example.

We have used a different naming scheme for two reasons:

1. We use a slightly different approach, since we use trees instead of context-free grammar rules.

2. We implemented an Earley parser, but we also had a *chart* parser in mind. It must be clear that we have implemented an Earley parser, not a chart parser. The Earley parser and chart parser are very alike, but we feel that the chart parser is a bit easier to run manually. Since the first parses were done on paper, we used the more human-oriented output of the chart parser. The statesets of the Earley parser are more computer oriented and are thus less easy to read.

9.2 Corpus

Although we have now described the data structures needed for an Earley (or chart) parser, we still need a data structure to store the grammar. In the DOP case, we do not have a grammar in the usual sense, but we have a multi-set of (partial) tree structures. The multi-set of (partial) tree structures is called a structured corpus, see definition 3.7.

In our case the `corpus` class is a list of `subtree`s.

9.2.1 Format of the corpus

Corpora are often non-changing multi-sets of sentences or in our case (partial) tree structures. They are often quite big (thousands of entries). Because of these properties, we want to have them stored in files, so we can let a program read them when we need them.

When we want to store corpora, we need to agree on a format. It does not really matter what the format is, just as long as we know how to read and write it.

We have chosen a very simple format. It was based on several other formats of corpora.

- Each tree structure starts on a new line.

- Each (sub)tree starts with the keyword `tree`.

- The data in the root of the (sub)tree follows the brace ('('), which directly follows the keyword `tree`.

- The data in the root of the (sub)tree is followed by a square bracket ('[') after which the direct children of the root are stored.

- After the last child of the root, the closing square bracket (']') is entered.

- At the end the closing brace (')') is entered.

- There is no white space at all (except for the newlines which separate the complete tree structures).
- The tree structure is followed by a period ('.').

See figure 9.1 for a sample from the corpus.

Figure 9.1 A small sample from the corpus of the C programming language

```
tree(program[tree(external_definition[])]).
tree(external_definition[tree(decl_specifiers[])tree(declarator[])tree(function[])]).
tree(decl_specifiers[tree(single_decl_specifier[])]).
tree(decl_specifiers[tree(single_decl_specifier[])]).
tree(single_decl_specifier[tree(int[])]).
tree(single_decl_specifier[tree(void[])]).
tree(primary_declarator[tree(identifier[])]).
tree(identifier[tree(identifier[])]).
tree(parameter_type_list[tree(parameter_decl_list[])]).
tree(parameter_decl_list[tree(parameter_decl[])]).
tree(parameter_decl[tree(decl_specifiers[])tree(parameter_declarator[])]).
tree(parameter_declarator[tree(primary_parameter_declarator[])]).
tree(function[tree(compound_statement[])]).
tree(compound_statement[tree('{'[])tree(return_statement[])tree('}'[])]).
tree(return_statement[tree(return[])tree(expression[])tree(';'[])]).
tree(expression[tree(assignment_expression[])]).
tree(assignment_expression[tree(conditional_expression[])]).
tree(conditional_expression[tree(binary_expression[])]).
tree(binary_expression[tree(postfix_expression[])]).
tree(postfix_expression[tree(constant[])]).
tree(constant[tree(integer[])]).
```

9.2.2 Filling the corpus

Now that we have the DOP parser, we need a corpus of tree structures. The DOP system uses elementary trees as its "grammar rules". These elementary trees are tree structures describing complete and partial tree structures, as described in section 3.1.

The corpus will reflect the C grammar. This means that all constructs that are part of the C programming language should be present in the corpus. When the context-free grammar of C is part of the corpus, we know the corpus is "complete". All C programs can then be parsed. This does not mean that the most probable parse will be calculated. The latter depends on the number of times certain tree structures are present in the corpus. We assume the corpora used are complete.

The easiest way (in our view) was first creating a corpus of tree structures of complete sentences (C programs in this project) and then transforming the corpus of tree structures of complete sentences into a corpus filled with elementary trees of these complete tree structures.

Compiler

In order to get the compiler to generate tree structures, we added procedure calls to each grammar rule which generates a tree structure. This behavior can be switched on at runtime, by a parameter of the C compiler; see section A.4.

When parsing, a tree structure is built dynamically. In the beginning of each grammar rule (in LLgen), the beginning of a new level in the tree is introduced and the value stored in the root node of the subtree is set. The root of the subtree is the non-terminal of the left hand side of the grammar rule and the leaves are made up of the symbols of the right hand side.

Normally we want to parse a complete program to correct the errors in it. A problem arises when large programs need syntax correction, because the DOP system is not very efficient in time and space. To circumvent these difficulties we tried to reduce the tree structure on which the error correction takes place. This was done by selecting a root node different than `program`. The possible root nodes are:

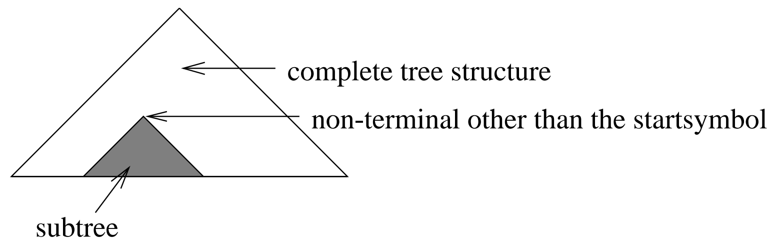
- `program`, the default value, used to parse and correct errors with as context the complete program.
- `block`, used to parse and correct errors with as context a block structure.
- `statement`, used to parse and correct errors with as context a statement.
- `expression`, used to parse and correct errors with as context an expression.

When an error is encountered a tree structure with the selected root node is built. This tree structure is parsed and disambiguated and the most probable parse can and will be generated using the DOP parser. The only difference is that a smaller context is used. Section A.3 describes how the necessary adjustments may be made.

When `program` is chosen to be the root of the tree structure to be built, a tree structure of the entire program will be built when an error is encountered. When another non-terminal (`block`, `statement`, `expression`) is chosen to be the root of the tree structure, a smaller tree structure will be built, because the non-terminal can only be part of a `program`. See figure 9.2 for a graphical representation. The subtree in the complete tree structure

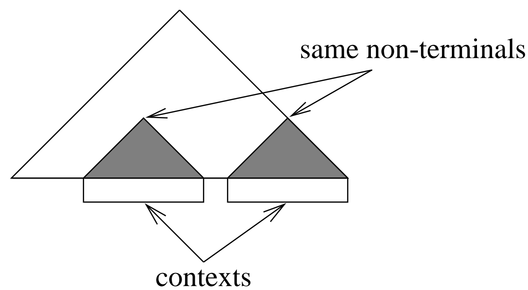
has as its root a non-terminal other than `program`. The calculation of the most probable parse will take less time, because smaller tree structures need to be considered.

Figure 9.2 A subtree in a complete tree structure



When a non-terminal other than `program` is chosen to be the root of the tree structure that is used to compute corrections in the input, a problem arises. As can be seen in figure 9.3, the contexts of the smaller tree structures do not necessarily cover the complete program. When an error occurs between the contexts, the error handler does not know how to correct it. It can only correct errors within the contexts. For example, `expression` is chosen to be the root of the subtrees describing the contexts. This means that the contexts are expressions. If an error occurs outside an expression, the error handler does not know how to correct it, because the error handler only knows how to correct expressions. In figure 9.3 this would be an error outside the contexts.

Figure 9.3 Contexts in a complete tree structure



The tree structures are appended to the file `corpus`, when compilation has finished. The compiler creates this file when it does not exist.

Elementary Trees

First we compile C programs and generate tree structures of them. After that we generate the elementary trees of these tree structures. If we do not

do this and the corpus exists only of tree structures of complete programs, the C compiler will always generate one of these complete programs upon error[1].

To compute the elementary subtrees from a corpus of complete tree structures, we have written a program `c2d`, which stands for *corpus to dop*. This program will generate a new file, consisting of all elementary subtrees of a certain maximum height. This maximum height can be chosen at runtime.

Our first implementation generates all possible elementary trees according to algorithm 9.1. The algorithm was used on all complete tree structures of the corpus.

Algorithm 9.1 Compute all possible elementary subtrees

Input
A tree structure `t`.
Output
A list of tree structures, *elem_trees*, which contains all elementary trees present in tree `t`.
Method
We assume that *elem_trees* is a global variable and is initially empty.
Execute routine **R**(`t`, *root_trees*).
The routine **R** is defined as follows:
R(`u`, *root_at_u*):

1. If `u` is a leaf then add `u` to *root_at_u* and return.
2. Let $\mathtt{v_1} \ldots \mathtt{v_n}$ be the tree structures that are direct children of the root of tree `u`.
3. For $i = 1$ to n, execute **R**($\mathtt{v_i}$, *root_at_*v_i). *root_at_*$v_1 \ldots$ *root_at_*v_n are local variables, that are initially empty.
4. Add the empty tree structure to all variables *root_at_*$v_1 \ldots$ *root_at_*v_n.
5. Generate tree structures by taking the root of `u` and attaching children by selecting all possible combinations of subtrees from *root_at_*$v_1 \ldots$ *root_at_*v_n (in the right order). Add these tree structures to *elem_trees* and *root_at_u*.

The problem with this algorithm was that with larger tree structures, it used too much memory and the number of elementary subtrees became too

[1] Another problem occurs here, depending on the error-handling method which is actually used. When the input remaining after an error cannot be changed into a program in the corpus, the error handling routine does not know how to handle this. The DOP part of the error handling routine will not generate a parse, so the error-handling routine will give up (at least in the implemented method, see section 12.3). That is why we assume the corpus is complete.

large to handle ($\gg$ 10000 elementary subtrees). We adjusted the algorithm to let it generate elementary subtrees with a certain maximum depth. This way the maximum depth of the elementary subtrees can be adjusted, so the number of elementary subtrees can be adjusted. The algorithm is given in algorithm 9.2.

The algorithm generates all elementary trees with a certain maximum depth by computing the elementary trees per level. First, the elementary trees with the maximum depth are generated, after that the elementary trees with the maximum depth minus one are generated and so on.

Routine R computes the elementary subtrees with a certain depth. It generates the elementary trees of the children of the complete tree with the depth minus one; this is done in step 3. If none of the children contain elementary subtrees with that depth, all $ret_1 \ldots ret_n$ will be false. If this is the case, no elementary tree with the correct depth can be calculated. A tree with a certain depth needs at least one child to have depth minus one.

9.3 Disambiguation

When the Earley (or chart) parser has generated a compact representation (chart), we want to calculate the most probable parse. This is done by *disambiguating* the chart.

The idea of disambiguation is to find out which tree structure has the highest probability. First of all, the elementary trees (the grammar rules and the parts in the chart) have a probability depending on the frequency with which they occur in the corpus. This probability is calculated by keeping a score of the number of times a certain subtree occurs in the corpus (stored in the instantiation of the class `corpus`) and the number of times a non-terminal occurred in a subtree as root (stored in the class `trans`).

The class `trans` exists because of two reasons:

1. `trans` allows us to map the name of a terminal or non-terminal to an `int`. This way a more compact representation of tree structures can be achieved. The corpus in a file is represented as a tree in which terminals and non-terminals are represented by character string, while in the internal representation of a tree terminal and non-terminals are represented by `int`s, terminals and non-terminals.

2. An `int` is stored with each member of the mapping in `trans`. This `int` is the number of times that member occurred as a root of a tree structure. This `int` is always zero for a terminal.

The probability of a tree structure can now be calculated using the formula in section 3.2.4. We now calculate it by the following formula, using

Algorithm 9.2 Generate elementary subtrees with a certain maximum depth

Input
A tree structure `t` and a maximum depth m.
Output
A list of tree structures, *elem_trees*, elementary trees with maximum depth m based on tree `t`.
Method
We assume that *elem_trees* is a global variable and is initially empty.

1. For i=1 to m, execute routine **R**(`t`, true, i, $root_trees_1$, c).

Routine **R** is defined as follows:
R(`u`, *insert*, *max_level*, *root_at_u*, *complete*):
root_at_u is initially empty.

1. If *max_level*=0 then add `u` to *root_at_u* and set *complete*=true. Return.
2. If `u` is a leaf then add `u` to *root_at_u* and set *complete*=false. Return.
3. If *insert* is true then execute for each tree structure $\mathtt{v_i}$, that is a direct child of the root of tree `u`: **R**($\mathtt{v_i}$, true, *max_level*, $root_trees_2$, *ret*).
4. Execute for each tree structure $\mathtt{v_i}$, (from $\mathtt{v_1} \ldots \mathtt{v_n}$) that is a direct child of the root of tree `u`: **R**($\mathtt{v_i}$, false, *max_level*-1, $root_at_v_i$, ret_i). All $root_at_v_i$ are initially empty. If $\mathtt{v_i}$ is not a leaf then add the root of $\mathtt{v_i}$ to $root_at_v_i$. If ret_i is false for all of the calls to **R** then set *complete*=false.
5. Generate tree structures by taking the root of `u` and add children by selecting all possible combinations of subtrees from $root_at_v_1 \ldots root_at_v_n$ (in the right order). Add them to *elem_trees* and to *root_at_u*. Set *complete*=true.

the values stored in the `corpus` and the `trans`:

$$p(\mathtt{t}) = \frac{\mathtt{int}\text{ stored together with }\mathtt{t}\text{ in }\mathtt{corpus}}{\mathtt{int}\text{ stored together with the root of }\mathtt{t}\text{ in }\mathtt{trans}}$$

9.3.1 Monte Carlo disambiguation

The main problem with disambiguation is that the chart can be used to generate exponentially many tree structures. So considering the probability of all possible tree structures may take too long. (Note that the chart is a compact representation of the exponentially many tree structures. The chart contains subtrees, which can be combined into complete tree structures of which there may be exponentially many.)

The method used here is a polynomial time algorithm, called *Monte Carlo* disambiguation. See [Bod95] for a more extensive description of this method. The idea behind Monte Carlo disambiguation is that when selecting random subtrees from the chart based on the probability of the subtrees, the more probable subtrees will be chosen more often. At each combination of subtrees, subtrees with a high probability have a high chance to be chosen, so the most probable parse (within a certain error) comes "boiling" up after a lot of samples.

Remember that the chart consists of elementary subtrees. When we want to obtain a complete tree structure, we select (randomly, but based on the probability of the subtree) a subtree with the startsymbol as root from the chart. This subtree might need some other subtree at a leaf to make it (more) complete. This subtree is then selected from the chart, again based on the probability of that subtree. When a complete tree is sampled from the chart, its probability is calculated (we know the probability of the separate subtrees, so we can calculate the probability of the complete tree using the formula in section 9.3) and the tree together with the probability is stored.

The sampling of tree structures from the chart is done multiple times. When a tree structure has been sampled that had already been sampled before, the probabilities of the samples are totaled. This is exactly the difference between a derivation and a parse; a sampled tree structure is a derivation, whereas the final tree structure (the parse) can have several different derivations of the same tree structure.

Note that with Monte Carlo disambiguation, the "probability" of a parse can get higher than one, because some derivations may be sampled more than one time from the corpus. In the "perfect" DOP disambiguation, all derivations are counted exactly once. Therefore we will make a distinction between the probability of a parse as used in the "perfect" DOP disambiguation and the *prominence* of a parse to denote the "probability" used in Monte Carlo disambiguation.

It is not a problem that the prominence can grow higher than one, because we are only interested in the most probable parse. This means we are

not interested in the exact probability of a parse, we are only interested in the parse with the highest probability. When Monte Carlo disambiguation is used, we are only interested in the parse with the highest prominence. See [Bod95] for more information on Monte Carlo disambiguation.

Chapter 10

LLgen

The basics of LLgen have been described in section 4.1. LLgen is the parser generator used to build the parser of the C grammar used in this project.

The grammar of the C programming language was already part of the source code, just as the code that had been added to the grammar rules. In addition to the grammar and the extra code attached to the grammar rules, LLgen also needs a couple of routines, to be added by the user. These routines are:

Lexical analyzer The lexical analyzer is a routine that reads the input and returns *tokens*.

Definition 10.1 (Token) *A* token *is a data structure, containing information about a lexical token. More in particular, the token contains at least the class and the contents of the lexical unit.*

Tokens are used because the parser often does not need to know the real input "123" but it needs to know it is a *integer*, as in the grammar rules. For example the input "123" will be turned into a token *integer*.

The lexical analyzer must have the facility to push back one token. In LLgen notation this routine is defined by defining *%lexical*; we will call this routine the `lexical`. This routine is obligatory, but because it was already part of the C compiler, it will not be discussed here.

`LLmessage` This routine handles error messages. When an error has occurred and the error-handling routine has decided what should happen, `LLmessage` is called. This routine puts an error message on the output. `LLmessage` also provides information so the parser can be put in a known state, so parsing may continue. This routine was also already part of the C compiler.

Error handler This is an optional routine. Most compilers do not use this routine, because LLgen has a built-in error handler. Because

we want to add our own error-handling routines, we needed to add this routine. LLgen needs to know which routine serves as the new error handler. This is done by assigning a routine to *%onerror* in LLgen notation. Therefore we will name our error-handling routine the %`onerror` routine.

In normal parsing, LLgen keeps calling the lexical analyzer to get new tokens to parse. When an error occurs, LLgen calls %`onerror`. %`onerror` computes what should be done: whether to delete a token, insert a token or do something else. Then it calls `LLmessage`, to put an error on the output. After that parsing can continue, so LLgen starts calling the lexical analyzer again.

To add our own error-handling routines, we had to change some things in the compiler. We had to change the `lexical` routine, the `LLmessage` routine and we had to add the %`onerror` routine.

10.1 `lexical` routine

The lexical analyzer of the C compiler could be reused almost completely; there was just one problem.

Normally when an error occurs and we are using LLgen's own error-handling routine LLgen's error-handling routine is called, which determines the appropriate correction. LLgen then calls `LLmessage` to put an error message on the output and get the necessary information needed to correct the input, so LLgen can continue parsing from the error point. LLgen always corrects only one error at a time, and the correction is done by inserting or deleting one token. To do this, the lexical analyzer needs to be able to push back one token, i.e. when correcting the error by inserting a token. If a token needs to be inserted, `LLmessage` puts the token to be inserted in a variable called *aside*. When `lexical` is called, `lexical` first checks if there is something in the variable *aside*. If this is the case, `lexical` returns this token; if *aside* is empty, it proceeds as usual by reading the next token from the input.

The problem arises when we make LLgen call our own error-handling routine in case of an error. This routine reads all remaining input after the error point and, based on that information, calculates the most probable parse and corrects the input. But now all of the input, which was read because it was needed to calculate the most probable parse, needs to be pushed back. Pushing it back is necessary because LLgen calls `lexical` when it needs a token, so `lexical` needs to know all of the remaining input (including the corrections made).

We solved this problem by first disabling the "aside"-mechanism in `lexical`. Now `lexical` never looks at the *aside* variable but instead, when `lexical` is called, it just reads a new token from the input.

We then added a new lexical analyzer, new_lexical, which may be called in two ways.

1. When the %onerror calls new_lexical it calls the old lexical analyzer lexical, but it also records the token in a data structure called tokenqueue. So each time the %onerror calls new_lexical, a token is read from input and it is appended to the tokenqueue.

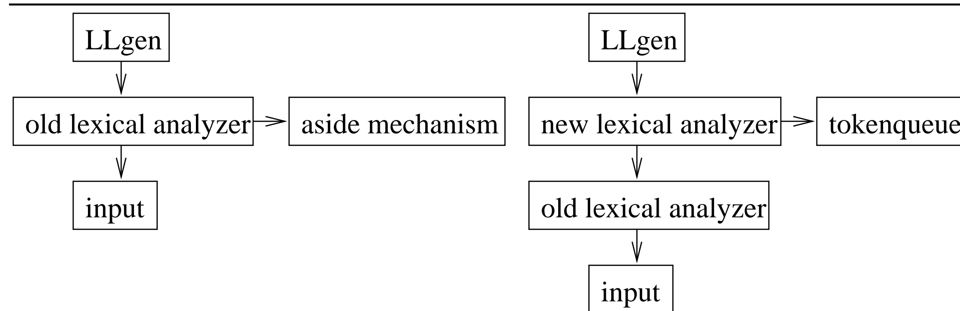
2. When LLgen calls new_lexical, it just calls the old lexical analyzer lexical when the tokenqueue is empty. This way parsing without errors works as usual. When the tokenqueue is not empty, it returns the first token from the tokenqueue. The tokenqueue is a FIFO, First In First Out, queue. Tokens can only get on the tokenqueue because %onerror has called new_lexical. So tokens on the tokenqueue are there because they have been read, but not yet given to LLgen to parse them.

The aside mechanism can be seen as a one-token tokenqueue. We have modified the existing aside mechanism to an aside mechanism that can remember as many tokens as the memory of the computer can handle.

The tokenqueue is used to store all tokens that have been read, but have not yet been processed by the parser. These tokens were used for error handling. Initially, the tokenqueue contains the exact input, so it still contains the errors that were present in the input. It should be evident that corrections of errors must be made on the tokenqueue. After the corrections, the tokenqueue consists of correct input, so LLgen can continue parsing.

Adding the new lexical analyzer can be seen as adding a new layer between LLgen and the old lexical analyzer, see figure 10.1. The arrows indicate calls, not data flow.

Figure 10.1 Old and new layers of the lexical analyzer



10.2 LLmessage routine

The LLmessage routine is called when LLgen finds an error in the input. This routine was already part of the C compiler, but it needed to be changed because of the "aside"-routine.

LLmessage has one parameter, an integer. This integer can be -1, 0 or a token number. When LLmessage is called with -1 as its parameter, LLgen expected EOF, End Of File, but it did not get it. When LLmessage is called with 0 as its parameter, the current token needs to be deleted and when LLmessage is called with a token number, that token needs to be inserted.

When LLmessage is called, it puts an error message on the output and it makes sure that next time the lexical analyzer is called, the correct token is returned. So next time LLgen calls the lexical analyzer, EOF is returned when LLmessage was called with a -1 as its parameter, the lexical analyzer returns the next token when LLmessage was called with a 0 and the lexical analyzer returns the inserted token when LLmessage was called with a token number.

With the C compiler, a call to LLmessage with a -1 will skip all tokens until the EOF token is found, a call to LLmessage with a 0 will delete the current token, so the next token is read and parsing may continue. But a call to LLmessage with a token number will have to create a new "dummy" token, together with some default properties. For example, if a variable needed to be inserted, a new token of kind variable is created with as its default properties that the variable is a ErrorType and its name is a. This token is then put in the *aside* variable, so the next time lexical is called, the token in the *aside* variable is returned.

Since we disabled the *aside*-routine in the lexical analyzer, we needed to change the part of the *aside*-routine in LLmessage, too. The only problem was when a token needed to be inserted. In this case we put the token aside in front of our new tokenqueue. When new_lexical is called, the newly created and inserted token is returned first, just as it would be in the C compiler with the *aside*-routine.

10.3 %onerror routine

With the %onerror routine it is possible to turn off the error-handling routine of LLgen and let LLgen call the procedure that replaces the error handling routine of LLgen. This is the heart of the error-handling part of the compiler. When LLgen notices an error, it calls this routine.

The %onerror routine is called with two parameters:

1. The first parameter is an integer. It contains the token number of the expected token or 0 when the error occurs at a point where there is more than one token possible (several other tokens could be expected).

2. According to [Jac94]:

   > The second parameter contains a list of tokens that are not to be skipped at the error point.

   This list may be used to implement a kind of panic mode error-handling technique, in which all tokens are skipped until a token from the list is found in the input. This is almost the method LLgen normally uses to recover from errors. First of all tokens are skipped until a token is found that is an "acceptable", i.e. an element in the list of tokens, then LLgen inserts tokens until the parser is in a state in which it can parse the now first token.

Although these parameters are very useful in some error-handling implementations, they are not in very useful in our implementation, because we can find out what the expected token is[1]. We do not need the second parameter, because that information is only interesting if we want to skip tokens. But we want to parse all of the input and only correct the input with the most probable correction. To calculate the most probable corrections, we need the Data Oriented Parsing method, described in chapter 9.

[1] We know all possible expected tokens, because we know all current items. The possible expected tokens are the non-terminals in the items to be read.

Chapter 11

Error correction

Now that we have a parser that can handle ambiguous input and calculate the most probable parse, we still cannot correct errors with it. The problem is, if the input contains an error, *there is no most probable parse, there is no parse at all.*

In order to handle errors correctly, we must take a look at what errors can occur. In [Tho76], Thompson brings up an interesting idea. The input can be seen as a *channel*, see figure 11.1. In this channel some noise is added. The noise "makes" the errors: the data before the addition of the noise is "correct" and the data after adding the noise may be incorrect. The *noise generation* will change the data with a certain probability. The noise generation may make several changes in the input, most often the following changes are recognized:

- a symbol is inserted in the input,
- a symbol is deleted from the input,
- a symbol is changed into another symbol,
- two adjacent symbols are exchanged (transposition).

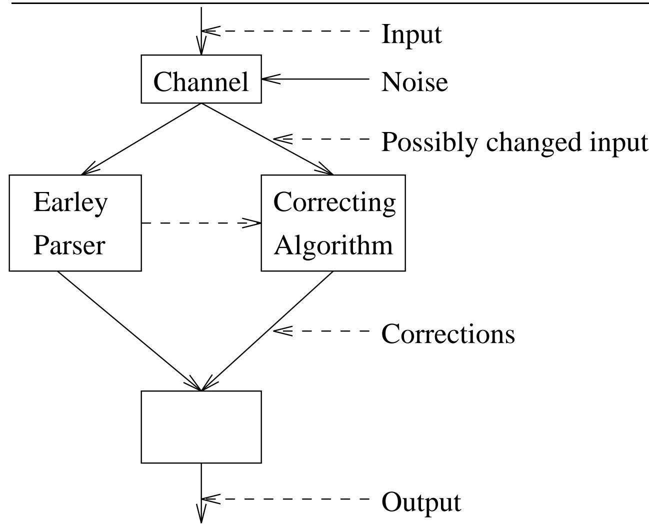
We will only assume the first two possible changes, because all other changes can be made by these two changes only.

After the noise is added to the input, the input goes to a parser and a correction algorithm. (In our case the correction algorithm is only "turned on" when an error is detected in the input.) In order to correct the above changes in the input, the opposite actions have to be taken, so to correct

an inserted symbol a symbol has to be deleted (the noise generation had inserted a symbol that should not be in the correct input),

a deleted symbol a symbol has to be inserted (the noise generation had deleted a symbol that should have been present in the correct input).

Figure 11.1 Error-correction system with channels



11.1 Handling errors in DOP

In our project, we only consider two possible errors in the input: deleted symbols, corrected by inserting symbols and inserted symbols, corrected by deleting symbols. When we want to change the implementation to consider other error correction operations as well, for example changing of symbols or transposition, we need to adjust the algorithms sketched here.

We added a `mode` flag in each edge. With normal parsing this flag is set to `match`, when inserting symbols it is set to `insert` and when deleting symbols it is set to `delete`. We will write this as [A→ϕ•ψ@k,`mode`].

Note that `mode` is only set to `delete` or `insert` if a terminal symbol is deleted or inserted (non-terminals are not deleted or inserted). This also means that when generating a derivation from the list of statesets, all used items need to be considered. Information about an inserted or deleted token is only present in the item added to the stateset by the insert or delete algorithm.

11.1.1 Handling an inserted symbol

Handling an inserted symbol from the original input is quite easy. We assume a symbol was inserted in the input; to correct it we must “skip” a symbol. This can be accomplished by pretending to have read the symbol, which means we can just copy some items to the next state. We have chosen to copy only the items which try to accept a terminal, to keep the algorithm simpler.

If [X→ϕ•aψ@k,m] is an item(tree) in state l, we add the item(tree) [X→ϕ•aψ@k,`delete`] to state $l+1$. This way it is just as if the symbol

in the input ($\neq$ a) was not part of the input (we skipped the symbol and tried the same item again on the next symbol in the input).

A graphical description of this algorithm can be found in figure 11.2.

Figure 11.2 Correcting an inserted symbol

stateset x		stateset x+1
[X$\rightarrow\phi\bullet$aψ@k,m]	$\longrightarrow$	[X$\rightarrow\phi\bullet$aψ@k,delete]
⋮		⋮

11.1.2 Handling a deleted symbol

When we expect a token to be deleted from the "original" input, we try to insert the possible tokens. We do this by the algorithm given in algorithm 11.1.

This rather complex algorithm is necessary, because we are "illegally" inserting items in a stateset. In order to let the parsing continue correctly, we first tried to just add the items into the stateset, but this did not work. When an item was complete, it could be used multiple times to reduce other items. Because of this, it would be (at least) difficult, if not impossible, to get the correct disambiguation information after parsing. In order to stop items being used multiple times, a kind of border had to be set in the stateset. This way it is possible to see which items are already used an which still can be used.

This algorithm inserts all possible tokens in a certain place, but does not insert multiple tokens followed by eachother. It only handles only *one* deleted token at a time. To let it handle multiple deleted tokens, the algorithm has to be started multiple times.

After some minor modifications in algorithm 2.2 it is possible to compute a parse together with the corrections used to generate this parse. This algorithm is described in section 11.2.

11.2 Correcting the input

When we have parsed the input with error correction, we want to have a derivation from it. Although algorithm 2.2 will generate a derivation from a list of statesets, we cannot use it directly, because the list of statesets we have generated by parsing the input with error correction is not a list of statesets the original Earley parser generates (see algorithm 2.1). So we need to adjust the algorithm a little bit, to enable the algorithm to handle the list of statesets with corrected errors in it. If the algorithm is given a list of statesets without any corrections in it (all the items in the statesets will

Algorithm 11.1 Inserting items in a stateset when inserting a symbol

Input

A list of statesets, with current state = I_l and a structured corpus.

Output

A new list of statesets, with new current state = I_l', with all possible, expected symbols inserted at state l, and a list of subsets $I_{l_0}, I_{l_1}, \ldots$ of I_l' defining an ordering on stateset I_l'. (For each possible item, only one symbol is inserted. No multiple insertions will be provided for.)

Method

$I_{l_0}, I_{l_1}, \ldots$ are initially empty subsets.

1. Set $I_{l_0} = I_l$ and set the current subset number, $s = 0$.
2. For each item in stateset I_{l_s} that is of the form [X→ϕ•aψ@k,m], add [X→ϕa•ψ@k, insert] to stateset $I_{l_{s+1}}$.
3. Let [A→τ•@i,m] be an item in stateset $I_{l_{s+1}}$. Examine stateset I_i for items of the form [X→ϕ•Aψ@k,n]. For each one found, add [X→ϕA•ψ@k,match] to stateset $I_{l_{s+1}}$.
4. Let [A→ϕ•Bψ@i, m] be an item in $I_{l_{s+1}}$. For all B→τ in the corpus, add [B→•τ@l,match] to stateset $I_{l_{s+1}}$.
5. Set $I_l' = I_l' \cup I_{l_{s+1}}$.
6. If stateset $I_{l_{s+1}} = \emptyset$ then replace I_l with I_l' and halt, else set s to $s + 1$ and goto (3).

have `match` as their `mode`), the algorithm is the same as the regular Earley parser, see algorithm 2.2. The revised algorithm is given in algorithm 11.2.

11.3 Overview

Now that we have discussed all separate pieces, we can give an overview of the complete system. There are many possible implementations, of which we will describe four in chapter 12. Each implementation consists of more or less the same steps:

- Parse using a normal (Earley) parser until an error is found.
- Correct the error by inserting or deleting one or more symbols. This is where algorithm 11.1 and the algorithm described in section 11.1.1 come into play. After correction of the error, the parsing may continue until all input is parsed or until another error is found, which can again be corrected using the algorithms.
- After parsing, we need to calculate the most probable parse. This is done by Monte Carlo disambiguation as described in section 9.3.1. The Monte Carlo disambiguation algorithm uses random derivations generated using algorithm 11.2.
- When we have the most probable parse with a list of corrections, it is possible to correct the input.

Algorithm 11.2 Calculating a derivation with corrections

Input

A structured corpus, an input string $\mathsf{w} = \mathsf{a}_1\mathsf{a}_2\ldots\mathsf{a}_n$ and a list of statesets $I_0, I_1, \ldots, I_n$ for w.

Output

π, a parse for w together with a list of corrections C, or an "error" message.

Method

If no item of the form $[\mathsf{S}\rightarrow\tau\bullet@0,\mathtt{m}]$ is on I_n, then w is not in $\mathcal{L}(\mathsf{G})$, so emit "error" and halt. Otherwise, initialize the parse π to empty and execute the routine $\mathbf{R}([\mathsf{S}\rightarrow\tau\bullet@0,\mathtt{m}], n)$ where the routine $\mathbf{R}$ is defined as follows.

Routine $\mathbf{R}([\mathsf{A}\rightarrow\phi\bullet@i,\mathtt{r}], j)$:

1. Let π be the previous value of π followed by h, where h is the number of production $\mathsf{A}\rightarrow\phi$. (We assume π is a global variable).

2. Let $\phi = \mathsf{X}_1\mathsf{X}_2\ldots\mathsf{X}_m$, set $k = m$ and $l = j$.

3. If `r`=`match` and $k > 0$
   - If $\mathsf{X}_k \in V_T$, subtract 1 from both k and l. Add `match` at the end of C.
   - If $\mathsf{X}_k \in V_N$
     - If l=j then let $[\mathsf{A}\rightarrow\mathsf{X}_1\mathsf{X}_1\ldots\mathsf{X}_k\bullet\mathsf{X}_{k+1}\ldots\mathsf{X}_m@i,\mathtt{p}]$ be an item in subset I_{j_z}. Find an item $[\mathsf{X}_k\rightarrow\phi\bullet@s,\mathtt{q}]$ in $I_{j_0} \cup \cdots \cup I_{j_{z-1}}$.
     - If $l < j$ then find an item $[\mathsf{X}_k\rightarrow\phi\bullet@s,\mathtt{q}]$ in stateset I_l for some s such that $[\mathsf{A}\rightarrow\mathsf{X}_1\mathsf{X}_2\ldots\mathsf{X}_{k-1}\bullet\mathsf{X}_k\ldots\mathsf{X}_m@i,\mathtt{p}]$ is in I_s.
     - Execute $\mathbf{R}([\mathsf{X}_k\rightarrow\phi\bullet@s,\mathtt{q}], l)$. Subtract 1 from k and set l to the return value of $\mathbf{R}$.

   else if `r`=`insert`
   - Subtract 1 from k.
   - Add `insert` at the end of C.

   else if `r`=`delete`
   - Subtract 1 from l.
   - Add `delete` at the end of C.

4. If $[\mathsf{A}\rightarrow\mathsf{X}_1\mathsf{X}_2\ldots\mathsf{X}_{k-1}\bullet\mathsf{X}_k\ldots\mathsf{X}_m@i,\mathtt{r}]$ is in I_{l_t} find an item $[\mathsf{A}\rightarrow\mathsf{X}_1\mathsf{X}_2\ldots\mathsf{X}_k\bullet\mathsf{X}_{k+1}\ldots\mathsf{X}_m@i,\mathtt{u}]$ in $I_{l_0} \cup \cdots \cup I_{l_{t-1}}$. Then set `r` to `u` and let ϕ=$\mathsf{X}_1\ \mathsf{X}_2\ldots\mathsf{X}_k$.

5. If $k \geq 0$ then goto step (3).

6. Set the return value of $\mathbf{R}$ to l.

Chapter 12

Different implementations

This chapter will describe different implementations, of which we have implemented some. The others can easily be implemented with the existing framework and some small changes.

12.1 A naive way: DOP 2

The first implementation we made was a system described by Bod in [Bod95]. It is called DOP 2 (from the range DOP 1–DOP 6). DOP 1 was the basic system as described here in section 3.2. DOP 2 is an extension of DOP 1 in that it can handle *unknown words*.

DOP 2 is essentially the same as the DOP 1 system. The only difference is that with DOP 2 the corpus does not have to be complete. We assumed in our system that the corpus used is complete, but because this implementation is the same as the DOP 2 implementation, we call this implementation DOP 2.

In DOP 2, when an unknown word is encountered, that unknown word is replaced by all possible words known (actually the *part-of-speech* tags are inserted, not the actual words). Then parsing and disambiguation may continue. The most probable parse then assigns the most probable part-of-speech tag to the unknown word.

12.1.1 Correcting the input: Replacement-only correction

Although we do not have unknown words (and even no unknown part-of-speech tags as we have in DOP, see DOP 3 in [Bod95]), our idea was to change an erroneous symbol in the input to all possible symbols, then continue parsing and disambiguating. This implementation works, but after some reflection, it could be made a lot simpler (and faster). Instead of changing the erroneous symbol into all possible symbols, we change it into all expected symbols. In the state where the error occurred, search all items

of the form of [$\mathsf{X}\rightarrow\phi\bullet\mathsf{a}\psi@k$] and pretend that the current token on the input was a. To simulate the parsing as if the expected token *was* on the input, [$\mathsf{X}\rightarrow\phi\mathsf{a}\bullet\psi@k$] is added to the next stateset. This way the erroneous symbol *changes* exactly into the expected symbol.

Note that this type of corrections does not use the algorithms given in section 11.1.

12.1.2 Results

The results of this first implementation were good, but a couple of problems could be easily spotted.

advantages

- If the error had been a replacement of a symbol, this system calculates the most probable parse and changes the erroneous symbol in the most probable symbol.
- The system is pretty simple. No complex insertion and deletion corrections are needed.

disadvantages

- If the error had been an insertion or deletion, a lot of error corrections (including error messages) may be generated, because it *changes* for example the inserted symbol into the right one, but after the inserted symbol, the expected symbol is on the input. This symbol is also changed, because the inserted symbol was changed into this symbol, and so on. See figure 12.1 for an example.
- Because the input has only been *changed*, the length of the input is still the same. This means the length of the input *must* generate a correct program. Fortunately, if it is possible to generate a correct program with that length (starting from the error point), the parser will do so.

12.2 A good way: DOPPER 1

The major disadvantage of the DOP 2 method is that the length of the program cannot change. We defined another system that assumed symbols could have been inserted or deleted. DOPPER 1[1] is an extension of the DOP 2 method. It can correct errors using insertions and deletions and it can also (in some cases) simulate the DOP 2 correction (replacement of symbols). This is done by inserting a symbol and directly after that deleting another symbol.

[1]DOPPER stands for *D*ata *O*riented *P*arsing used in *P*ersistent *E*rror *R*ecovery.

Figure 12.1 Input with "wrong" correction

incorrect input sentence

int + int int + int + - int

correction according to DOP 2

int	+	int	+	int	+	int	-	int
-	-	-	int→ +	+→ int	int→ +	+→int	-	-

more probable correction

int	+	int		+	int	+	-	int
-	-	-	delete int	-	-	-	-	-

Here the first line describes the (possibly corrected) input. The second line (if present) reflects the corrections made. In the second line a - denotes no change in the input, x→y denotes the changing of x in the original input to y and delete x denotes the deleting of x in the original input.

12.2.1 Correcting the input: Simple insert and delete

When an error occurs, the error correction routine is called. The error correction routine uses the information of the part of the input already parsed and it tries to insert the possible symbols and delete a symbol from the input. Now parsing can continue. The routine continues parsing until all input is parsed or another error is encountered. When this happens, again it tries to correct the input by insertion and deletion. The way the corrections are handled in the chart are described in section 11.1.

Note that in DOPPER 1 each time only 1 symbol is inserted or deleted. After that parsing can continue, but maybe after parsing one symbol again a correction is needed.

A replacement of a symbol occurs when deleting a symbol will not enable the parser to continue parsing, so a symbol will be inserted. Parsing can continue now. If directly after the inserted symbol another error point occurs, this time it could be the case that a symbol can be deleted to let the parsing continue. First a symbol is inserted and directly after that a symbol is deleted, resulting in a replacement. See figure 12.2 for an example.

12.2.2 Results

The results of the second implementation were a lot better, with only some errors "corrected" incorrectly.

advantages

- When this method is used, changes can be made to change the length of the program. But the correction by changing a symbol

Figure 12.2 DOPPER 1 replacing a symbol

Input program:	Corrections:	Corrected program:
int	match 'int'	int
main(void) {	match 'main'	main(void) {
if(1) {	match '('	if(1) {
return 1)	match 'void'	return 1;
}	match ')'	}
}	match '{'	}
	match 'if'	
	match '('	
	match '1'	
	match ')'	
	match '{'	
	match 'return'	
	match '1'	
	insert ';'	
	delete ')'	
	match '}'	
	match '}'	

can be made, too. This is done by inserting the "new" symbol, parse it, then in the next state delete the old symbol.

- This method uses the DOP system and computes the most probable error correction by inserting or deleting *one* symbol.

disadvantages

- Not all corrections are possible. For example multiple deletions are not possible.

12.3 A better way: DOPPER 2

The method described here takes care of the disadvantage of the previous system in which not all possible (combinations of) corrections are possible.

12.3.1 Correcting the input: Multiple insert and delete

The system described here is almost the same as the previous one. Just like the previous system, when an error is encountered, first all possible symbols are inserted and a symbol from the input is deleted. Then again all possible symbols are inserted and a symbol from the input is deleted. Because the insertion of a token creates a new item in the current stateset, it might

generate a new possible token, so that might create a new item in the next stateset by deleting it.

A problem occurs when we repeatedly insert tokens. If we use a non-terminal that is defined recursively, the algorithm will again and again try to generate that non-terminal. This process will never end, therefore we need to restrict the algorithm. This can be done in several ways, by limiting the number of insertions or by considering the probability of inserting a token.

Our opinion is that the program will not contain six successive deleted tokens, so after using the algorithm described in section 11.1.1 six times we stop inserting symbols. Six is just an arbitrary number. If six is not enough, larger numbers can be used, as long as there is a boundary.

Our implementation limits the number of inserted tokens by considering the probability of the inserted tokens (if the probability of an inserted token is smaller than a certain level, do not insert).

This system is more powerful than the previous one, because with this system, multiple deletions *are* possible.

12.3.2 Results

advantages

- This system can change the input in all possible ways. All possible corrections can be generated.
- This system correctly changes the input to the most probable input, by generating the most probable parse.

disadvantages

- We have assumed that the input until the error point was correct, but the actual error could have been before the error point. Although LLgen has the correct prefix property, the input *before* the error point might need a correction to get the "real" most probable parse.

12.4 The best way: DOPPER 3

The only way to compute the "real" most probable parse is to take the complete program as the context. This is a global error-handling method, in contrast to the previous methods, which were regional.

12.4.1 Correcting the input: Correct and restart

With this system, when an error occurs, again the error-handling method is called. This time, parsing is started not at the error point, but at the

beginning of the program. This way, a most probable parse of the complete program is generated.

Of course, when parsing the complete program, the previous system is used. This way all possible combinations of error corrections are considered. When the most probable parse (or the most probable program) is calculated, LLgen is started all over again, but now with the corrected input. This time, no errors will be found, because we have given LLgen correct input. This method first computes the correct input, then starts parsing again but with the correct input. This stands in contrast to the other methods, which calculate the most probable correction and *continue* parsing from the error point.

12.4.2 Results

advantages

- Errors will be corrected even before the found error point.
- Error correction only takes place once.

disadvantages

- All of the input is considered with error correction, even if the first part (until the error point) *is* correct. This will take more time.
- LLgen needs to be started again (with different input).

12.5 Results

12.5.1 Examples

This section contains some sample parses, together with the corrections given by the various implementations. The examples are given in figures 12.3 and 12.4.

First, the incorrect program is given. Second, the corrections, generated by the different systems, are shown. Last, the corrected programs are given.

Note that the results given here may vary depending on the corpus used and the probabilities of insertion and deletion.

Also note that we changed the DOP 2 method in order to let it generate a parse. If necessary, DOP 2 may insert tokens when there is no input left to change. This method is used in figure 12.4.

A parse together with its list of statesets can be found in section B.

Example 1

In figure 12.3, the DOP 2 method works good. It parses until the error point, the second '1'. DOP 2 tries to change the second '1' into something else, so the parsing may continue. The ';' was most probable, so DOP 2 changes the second '1' into the ';'.

DOPPER 1 does not work correctly. It parses until the error point and tries to insert a token or delete a token. Because deleting the second '1' does not let the parsing continue, inserting a symbol is the only way. DOPPER 1 inserts a ';' and parsing may continue. Again an error is found, when DOPPER 1 considers the first '}' directly following the second '1'. Again deleting a symbol does not let the parsing continue, so inserting a symbol is again the only way. DOPPER 1 inserts another ';' and parsing can complete.

DOPPER 2 can delete and insert multiple tokens at the error point. Therefore DOPPER 2 does not make the error by DOPPER 1. At the error point DOPPER 2 deletes the '1' and inserts a ';'. This way a more probable parse can be generated.

DOPPER 3 is in this case the same as the DOPPER 2 method. DOPPER 3 finds the error and starts all over again, generating the same parse as the DOPPER 2 method.

Example 2

In example 2, figure 12.4, DOP 2 does not work very well. DOP 2 parses until the error point, the ';'. It then changes the ';' in a ')'. Parsing can continue, but following the ')' a ';' should be present, so the next error corrected by changing the '}' into a ';'. The parsing can continue again, but the program misses a '}', so DOP 2 has to insert a '}' at the end of the program. This time DOP 2 has corrected the program using three corrections.

DOPPER 1 works better this time. It parses until it reaches the error point. At that point it may insert or delete one token, because deletion of a token does not let parsing continue, a token is inserted. DOPPER 1 inserts a ')'. Parsing can continue until the end of the program.

DOPPER 2 works exactly the same way as DOPPER 1 this time. It parses until it reaches the error point. At that point it may insert or delete one or more tokens. Just like with DOPPER 1, inserting a ')' is enough.

DOPPER 3 does an even better job. The standard parser of the compiler parses until an error is found and DOPPER 3 is started. DOPPER 3 starts parsing all over again and at each token it tries if changing tokens in the input generates a more probable parse. Double '("s are not very probable, so DOPPER 3 removes the second '('. This way an even more probable parse is generated.

Figure 12.3 Example 1

Input program:

```
int
main(void) {
  if(1) {
    return 1 1
  }
}
```

DOP 2	DOPPER 1	DOPPER 2	DOPPER 3
Corrections:			
match ‘int’	match ‘int’	match ‘int’	match ‘int’
match ‘main’	match ‘main’	match ‘main’	match ‘main’
match ‘(’	match ‘(’	match ‘(’	match ‘(’
match ‘void’	match ‘void’	match ‘void’	match ‘void’
match ‘)’	match ‘)’	match ‘)’	match ‘)’
match ‘{’	match ‘{’	match ‘{’	match ‘{’
match ‘if’	match ‘if’	match ‘if’	match ‘if’
match ‘(’	match ‘(’	match ‘(’	match ‘(’
match ‘1’	match ‘1’	match ‘1’	match ‘1’
match ‘)’	match ‘)’	match ‘)’	match ‘)’
match ‘{’	match ‘{’	match ‘{’	match ‘}’
match ‘return’	match ‘return’	match ‘return’	match ‘return’
match ‘1’	match ‘1’	match ‘1’	match ‘1’
change ‘1’ → ‘;’	**insert** ‘;’	**delete** ‘1’	**delete** ‘1’
match ‘}’	match ‘1’	**insert** ‘;’	**insert** ‘;’
match ‘}’	**insert** ‘;’	match ‘}’	match ‘}’
	match ‘}’	match ‘}’	match ‘}’
	match ‘}’		
Corrected programs:			
int	int	int	int
main(void) {	main(void) {	main(void) {	main(void) {
if(1) {	if(1) {	if(1) {	if(1) {
return 1;	return 1; 1;	return 1;	return 1;
}	}	}	}
}	}	}	}

Figure 12.4 Example 2

Input program:

```
int
main(void) {
  ((1+1);
}
```

DOP 2	DOPPER 1	DOPPER 2	DOPPER 3
Corrections:			
match ‘int’	match ‘int’	match ‘int’	match ‘int’
match ‘main’	match ‘main’	match ‘main’	match ‘main’
match ‘(’	match ‘(’	match ‘(’	match ‘(’
match ‘void’	match ‘void’	match ‘void’	match ‘void’
match ‘)’	match ‘)’	match ‘)’	match ‘)’
match ‘{’	match ‘{’	match ‘{’	match ‘{’
match ‘(’	match ‘(’	match ‘(’	match ‘(’
match ‘(’	match ‘(’	match ‘(’	**delete** ‘(’
match ‘1’	match ‘1’	match ‘1’	match ‘1’
match ‘+’	match ‘+’	match ‘+’	match ‘+’
match ‘1’	match ‘1’	match ‘1’	match ‘1’
match ‘)’	match ‘)’	match ‘)’	match ‘)’
change ‘;’ → ‘)’	**insert** ‘)’	**insert** ‘)’	match ‘;’
change ‘}’ → ‘;’	match ‘;’	match ‘;’	match ‘}’
insert ‘}’	match ‘}’	match ‘}’	
Corrected programs:			
int	int	int	int
main(void) {	main(void) {	main(void) {	main(void) {
((1+1));	((1+1));	((1+1));	(1+1);
}	}	}	}

12.5.2 Practical results

Although this project is a more theoretical one, the DOP 2, DOPPER 1 and DOPPER 2 methods are implemented in a compiler. These system are primarily implemented to check if the methods work at all and how certain corrections are made. Not much attention is paid to the time and space complexities. The implementations of the DOP framework used in the compiler are very simple, but also very inefficient. Other implementations of the DOP framework have been devised, but are more complex to implement.

The programs we have tested did not use a complete corpus; the corpus did not describe the complete C language. It consisted of approximately 1000 elementary subtrees, generated from roughly 10 programs. We used a very small corpus, so the execution times of the programs would not drastically increase. Execution times were several minutes on a SPARCstation 4, when compiling small programs. The size of these programs were roughly like the size of the example programs given in figure 12.3 and 12.4.

We did not test the implementation by letting it compile various programs, because we had to limit the corpus too much. Limiting the corpus automatically limits the number of C constructs that could be used in the test programs.

12.6 Insertion and deletion probabilities

All trees in the chart have a certain probability. The trees taken directly from the corpus have a probability that can be calculated from the number of occurrences in the corpus (see section 9.3).

There are three reasons why trees are inserted into a stateset[2]:

Normal During parsing (without corrections), we want to keep the probability of the tree itself. This is exactly the way the probabilities work in DOP.

Insertion When a symbol is inserted, an item is inserted, but we do not want the probability of that item to be the same as the item it originated from, because the insertion has a smaller chance than a normal match.

Deletion When a symbol is deleted, just like insertion, another probability has to be assigned.

As a probability for deletion and insertion, we took 0.05. This was easiest to implement, but this is probably not the best way.

The probability of insertion and deletion corrections are

[2]The three reasons stem from our implementation. If you acknowledge other corrections, like transposition, there are more possibilities.

1. probably not the same,
2. probably dependent on the writer of the program,
3. probably dependent on the corpus used,
4. probably dependent on context.

This is a complex problem not easily solved. We have chosen for small constants, just as in the original DOP system. It seems to work well, although some more complex function might work better.

Chapter 13

Conclusion

13.1 Goal

The goal of this project was to see if a system could be built that computes the most probable error correction in a compiler. Four requirements on the error handling routine were given and discussed.

1. It should never fail to return control to the parser, and when it does, parsing should be able to continue. The interval of text that escapes analysis in the course of attempting recovery from an error should be minimal.

2. It should not introduce spurious errors into an otherwise syntactically correct text fragment as a consequence of its actions in recovering from a single error.

3. It should not have higher space or time complexity than the parser itself.

4. It should not degrade the performance of the parser on error-free text segments.

Not all of these requirements are met in the systems described in this thesis, but other implementations of the system described here can be used to meet the requirements. This implementation was chosen, because it is easy to understand, whereas the other implementations are more complex.

13.2 Error-correction

An Earley parser cannot handle incorrect input, so some adjustments had to be made.

The systems as described here can only handle insertion and deletion of symbols in the input. All other error correction types, for example transposition, can be described by these two error corrections.

The following algorithms were devised:

- an algorithm to handle insertions in the statesets of the Earley parser,
- an algorithm to handle deletions in the statesets of the Earley parser,
- an algorithm to generate the parse from a list of statesets that contains corrections of errors in the input.

13.3 Selecting the most probable corrections

When compiling an incorrect program, often there are a lot of possible corrections. Our opinion is that the parser should use the most probable corrections. We select the most probable correction, by first computing all possible programs with their corrections and second selecting the most probable correction by disambiguating between all the most probable programs.

We chose to use the DOP system to disambiguate between the possible corrections. The DOP system selects the most probable program with its corrections based on a corpus of subtrees. This corpus serves as the "memory" of the parser. The corpus contains parts of correct programs. Probabilities of the parts of the programs are calculated, based on the number of occurrences of each part.

13.4 Different implementations

With the algorithms devised in this thesis, the following implementations were built.

DOP 2 The DOP 2 system as described in [Bod95]. The only possible error correction is the change of a symbol.

DOPPER 1 The DOPPER 1 system tries to correct the input by inserting or deleting *one* symbol from the error point. When the correction is made, the parsing can continue until another error is found.

DOPPER 2 The DOPPER 2 system tries to correct the input by inserting or deleting *multiple* symbols from the error point. When the corrections are made, parsing can continue until the end of the input is found. No more errors will be found, because all errors in the input are corrected the first time.

DOPPER 3 The DOPPER 3 system tries to correct the input by inserting or deleting multiple symbols, but the correction of the input can also take place before the error point. The DOPPER 3 system stops parsing completely, calculates the most probable input and starts parsing from the beginning.

13.5 Comparison

Error correction using DOP 2 is very simple. It works correctly when the error can be corrected using the replacement of a symbol. Most of the time, however, this method does not work correctly. It changes large parts of the programs, where a simple insertion or deletion of a symbol would have corrected the error; see for example figure 12.1.

DOPPER 1 is the first system that tries to correct errors by inserting and deleting symbols. It corrects an error by inserting or deleting one symbol. DOPPER 1 corrects the programs in a better way than DOP 2 most of the time, but it cannot correct by deleting 2 successive symbols. This can lead to incorrect results as demonstrated in figure 12.3.

DOPPER 2 corrects errors just as the DOPPER 1 system, but can also handle multiple insertions and deletions. DOPPER 2 performs better than the DOPPER 1 system. In figure 12.3 DOPPER 2 corrects the program in a better way than the DOPPER 1 system does.

The DOPPER 3 system is the best system in that it corrects the program in the most probable way possible. In figure 12.4, DOPPER 3 is the only system that can correct the program in the most probable way. This is because DOPPER 3 completely restarts parsing. The disadvantage is that DOPPER 3 is less efficient in time and space. When DOPPER 3 finds an error, it discards all of the previously found tree structure. The discarded piece of tree structure needs to be parsed again, resulting in extra overhead.

The technique, which is used in the previously described systems, is an interesting technique. The major disadvantage is that it is at the moment still inefficient in time and space.

Appendix A

Computer implementation

This section contains information about the actual computer implementation. First, the main program will be described together with its usage. After that, the class structure will be depicted.

The actual implementation consists of two programs. Most important the C compiler, second `c2d`, which translates a corpus to a corpus consisting of elementary trees.

A.1 C-compiler

The C compiler used had already been written, so we will only describe the extensions we have implemented.

From the outside the extensions we have implemented can be noticed by several parameters, which we will describe here shortly.

A.2 `-e` flag

The `-e` flag was implemented so the level on which the error correction should take place could be altered. As described in section 9.2.2 the compiler was designed to be used to the error correction could take place on several levels, `expression`, `statement`, `block` or `program`. This level can still be set on runtime, although it has not been tested. This is actually not used very much, because of the problems as described in section 9.2.2.

After the `-e` flag, an integer may be given. If no integer is given, the default value of four is set. The possible values are:

1. The level of error correction will be `expression`.
2. The level of error correction will be `statement`.
3. The level of error correction will be `block`.
4. The level of error correction will be `program`.

If an integer other than these is given, the default value will be used.

A.3 `−f` flag

With the `−f` flag it is possible to select the different error handling schemes. We have implemented the DOP2 error handling scheme, as described in section 12.1, the DOPPER1 error handling scheme, as described in section 12.2 and the DOPPER2 error handling scheme, as described in section 12.3.

The different schemes can be selected by putting an integer between one and three behind the `−f` flag. The default value is three.

1. DOP2 error handling scheme.
2. DOPPER1 error handling scheme.
3. DOPPER2 error handling scheme.

If an integer other than these is given, the default value will be used.

A.4 `−c` flag

The C compiler can also be used to fill a corpus of tree structures describing the structure of the compiled C program. The format of the corpus is described in section 9.2.

The compiler can be used to generate tree structures of various different levels. An integer directly after the `−c` flag indicates what level tree structure will be generated.

1. The root of the tree(s) generated is `expression`.
2. The root of the tree(s) generated is `statement`.
3. The root of the tree(s) generated is `block`.
4. The root of the tree generated is `program`.

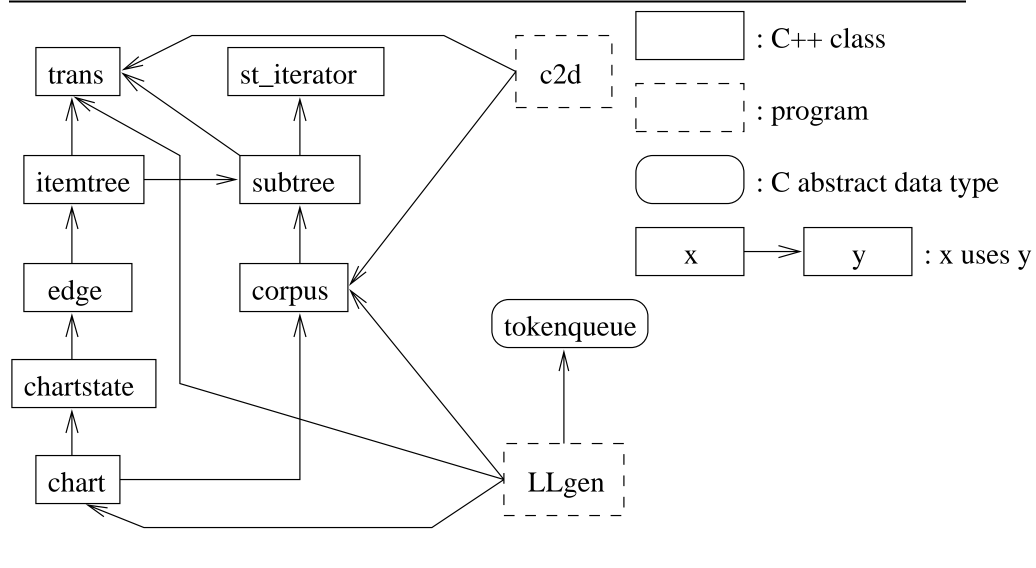
If an integer other than these is given, the default value will be used.

The default value is four, so when no integer is present, the compiler will generate tree structures with `program` as its root.

The tree structures are written to a file, `corpus`. When the file already exists, tree structures will be added.

A.5 Class structure

Figure A.1 Class structure



Appendix B

Example parse

This example parse was generated using the DOPPER 2 error correction scheme.

B.1 Program code

```
int
main(void) {
   return 9 9.1;
}
```

B.2 Statesets

STATE [0]
0 tree(program[tree(external_definition[])]).item0 1 match
0 tree(external_definition[tree(decl_specifiers[])tree(declarator[])
tree(function[])]).item0 1 match
0 tree(decl_specifiers[tree(single_decl_specifier[])]).item0 1 match
0 tree(single_decl_specifier[tree(int[])]).item0 0.5 match
0 tree(single_decl_specifier[tree(void[])]).item0 0.5 match

STATE [1]
0 tree(int[]).item0 1 match
0 tree(single_decl_specifier[tree(int[])]).item1 1 match
0 tree(decl_specifiers[tree(single_decl_specifier[])]).item1 1 match
0 tree(external_definition[tree(decl_specifiers[])tree(declarator[])
tree(function[])]).item1 1 match
1 tree(declarator[tree(primary_declarator[])tree('('[])
tree(parameter_type_list[])tree(')'[])]).item0 1 match
1 tree(primary_declarator[tree(identifier[])]).item0 1 match

1 tree(identifier[tree(identifier[])]).item0 1 match

STATE [2]
1 tree(identifier[]).item0 1 match
1 tree(identifier[tree(identifier[])]).item1 1 match
1 tree(primary_declarator[tree(identifier[])]).item1 1 match
1 tree(declarator[tree(primary_declarator[])tree('(')[])
tree(parameter_type_list[])tree(')'[])]).item1 1 match

STATE [3]
2 tree('('[]).item0 1 match
1 tree(declarator[tree(primary_declarator[])tree('('[])
tree(parameter_type_list[])tree(')'[])]).item2 1 match
3 tree(parameter_type_list[tree(parameter_decl_list[])]).item0 1 match
3 tree(parameter_decl_list[tree(parameter_decl[])]).item0 1 match
3 tree(parameter_decl[tree(decl_specifiers[])
tree(parameter_declarator[])]).item0 1 match
3 tree(decl_specifiers[tree(single_decl_specifier[])]).item0 1 match
3 tree(single_decl_specifier[tree(int[])]).item0 0.5 match
3 tree(single_decl_specifier[tree(void[])]).item0 0.5 match

STATE [4]
3 tree(void[]).item0 1 match
3 tree(single_decl_specifier[tree(void[])]).item1 1 match
3 tree(decl_specifiers[tree(single__decl_specifier[])])
.item1 1 match
3 tree(parameter_decl[tree(decl_specifiers[])
tree(parameter_declarator[])]).item1 1 match
4 tree(parameter_declarator[tree(primary_parameter_declarator[])])
.item0 1 match

STATE [5]
4 tree(primary_parameter_declarator[]).item0 1 match
4 tree(parameter_declarator[tree(primary_parameter_declarator[])])
.item1 1 match
3 tree(parameter_decl[tree(decl_specifiers[])
tree(parameter_declarator[])]).item2 1 match
3 tree(parameter_decl_list[tree(parameter_decl[])]).item1 1 match
3 tree(parameter_type_list[tree(parameter_decl_list[])]).item1 1 match
1 tree(declarator[tree(primary_declarator[])tree('('[])
tree(parameter_type_list[])tree(')'[])]).item3 1 match

STATE [6]
5 tree(')'[]).item0 1 match

1 tree(declarator[tree(primary_declarator[])tree('('[])
tree(parameter_type_list[])tree(')'[])]).item4 1 match
0 tree(external_definition[tree(decl_specifiers[])tree(declarator[])
tree(function[])]).item2 1 match
6 tree(function[tree(compound_statement[])]).item0 1 match
6 tree(compound_statement[tree('{'[])tree(return_statement[])
tree('}'[])]).item0 1 match

STATE [7]
6 tree('{'[]).item0 1 match
6 tree(compound_statement[tree('{'[])tree(return_statement[])
tree('}'[])]).item1 1 match
7 tree(return_statement[tree(return[])tree(expression[])tree(';'[])]).item0 1 match

STATE [8]
7 tree(return[]).item0 1 match
7 tree(return_statement[tree(return[])tree(expression[])tree(';'[])]).item1 1 match
8 tree(expression[tree(assignment_expression[])]).item0 1 match
8 tree(assignment_expression[tree(conditional_expression[])]).item0 1 match
8 tree(conditional_expression[tree(binary_expression[])]).item0 1 match
8 tree(binary_expression[tree(postfix_expression[])]).item0 1 match
8 tree(postfix_expression[tree(constant[])]).item0 1 match
8 tree(constant[tree(integer[])]).item0 1 match

STATE [9]
8 tree(integer[]).item0 1 match
8 tree(constant[tree(integer[])]).item1 1 match
8 tree(postfix_expression[tree(constant[])]).item1 1 match
8 tree(binary_expression[tree(postfix_expression[])]).item1 1 match
8 tree(conditional_expression[tree(binary_expression[])]).item1 1 match
8 tree(assignment_expression[tree(conditional_expression[])]).item1 1 match
8 tree(expression[tree(assignment_expression[])]).item1 1 match
7 tree(return_statement[tree(return[])tree(expression[])tree(';'[])]).item2 1 match
7 tree(return_statement[tree(return[])tree(expression[])tree(';'[])]).item3 0.1
insert
6 tree(compound_statement[tree('{'[])tree(return_statement[])
tree('}'[])]).item2 1 match

STATE [10]
9 tree(floating[]).item0 1 match
7 tree(return_statement[tree(return[])tree(expression[])tree(';'[])]).item2 0.1
delete
7 tree(return_statement[tree(return[])tree(expression[])tree(';'[])]).item3 0.01
insert

6 tree(compound_statement[tree('{'[])tree(return_statement[])tree('}'[])]).item2
1 match

STATE [11]
10 tree(';'[]).item0 1 match
7 tree(return_statement[tree(return[])tree(expression[])tree(';'[])]).item2 0.01
delete
7 tree(return_statement[tree(return[])tree(expression[])tree(';'[])]).item3 0.001
insert
7 tree(return_statement[tree(return[])tree(expression[])tree(';'[])]).item3 0.1
match
6 tree(compound_statement[tree('{'[])tree(return_statement[])
tree('}'[])]).item2 1 match

STATE [12]
11 tree('}'[]).item0 1 match
7 tree(return_statement[tree(return[])tree(expression[])
tree(';'[])]).item2 0.001 delete
6 tree(compound_statement[tree('{'[])tree(return_statement[])
tree('}'[])]).item3 1 match
6 tree(function[tree(compound_statement[])]).item1 1 match
0 tree(external_definition[tree(decl_specifiers[])tree(declarator[])
tree(function[])]).item3 1 match
0 tree(program[tree(external_definition[])]).item1 1 match

B.3 List of corrections

match	(int)
match	(identifier)
match	('(')
match	(void)
match	(')')
match	('{')
match	(return)
match	(integer)
delete	(float)
match	(';')
match	('}')

B.4 Most probable parse

tree(program[

```
tree(external_definition[
  tree(decl_specifiers[
    tree(single_decl_specifier[
      tree(int[
      ])
    ])
  ])
  tree(declarator[
    tree(primary_declarator[
      tree(identifier[
        tree(identifier[
        ])
      ])
    ])
    tree('('[
    ])
    tree(parameter_type_list[
      tree(parameter_decl_list[
        tree(parameter_decl[
          tree(decl_specifiers[
            tree(single_decl_specifier[
              tree(void[
              ])
            ])
          ])
          tree(parameter_declarator[
            tree(primary_parameter_declarator[
            ])
          ])
        ])
      ])
    ])
    tree(')'[
    ])
  ])
  tree(function[
    tree(compound_statement[
      tree('{'[
      ])
      tree(return_statement[
        tree(return[
        ])
        tree(expression[
          tree(assignment_expression[
```

```
              tree(conditional_expression[
                tree(binary_expression[
                  tree(postfix_expression[
                    tree(constant[
                      tree(integer[
                      ])
                    ])
                  ])
                ])
              ])
            ])
          ])
          tree(';'[
          ])
        ])
        tree('}'[
        ])
      ])
    ])
  ])
]).
```

B.5 Corrected program code

```
int
main(void) {
   return 9;
}
```

Bibliography

[AP72] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parsing for context-free languages. *SIAM J. Computing*, 1(4):305–312, December 1972.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[AU72] Alfred V. Avo and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling - Volume 1: Parsing.* Prentice-Hall, Inc., 1972.

[Bod95] R. Bod. Enriching linguistics with statistics: Performance models of natural language. Master's thesis, Universiteit van Amsterdam, 1995.

[BS96] Rens Bod and Remko Scha. Data-oriented language processing: An overview. Technical report, ILLC: Institute for logic, language and computation, University of Amsterdam, 1996.

[Cha93] Eugene Charniak. *Statistical Language Learning.* The MIT Press, 1993.

[Ear70] Jay Early. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[GJ95] Dick Grune and Ceriel Jacobs. *Parsing Techniques - A Practical Guide.* Printout by the Authors, 1995.

[Iro63] E.T. Irons. An error-correcting parse algorithm. *Communications of the ACM*, 6(11):669–673, November 1963.

[Jac94] Ceriel J.H. Jacobs. LLgen, an extended LL(1) parser generator. Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1994.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., 1978.

[Lév74] J.-P. Lévy. Automatic correction of syntax-errors in programming languages. *Acta Informatica*, 4:271–292, 1974.

[LF77] Shin-Yee Lu and King-Sun Fu. Stochastic error-correcting syntax analysis for recognition of noisy patterns. *IEEE Transactions on computers*, C-26(12):1268–1276, December 1977.

[Lin90] Peter Linz. *An Introduction to Formal Languages and Automata*. D.C. heath and Company, 1990.

[Lip93] Stanley B. Lippman. *C++ Primer*. Addison-Wesley Publishing Company, 2nd edition, 1993.

[Lyo74] Gordon Lyon. Syntax-directed least-errors analysis for context-free languages: A practical approach. *Communications of the ACM*, 17(1):3–14, January 1974.

[Pai80] Ajit B. Pai. Global context recovery: A new strategy for syntactic error recovery by table-driven parsers. *ACM Transactions on Programming Languages and Systems*, 2(1):18–41, January 1980.

[Sima] K. Sima'an. Learning efficient parsing - with application to data oriented parsing and speech understanding. Research Institute for Language and Speech, Utrecht University.

[Simb] K. Sima'an. An optimized algorithm for data oriented parsing. Utrecht University.

[SL95] Alexander Stepanov and Meng Lee. The standard template library. Technical report, Hewlett-Packard Company, 1995.

[Tho76] Richard A. Thompson. Language correction using probabilistic grammars. *IEEE Transactions on Computers*, C-25(3):275–286, March 1976.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley Publishing Company, 1995.