Eindhoven University of Technology
Department of Electrical Engineering
Information and Communication Systems Group

Error Correction and Recovery in a LL(1) Parser

by J.C.C. Hermeler

practical training report

December 1998
Supervised by:
ir. Pieter J. Schoenmakers

Abstract

This report describes the various aspects of error recovery in general and for use with GP, one of the available TOM programs, in particular. GP is a program that generates recursive descent parsers, given an LL(1) grammar. With the possibility to recover from syntax errors, the parsers GP generates will be more useful.

The general idea of error handling is subdivided in three stages: error detection, error recovery and error correction. Each stage is briefly discussed in order to determine what can be expected from them. Next, several well known routines are presented. Each method is shortly explained and the main advantages and disadvantages are presented. Finally, the error corrector using insertions only is discussed in detail. Next to the original algorithm several optimizations are studied.

The main parts of the discussed error corrector have been implemented for GP. Based upon the research and this report it should not be difficult to finish the corrector in the nearby future.

Contents

Chapter 1

Introduction

The object-oriented programming language TOM, which is developed at the TUE, is supported by various tools and libraries. One of the tools is GP, which generates parsers. This program requires an annotated LL(1) attribute grammar as input. For this grammar, GP will produce a class that implements a recursive descent parser. This parser is able to process any input, as long as it is syntactically correct. If the input contains syntax errors, for instance caused by a typing error, the program will report a syntax error after encountering the first erroneous symbol and terminate. The user is reported at which line an error was detected so it can easily be found and corrected. This mechanism is very effective for small pieces of input. However, it becomes quite a burden when a complete computer program is parsed. One would prefer the parser to process the entire input and present a list with all errors detected. In order to achieve this, the parser has to be equiped with proper error handling capabilities.

The goal of this report is to investigate the three different stages of error handling: error detection, error recovery, and error correction. First of all, a detailed specification of those stages is presented. Why is there a distinction between them and what can be expected from them? Second, several interesting approaches to the theoretical and practical problems of error handling are discussed. Armed with this knowledge, one algorithm is selected for GP.

Chapter 2 provides an overview of the general idea of error handling. Various error correction and recovery techniques are discussed in chapter 3. Chapter 4 follows with details about the implementation of the selected algorithm. Finally, chapter 5 presents the conclusions and recommendations of this research.

Chapter 2

Error Handling

When a parser processes a certain input, it is possible that this input wil contain errors. For instance typing errors or misconceptions. These errors can not be ignored by the parser. There has to be a well defined behaviour in these situations. Depending on the circumstances, user specifications have to dictate what kind of behaviour is required. Will it be enough to halt the parser gracefully, with or without a decent error message? Or does the parser have to continue the process for as long as possible, looking for other mistakes. This chapter will discuss the three successive error handling stages: error detection, error recovery and error correction.

2.1 Error Detection

The most obvious part is that an error has to be detected by the parser before any action can be taken. A nice crash of the program is of course not considered to be an elegant error detection mechanism. As soon as a sentence is parsed which can not be described by the grammar of the language, a syntax error has to be detected. Nearly every parser will be able to do this without encountering great problems. It is more difficult to tell where exactly this error occured. It is very inconvenient if the input takes many lines and the parser just gives up, telling: "The input contains a syntax error." Finding the exact location though, is almost impossible. Assume for instance a parser which detects an error at the first symbol in the input that results in an incorrect prefix of a sentence of the language (this is called the correct-prefix property). It is obvious that this is not necessarily the real location of the error. It is possible that an error occured at an earlier stage, but that the prefix still was syntactically correct. However, for many applications of a parser it suffices to recognize at which line or position the parser detected an error. At least a good hint to find the actual position is given. Now, this error detection capability may be satisfactory for small parsers, like a command line parser, for most others it is not. When a complete computer program is parsed, it is a nuisance if the parser stops after every single error. A list with as many syntax errors detected in the input as possible is more desirable. Therefore, a parser which pretends to have good error handling capabilities needs the concept of error recovery.

2.2 Error Recovery

Like stated in section 2.1, it is desirable for a parser to continue processing after an error has been found. In general, this can not be achieved by throwing the erroneous symbol away and just continue to parse. The parser will have to reorganize its internal state in order to be able to restart the parsing process. This adaption of the internal state is called error recovery. It allows the parser to continue after the occurence of a syntax error and to try to detect more errors in the input. One of the difficulties is that after an improper adaption of the internal state all kinds of spurious error messages can be encountered. Obviously this should be reduced to a minimum. By trying to 'repair' the error we are getting close to error correction.

2.3 Error Correction

Instead of just recovering from a syntax error, error correction attempts to get much closer to the ideal situation. However, it would be an utopia to have an algorithm that can correct all errors in the input. We can not expect the computer to write our programs. Nevertheless, error correction does transform the input into syntactically correct code. Therefore the term 'error repair' might be more suitable since the errors are not really corrected. The transformation of the input is usually done by inserting, deleting, or altering symbols. The main benefit of these repairs is that the parser will be able to continue. All semantic actions associated with the grammer rules will be properly executed. As a drawback, this error correction capability requires more heuristics than error recovery does. Both in contrast to error detection, which basically comes for free. The next chapter discusses several different error handling techniques, some of which only recover from an error and some of which actually correct it.

Chapter 3

Error Correction and Recovery Techniques

The issue of error correction and error recovery has received much attention since the existence of parsers and many possible solutions have been presented by many authors. Each and every solution has its own benefits and drawbacks. Some methods are essentially ad-hoc and are mainly hand-coded, therefore inflexible. Others are, in certain situations, forced to skip large portions of the input. When there are several possible repairs, an arbitrary and sometimes unreasonable choice is made. The more sophisticated routines can suffer from non-linear space or time bounds and are considered impractical.

The following sections discuss some of the different approaches to error correction and error recovery. The first section presents the notation and definitions used.

3.1 Notation and Definitions

The following notation and definitions are taken from [vdS88], [FMQ80] and [Röh80].

Input to a parser is a sequence of symbols taken from a set of terminal symbols V_t, called the alphabet. The following notation applies to this set:

- V_t^* is the set of all finite sequences of elements of V_t.

- ε is the empty sequence (also in V_t^*).
- $V_t^+ = V_t^* - \{\varepsilon\}$.

The syntax of a language L defines which symbol sequences are elements of L. To describe the sets of sequences and their structure which form L, a context-free grammar is used. A grammar G is a 4-tuple (V_n, V_t, P, S), where:

- V_n is the nonterminal vocabulary of G.
- V_t is the terminal vocabulary of G.
- $P \subset V_n \times V^*$ is the set of production rules of G, where $V = V_n \cup V_t$.
- S is the start symbol of G.

The parsing process is a rewriting process on elements of V^*. The process begins with the start symbol S. By repeatedly using the production rules the start symbol is rewritten until a string of terminals remains. A production rule (A, α) denotes that the nonterminal A can be replaced by the sequence α, which is usually written as $A \rightarrow \alpha$. Furthermore, string catenation is denoted by **cat** and the following relations exist:

- $\forall x, y \in V^* : x \Rightarrow y$ iff $(\exists A \in V_n; \alpha, \beta, \gamma \in V^* | x = \beta A \gamma \wedge y = \beta \alpha \gamma \wedge A \rightarrow \alpha \in P)$
- The relation $\Rightarrow^+$ is the transitive closure of $\Rightarrow$
- The relation $\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$

Some parsing techniques and error recovery algorithms require an augmented grammar notation in order to be well-defined. Let $G = (V_n, V_t, P, S)$, then the augmented grammar is defined as:

$$G' = (V_n \cup \{S'\},\ V_t \cup \{\perp\},\ P \cup \{S' \rightarrow S \perp\},\ S') = (\hat{V}_n,\ \hat{V}_t,\ \hat{P},\ S')$$

where $\{S', \perp\} \cap (V_n \cup V_t) = \emptyset$. The essential property of this grammar is that all input strings will be terminated by the endmarker symbol $\perp$.

If $z = xy \in V_t^*$ then x is called a prefix of z and y is called a suffix of z. If $z \in G$ then x is also called a prefix of G. If $z \notin G$ then the parsing error in z is the ordered pair (u, v) where $z = uv$ and u is the longest common prefix of z and G. The term parsing error derives from the fact that no parser reading z from left to right can detect an error inside uv before scanning v. If (u, v) is a parsing error then there exists at least one word $w \in V_t^*$ with $uw \in G$. Such a word w is called a continuation of u (with respect to G).

The term error detection point is used to indicate the point where the parser detects the error. The term error symbol indicates the symbol on which the error is detected.

3.2 Least-error Correction Method

The least-error correction method is a form of global error handling. These methods use a global context, which implies that they consider the entire input. The basic idea of this algorithm is to find a syntactically correct input with as few corrections as possible. Every single edit operation is considered to be a correction. This can be either a symbol insertion, a symbol deletion or a symbol change. A maximum to the number of corrections can be found by considering the shortest sentence that can be generated by the language. Figure 3.1 shows a shortest sentence with lenght m. If the input (length n) is longer than the shortest sentence, the first m symbols, part x of the input, are changed to the symbols of the shortest sentence. The last $n - m$ symbols, part y, are simply deleted. This procedure will need n corrections at most (some symbols in m and x may coincide). If the input is shorter than the shortest sentence, $m - n$ symbols will have to be inserted at the end of the input, after altering the first n. This time m corrections are needed. Therefore the maximum number of operations needed to correct the input will be $max\{m, n\}$.

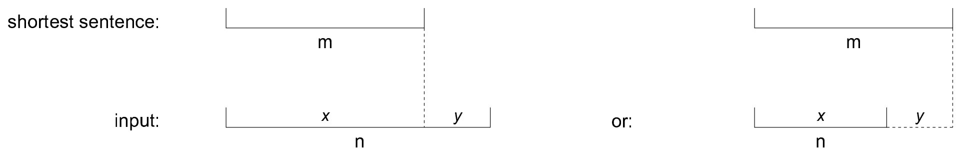

Figure 3.1: Shortest sentence

When given erroneous input, the least-error correction method will apply all grammar rules to find correct alternatives. Since a maximum is already known, possibilities which require more corrections than this maximum are discarded. In case several alternatives are found for correcting the input with the same number of corrections, the parser writer decides which one will be used. A nice example of the working of this method can be found in [GJ90].

It is clear that the major disadvantage of this method, without any optimization, is that all possible corrections have to be computed before it is known which one is the smallest. In general, global error handling techniques like the least-error correction method are very effective, but always very time consuming.

3.3 Ad Hoc Error Recovery Methods

The typical property of ad hoc error recovery methods is that they can not be automatically generated from the given grammar. They are mainly hand coded and their effectieness depends on how good the parser writer can anticipate possible syntax errors. Three ad hoc methods are discussed: error productions, empty table slots and error tokens.

Error productions are additional grammar rules. In fact, anticipated syntax errors actually become part of the language. One could argue that therefore they cease to be a syntax error. Whenever a certain anticipated error occurs, the corresponding error production is used. This production rule will trigger a specific semantic action which will, for instance, generate an appropriate error message. This is in fact the only advantage compared to other methods: adequate error messages for frequent errors. The disadvantages are clear: only anticipated errors are handled and the rules added can introduce conflicts in the modified grammar.

If a parser uses parse tables, empty table slots are a way to provide error handling. Whenever such an empty slot is consulted, an error is detected. The empty slot will refer to a certain error handling routine which will have to be provided by the parser writer. Only by careful design of these error handling routines, good results can be achieved. This requires quite some effort and therefore this approach is not considered to be practical.

A third way to provide ad hoc error handling is by using error tokens. These tokens are special tokens that are inserted before the error detection point. After detection of an error, the parser will pop states from the parse stack until this token is valid. From this point on, it will skip symbols from the input until a certain designated symbol is found, for instance a semicolon or a newline symbol. Although this method can be quite effective, it can throw away large portions of the input.

3.4 Panic Mode

Panic mode is a very simple form of a local error handling technique or so-called acceptable-set recovery technique. When a parser, equipped with this technique, detects an error, it will calculate a certain set from the parser state, called the acceptable set. Symbols of the input are skipped until a member of the acceptable set is found. Finally, the parser state is adapted to make the found symbol acceptable. The panic mode variant is simple because the acceptable set is not calculated at all, but determined by the parser writer. This set normally consists of symbols which denote the end of a syntactic construct, like in several languages the semicolon. When an error is detected, symbols of the input are skipped until, for instance, a semicolon is found. Then, the parser must be brought into a state that makes the semicolon acceptable. This adaption of the parser state depends on the type of parser being used. This error recovery method is quite adequate and easy to implement. The major disadvantage is that many errors will not be detected, since large parts of the input can be skipped. This is of course dictated by the effectiveness of the given acceptable set.

3.5 Insertion-Only

Another variant of local error handling is insertion-only error correction. This method is also known as the FMQ error correction method, because it has been developed by Fischer, Milton and Quiring [FMQ80]. The acceptable set in this method is the entire set of terminal symbols. The erroneous symbol is directly found in this set and no input is skipped. Therefore this method only tries to reconstruct the internal state of the parser.

Since only insertions are used, the question arises if every input can be corrected like this. Unfortunately this is not the case, although the class of grammars which can be corrected is a large one. Members of this class are called insert-correctable. This means that for every prefix x of a sentence and every symbol a in the language there is a continuation of x that includes a, so an insertion can always be found. Next to this constraint it is necessary for the parser to have the immediate error detection property. Both prerequisites can be determined, like demonstrated in [FMQ80].

The basic idea of this method is very simple. Suppose that the parser detects an error on input symbol a. The error corrector will now determine the cheapest insertion containing a. The price of an insertion is determined by the cost of each inserted terminal symbol, which is given by the parser writer. To compute this insertion two tables are used. First, the S-table, containing the cheapest derivation of each symbol. For a terminal this is of course the terminal itself. Second, the E-table, which contains the cheapest insertion for each symbol/terminal combination (X, a). This is either an insertion y, such that $X \Rightarrow^* ya...$, or an indication that there exists no valid insertion. How the actual computation of the insertion based upon these two tables takes place, is discussed in section 4.1.

The main disadvantage of this method is that it behaves poorly on errors which are better corrected by a deletion or an alteration. On the other hand there are several advantages:

- it can be automatically generated from the grammar and cost vector
- it ensures that any input string can be parsed
- the actual choice of insertions can be fine-tuned by adjusting the cost-vector
- it is relatively easy to implement

The last item is the main reason why this method has been chosen for GP.

3.6 Backtrack-free Error Correction using Continuations

This is the last local error handling method which is discussed. This time the acceptable-set is found by deriving a continuation of a certain correct prefix u. The term backtrack-free indicates that the prefix u of the parsing error (u, v) remains unchanged. Three problems are encountered which have to be solved for this method to work properly: the consistency, the detection and the continuability problem. Details of these problems and their solution are discussed in [Röh80].

Assume that the parser encounteres the error (u, v) in a word z with respect to the language L. The error correction algorithm will first determine a valid continuation w of u. Let w' be a prefix of w and let v'' be a postfix of v. Then $z' = uw'v''$ is called a backtrack-free correction of (u, v), as shown in figure 3.2. The best correction is the one that preserves the most of v, which implies that w' should be as short as possible, or v'' should be as long as possible. The corrected word $z' = uw'v''$ may contain further parsing errors within v''. This is allowed if any such error may be written as $(uw'x, y)$ where $xy = v''$, but $x \neq \emptyset$. The correction process can then be repeated until the final word z' does belong to L. An example of this method can be found in [Röh80].

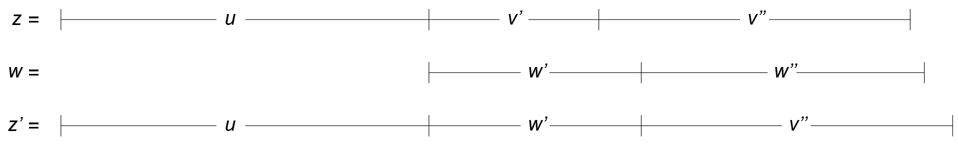

Figure 3.2: Backtrack-free correction of a parsing error (u, v) within a word z

This method has similar advantages as the insertion-only error correction method, except for the possibility to fine-tune the result with a cost-vector and it is less easy to implement.

Chapter 4

Insertion-Only Error Correction

Section 3.5 shortly discussed the insertion-only error correction algorithm. Since this error corrector has been implemented for GP, this chapter will dive into the details. Next to the theoretical discussion, some optimizations to the original design are studied.

4.1 The Insertion-Only Algorithm

The general idea of this algorithm is clear: being able to correct and parse any input string by insertions only. In order to achieve this, the method is limited to insert-correctable LL(1) grammars. A context free grammar is said to be insert-correctable iff for all $x \in V_t^*$ and $a \in \hat{V_t}$ such that $S' \Rightarrow^+ x...$ and $S' \not\Rightarrow^+ xa...$, there exists $y \in V_t^+$ such that $S' \Rightarrow^+ xya....$ Tthese are the grammars for which it is always possible to repair an error by a insertion of a suitable terminal string. It is shown in [FMQ80] how the grammar can be tested for this property. A second prerequisite is the immediate error detection (IED) property of the parsing algorithm. This requires the parser to directly detect the error when an erroneous symbol is encountered. Information about the IED can be found in [FTM79]. Here, it is assumed that the parser guarantees the IED property and that the grammar is insert-correctable.

Since it is possible to have an insertion at the end of an input string, it is necessary to use an augmented grammar (as defined in section 3.1).

The basic error correction algorithm is defined like this: given as input a string $xa...$ ($x \in V_t^*$, $a \in \hat{V}_t$) such that $S' \Rightarrow^+ x...$ but $S' \not\Rightarrow^+ xa...$, the correction algorithm will find a least-cost string $y \in V_t^+$ such that $S' \Rightarrow^+ xya...$. Figure 4.1 shows the corresponding state of the parser: the current prediction is $X_n...X_1$, x has already been parsed and a is the detected error.

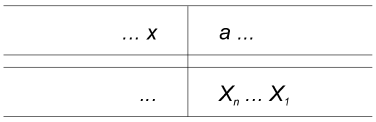

Figure 4.1: State of the parser after detecting an error

Since the grammar is insert-correctable, $X_n...X_1$ must derive a terminal string containing a. To calculate a least-cost string, the error correcting algorithm requires two auxiliary tables, S and E. Both tables rely on the a priori given terminal insertion-cost vector $C(x)$, $x \in V_t$. The first table, $S(X)$ with $X \in V$, identifies the cheapest derivation from X. For a terminal this is of course the terminal itself. The other table, $E(A, a)$ with $A \in V$ and $a \in \hat{V}_t$, gives the cheapest insertion starting from A which makes a acceptable. Not every symbol/terminal combination will have a valid insertion, which the E-table will explicitly indicate. Both tables are precalculated for the desired grammar by the parser generator.

The computation of the insertion-string y starts with consulting $E(X_n, a)$. Two results are possible:

- $E(X_n, a)$ contains an insertion y_n. A cheapest insertion is directly found and the parsing process can be restored.
- $E(X_n, a)$ does not contain an insertion. Now the cheapest derivation $S(X_n)$ is used as the first part of the insertion and the process is continued with X_{n-1}. In the end this will result in an insertion $y_n...y_{i+1}$ retrieved from the S-table and an insertion y_i from the E-table, shown in figure 4.2.

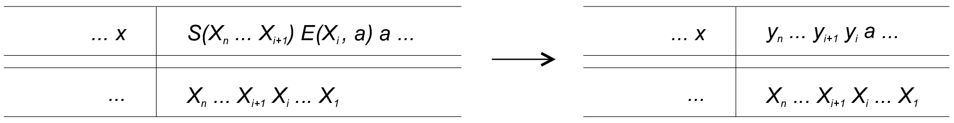
... x
S(X_n ... X_i+1) E(X_i, a) a ...
...
X_n ... X_i+1 X_i ... X_1
... x
y_n ... y_i+1 y_i a ...
...
X_n ... X_i+1 X_i ... X_1

Figure 4.2: State of the parser after insertion of the derived string

4.2 Optimizations

Two optimizations are proposed by Anderson and Backhouse [AB82]. The first one is the factorisation lemma, which is based on the observation that it is sufficient to compute the first symbol of the insertion. If an error symbol a is detected after having read the prefix u and $w = w_1w_2...w_n$ is the cheapest insertion, then $w_2..w_n$ is a cheapest insertion for the error a after having read uw_1. The main disadvantage is that the error corrector now has to be started n times, therefor increasing the processing time. The benefit of only computing the first symbol is the reduction in size of both the S and E-table, since elements of length greater then one are discarded. Nevertheless, this is not the reason why GP makes use of the factorisation lemma. Since GP generates recursive descent parsers, the prediction is not explicitly available, only the first part X_n is. In this case, X_n is used to compute y_n by consulting the E-table. If no valid insertion is found, $S(X_n)$ will be inserted and the process is repeated (X_{n-1} now explicitly available), like figure 4.3 shows.

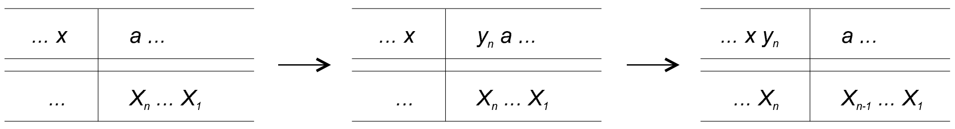

Figure 4.3: String insertion by using the factorisation lemma

The second observed optimization considers the immediate error detection property (IED). The parser has to possess this property in order for the error corrector to work properly. However, it is clear that the cheapest insertion for a correct input symbol wil be no insertion at all. So, if the error corrector is placed in between of the lexer and the parser, the latter will be garuanteed to recieve correct input. Therefore it is not necessary for the parser to have the IED property. But since GP already has this property and this approach reduces the parsing speed, this optimization is not used.

Finally, during the implementation of the error corrector, it showed that a great number of the elements of the E-table were identical. The following table illustrates this effect:

insertion length	number of insertions	unique insertions
0	1739	1
1	3318	36
2	4448	96
3	3100	94
4	800	69
5	282	19
6	88	11
7	99	4
8	0	0
9	3	1

This table has been generated with the parser generator and the TOM grammar as input. This grammar consists of 89 terminals, 59 nonterminals and 287 production rules. It clearly shows that there are only few unique insertions, compared to the total number of valid insertions. Take for example the cheapest insertions of length two. In total there are 4448 symbol/terminal combinations which have a valid insertion of this length. But only 96 unique insertions exist among those 4448. Therefor, in order to reduce the size of the E-table, only the unique insertions are stored, every one of which is referenced by many different symbol/terminal combinations.

Chapter 5

Conclusions

The goal of this report was to describe the investigation of the possibilities of adding error handling facilities to parsers generated by GP. First of all, it is shown what can be expected from both error recovery and error correction, by looking at the entire process of error handling. With this knowledge, several well known methods are studied. Based upon several selection criteria the error correction algorithm using only insertions has been selected. Four criteria were used:

- The speed at which error recovery takes place:

  The insertion only error corrector uses lookup tables to find the cheapest insertion. All the hard work of preparing the tables is done in advance by the parser generator.

- The speed of error-free operation of the parser:

  The parsing speed is not altered in an error-free situation, since the error correction algorithm is not started at all.

- How effective the errors are corrected:

  This algorithm guarantees a syntactically correct input, regardless what kind of error is encountered.

- The effort needed to implement the algorithm:

  Both the computation of the tables and the error corrector itself are easily implemented in a parser generator. Only the representation of the production rules in GP was adjusted slightly.

Although the implementation has not been finished yet, it can be concluded that the functionallity of GP has been greatly extended. It is only a matter of some implementational effort to finish the work on the error corrector. Both the S and E-table are completed and show exactly the results as expected.

Bibliography

[AB82] S.O. Anderson and R.C. Backhouse. An alternative implementation of an insertion-only recovery technique. *Acta Informatica*, 18:289–298, 1982.

[AU72] A.V. Aho and J.D. Ullman. *The theory of parsing, translation and compiling*, volume 1, chapter 5.1. Prentice-Hall, 1972.

[AU73] A.V. Aho and J.D. Ullman. *The theory of parsing, translation and compiling*, volume 2, pages 674–675. Prentice-Hall, 1973.

[Com98] Apple Computer, editor. *Object-Oriented Programming and the Objective-C Language*. Apple Computer, Inc., 1998.

[FMQ80] C.N. Fischer, D.R. Milton, and S.B. Quiring. Efficient LL(1) error correction and recovery using only insertions. *Acta Informatica*, 13:141–154, 1980.

[FTM79] C.N. Fischer, K.C. Tai, and D.R. Milton. Immediate error detection in strong LL(1) parsers. *Information Processing Letters*, 8(5):261–266, 1979.

[Ghe75] C. Ghezzi. LL(1) grammars supporting efficient error handling. *Information Processing Letters*, 3(6):174–176, 1975.

[GJ90] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques – A Practical Guide*, chapter 10, pages 229–248. Vrije Universiteit Amsterdam, 1990.

[Jac85] Ceriel J.H. Jacobs. LLgen, an extended LL(1) parser generator. Technical report, Dept. of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1985.

[KD95] Helmut Kopka and Patrick W. Daly. *A Guide to LaTeX 2ε*. Addison–Wesley, second edition, 1995.

[Röh80] Johannes Röhrich. Methods for the automatic construction of error correcting parsers. *Acta Informatica*, 13:115–139, 1980.

[Sch98a] Pieter J. Schoenmakers. *TOM Language Reference Manual*, 1998. Draft.

[Sch98b] Pieter J. Schoenmakers. *The TOM Programming Language*, 1998. Draft.

[vdS88] J.L.A. van de Snepscheut. *Compilers 1, Syllabus bij het college (2K610).* Technische Universiteit Eindhoven, 1988.