Syntax Error Handling in Scannerless Generalized LR Parsers

Ron Valkering

Master's thesis

August 2007

17 August 2007

One year master program Software Engineering

Thesis supervisor: Jurgen Vinju

Internship supervisor: Rob Economopoulos

Company or institute: CWI

Availability: Public Domain

Universiteit van Amsterdam

Acknowledgement

This thesis is the outcome of a project I conducted at the CWI in Amsterdam. I would like to thank all people working at SEN1 for providing a pleasant environment to work. Special thanks have to go to Rob Economopoulos and Jurgen Vinju for getting me started and for their valuable suggestions. Especially Jurgens suggestion to start with implementing a visualisation tool for GSS proved to be extremely useful. Both for getting insight in the workings of SGLR and for debugging the new algorithms.

Furthermore I want to thank everybody who reviewed the draft versions of this thesis, Jurgen and Rob again, and Paul Klint and my fellow students Bas Basten and Jan Derriks. Their remarks and suggestions were very valuable.

I also want to thank the Open Universiteit Nederland, as an institution and in particularly its instructors/course-writers. Without them, it would never have been within my grasp to conduct a project like this, not at my age and with my level of pre-education.

Contents

Abstract

This thesis is about a master's project as part of the one year master study 'Software-engineering'. This project is about methods for improving the quality of reporting and handling of syntax errors that are produced by a scannerless generalized left-to-right rightmost (SGLR) parser, and is done at Centrum voor Wiskunde en Informatica (CWI) in Amsterdam.

SGLR is a parsing algorithm developed as part of Generic Language Technology Project at SEN1, one of the themes at CWI. SGLR is based on the GLR algorithm developed by Tomita.

SGLR parsers are able to recognize arbitrary context-free grammars, which enables grammar modularization. Because SGLR does not use a separate scanner, also layout and comments are incorporated into the parse tree. This makes SGLR a powerful tool for code analysis and code transformations. A drawback is the way SGLR handles syntax errors.

When a syntax error is detected, the current implementation of SGLR halts the parsing process and reports back to the user the point of error detection only. The text at the point of error detection is not necessarily the text that has to be changed to repair the error.

This thesis describes three kinds of information that could be reported to the user, and how they could be derived from the parse process when an error is detected. These are:

–The structure of the already parsed part of the input in the form of a partial parse tree.

–A listing of expected symbols; those tokens or token sequences that are acceptable instead of the erroneous text.

–The current parser state which could be translated into language dependent informative messages.

Also two ways of recovering from an error condition are described. These are non-correcting recovery methods that enable SGLR to always return a parse tree that can be unparsed into the original input sentence.

–A method that halts parsing but incorporates the remainder of the input into the parse tree.

–A method that resumes parsing by means of substring parsing.

During the course of the project the described approaches have been implemented and incorporated in the implementation of SGLR as used by the Meta-Environment, some fully, some more or less prototyped.

Chapter 1

Introduction

The Generic Language Technology Project at CWI (Centrum voor Wiskunde en Informatica / Centre for Mathematics and Computer science) accumulates to the Meta-Environment[METAWS], an interactive environment for language development, source code analysis, and source code transformation based on ASF+SDF. SDF (Syntax Definition Formalism[HEERING1989]) is the grammar defining part. Texts obeying these grammars can be analysed and transformed with rules specified in ASF (Algebraic Specification Formalism).

The Meta-Environment is constructed of a series of tools. It uses the ToolBus [BERGSTRA1994], a generic coordination architecture, to operate these tools. The workings of the ToolBus are controlled by means of ToolBus scripts. Alongside the ToolBus interface for use with the ToolBus, most tools also provide for a command line interface.

For ASF+SDF a parser generator has been developed. Parsers are used to convert a sentence of a language into some structure called a parse tree. The parsers generated by the Meta-Environment use a very generic parse algorithm called generalized left-to-right rightmost (GLR). This is one of a whole family of left-to-right rightmost (LR) parsing algorithms.

The GLR algorithm can be used to parse sentences defined by any context-free grammar. A grammar defines the structures used to build the parse tree and limits the number of acceptable trees. Because the result of combining two context-free grammars is also a context-free grammar, the use of this algorithm enables the modularization and mixing of grammars. This is contrary to other LR algorithms which can be used for only a certain subset of context-free grammars. A combination of grammars from such a subset will often not be a member of that subset.

Furthermore these parsers are scannerless, which means they take as input raw sentences of the language character by character, instead of a stream of tokens, which are produced by a lexical syntax interpreting scanner.

Usually a scanner removes parts of the sentence that have no semantic meaning like layout and comments. A scannerless parser on the other hand does not remove any of those. The resulting parse tree therefore can be converted back, or unparsed, into an exact copy of the input sentence.

Therefore the algorithm used by the generated parsers is called SGLR, which is

shorthand for scannerless generalized left-to-right rightmost.

Any parsing algorithm needs some way of determining how a certain token should be interpreted. Mostly this can be determined by the remainder of the input sentence. If there exists a sentence that is acceptable for some grammar, but has more than one possible interpretation, then both the sentence as the grammar is called ambiguous.

For programming languages, often grammar independent disambiguation rules are used to disambiguate otherwise ambiguous grammars.

Many parsing algorithms do their job with the aid of a parse table and a parse stack. The parse table is used to determine the actions to be taken, and the parse stack is used to store intermediate results.

These algorithms only need a different parse table to be able to parse texts for different grammars, and parser generation then boils down to parse table generation. SGLR is one of those algorithms.

When a grammar is ambiguous, then the parse table will contain non deterministic entries. The usual methods to deal with non-deterministic parse table entries limit the class of accepted grammars to a sub class of the context-free grammars. For GLR there is not such limitation.

GLR parsers handle non-deterministic parse table entries by means of splitting up the parse stack, creating a branch for each possibility. Such a branch is terminated again when a token is read which cannot be a legitimate continuation for that branch. Merging of branches where possible, prevents that memory usage will grow exponential.

When the stack has no branches left before the end of the sentence is reached, or when the parse process does not result in an acceptable parse tree, the sentence is rejected. Such a rejection is called a syntax error.

The text at the place of error detection is not acceptable for the specific text that came before it. This does not mean it will be unacceptable for every text that comes before it. Therefore the error might be repaired by changes in the text at a certain number of points left of the point of error detection.

Syntax errors should be reported in such a way that the report aids the user as much as possible in finding the best repair for the error. The best repair is such a change to the sentence, that the repaired sentence can be parsed into the structure that the user intended in the first place.

Error reports should include the nature of the error, the position of the error, and possible corrections. If an error is detected before the end of the sentence is reached, then normal parsing cannot commence. Therefore a strategy should exist for how to recover from an error situation.

The specification of SGLR says nothing about error handling or error reporting. This is left to the implementation. The current implementation provides for a bare minimum. It halts operations and reports the location where it was forced to do so. This minimal error management decreases the usability of the generated parsers and the Meta-Environment as a whole. [BRAVENBOER2006] mentions this too as one of the points on which SGLR might be improved.

This project aims to improve the quality of both reporting and recovering from syntax errors. As a starting point it tries to do this in a grammar independent way and without alterations to the grammars. It focusses on what information can be retrieved from the parse stack, at the moment of error detection.

1.1 Improved error handling for SGLR

SGLR differs from other LR parsing algorithms in that it uses a parse table that is not deterministic. At any moment during parsing multiple actions might be possible. Also SGLR uses tokens of one character each. Because of these differences error handling routines developed for LR parsers are not applicable for SGLR just like that. And because there exists no separate scanner, errors in the lexical syntax have to be handled by the parser also.

This project investigated three possible pieces of information that can be extracted from the parse stack and that can be used to create an error report that is more informative then the reports currently issued:

1. The structure of the consumed part of the input text.
2. The expected symbols, the tokens or token sequences that are acceptable instead of the erroneous text.
3. Language specific error messages, extracted from the parser state that the error is detected at.

Also two non-correcting methods of recovering from an error situation have been looked in to:

1. Halting after an error.
2. Continuing after an error as a substring parser.

All these approaches have been integrated into the current implementation of SGLR.

No existing literature has been found about error management for SGLR or GLR parsing algorithms. For the more widely used LR(k) algorithm, on the other hand lots of research in this field has been done. The results are evaluated for the conditions found in that literature.

1.2 Structure of this thesis

The background for this study is described in chapter 2. That chapter gives an overview of parsing and the major algorithms for that, and it describes the SGLR algorithm fully. The second section of chapter 2 describes how error handling (error-detection, -reporting, -recovering) is done in LR parsers.

The chapters 3 to 7 discuss the techniques developed during this project. Each chapter describes a single technique in depth. Each chapter is set up of four sections. The first section discusses the background for the technique, why it could be useful, how it is used in other situations, what is the current situation in SGLR, etc. The second section gives a full description of the algorithms used to implement the technique in SGLR. The third section of each chapter evaluates the results of the technique. Section 2.2.2 lists requirements derived from literature that are used to aid this evaluation. Each chapter wraps up with a conclusion.

Chapter 8 approaches the five techniques as a whole. It gives some examples of how the results are visualized in the Meta-Environment. It also provides measurements about the implementation. Overall conclusions are drawn and suggestions for future work are given in chapter 9.

Chapter 2

Background and related work

This chapter provides background information about parsers. The first section describes briefly the parsing process in general and the SGLR algorithm in particular. The second section describes literature about error handling for parsers and compilers.

2.1 Parsing

A parser processes an input text into some structure. See fig 2.1. This structure is referred to as the parse tree and is formally described by a (usually context-free) grammar. A grammar is defined by a four tuple $\{N, \Sigma, P, S\}$ where N is a set of non-terminals, Σ is a set of terminals disjoint from N, P is a set of production rules, and S is an element of N that is the start symbol. A grammar is context-free when the production rules are of the form $V \rightarrow w$ where $V \in N$ and w is any combination of elements of $N \cup \Sigma$. Non-terminals of the grammar form the nodes of the tree and the leaves represent the terminals of the grammar. Many examples in this thesis use the following grammar, G_{sum}, which is adopted from [FTAT]:

$N = \{S, N, D\}$

$\Sigma = \{+, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$P = \{S \rightarrow N,\ S \rightarrow N + S,\ N \rightarrow D,\ N \rightarrow DN,\ D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\}$

$S = S$

Usually parsing is performed in two steps. First a lexical analysis converts a sequence of characters into a sequences of tokens that correspond with the terminals of the grammar. This analysis is performed by the scanner or lexer. The scanner is usually based on a finite state machine which can recognize regular expressions. The scanner also removes parts that have no semantic meaning like layout and comments.

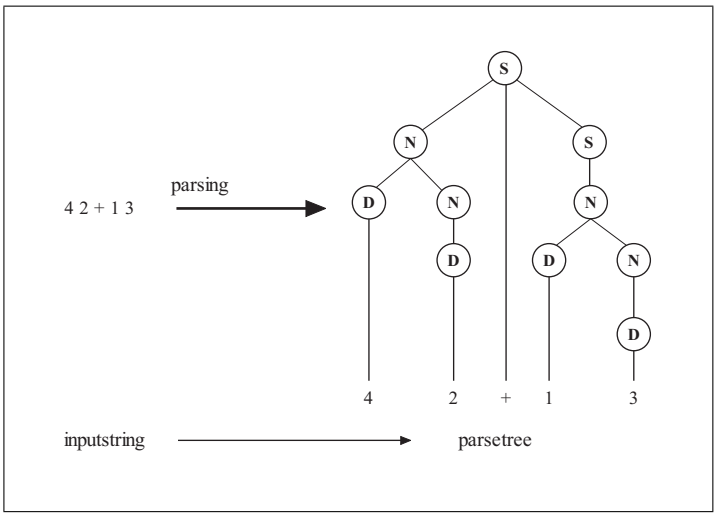

Figure 2.1: Parse tree for the sentence "42+13"

The sequence produced by the scanner forms the input string for the parser. For parsing there are two main strategies: top-down, which builds the parse tree starting with the root, and bottom-up which builds the parse tree starting with the leaves.

The most widely used technique for parsing programming languages is a bottom-up technique called LR parsing. L(eft-to-right) for reading the input string from left to right and R(ightmost) for making rightmost derivations from the grammar in reverse.

LR parsers use a stack upon which shift and reduce actions are taken. Shift actions put tokens from the input string, which are terminals from the grammar, on the stack. Reduce actions pop one or more items from the stack and replace them with a non-terminal from the grammar. (table 2.1)

What reduce and or shift actions have to be performed for a given parser state is determined by the lookahead. The lookahead consists of the tokens that have to be read next.

Parsers that use a lookahead of k tokens are called LR(k) parsers. The grammars, they accept are called LR(k) grammars. These grammars are deterministic for a lookahead k.

An algorithm called simple left right (SLR)[DEREMER1971] is used to generate a table, the parse table, from which shift and reduce actions can be looked up. The existence, in the parse table, of multiple entries for a given parser state and lookahead k, indicate that the grammar is not LR(k). Such a grammar is non-deterministic for the given k.

2.1.1 SGLR

The Meta-Environment uses a parsing algorithm called scannerless generalized left-to-right rightmost (SGLR).[VISSER1997] It is based on the GLR algorithm of [REKERS1992], which itself is a slightly improved implementation of the

input string	action	stack
42+13	shift	4
42+13	reduce1	D
42+13	shift	2D
42+13	reduce1	DD
42+13	reduce1	ND
42+13	reduce2	N
42+13	shift	+N
42+13	shift	1+N
42+13	reduce1	D+N
42+13	shift	3D+N
42+13_	reduce1	DD+N
42+13_	reduce1	ND+N
42+13_	reduce2	N+N
42+13_	reduce1	S+N
42+13_	reduce3	S

Table 2.1: LR stack for parsing the sentence "42+13"

GLR algorithm originally proposed by [TOMITA1987]. Tomita developed this algorithm for parsing natural languages.

Advantages of GLR are that it can be used for arbitrary context-free grammars. The set of context-free grammars is closed under union, which means that combining two or more context-free grammars results in another context-free grammar. This enables modularization of the grammar, also two or more grammars can be mixed together, which enables embedded languages and cross language transformations. All this is contrary to the more widely used LL(k) or LR(k) algorithms whose sets of accepted grammars are not closed under union. LL(k) and LR(k) grammars are deterministic for a specific lookahead k, combined grammars with the same k are not necessarily deterministic for that k.

In GLR, which is based on LR, non deterministic parse table entries are handled by starting a new parallel parse stack each time more than one continuation is possible. Such stack is terminated when no more actions are possible given the next token from the input string. This way, while using a lookahead of only one, the real lookahead is as long as the remainder of the input string. Thus GLR provides for a dynamic lookahead.

A LR(k) parser uses a stack to hold information during the parse process. In SGLR this leads to an extra stack each time a non deterministic parse table entry is encountered. Instead of many separate stacks, all stacks are handled with a memory efficient graph structured stack (GSS) as proposed by [TOMITA1987]. GSS consists of nodes, which hold parser states, and edges, which hold parse trees. These trees are subtrees from the resulting parse tree.(figure 2.2)

Each token from the input string provides for a level in GSS. Each level in GSS contains at much one node with the same parser state, multiple stacks may share such a node. GSS forms a stack with one common bottom, and several tops.

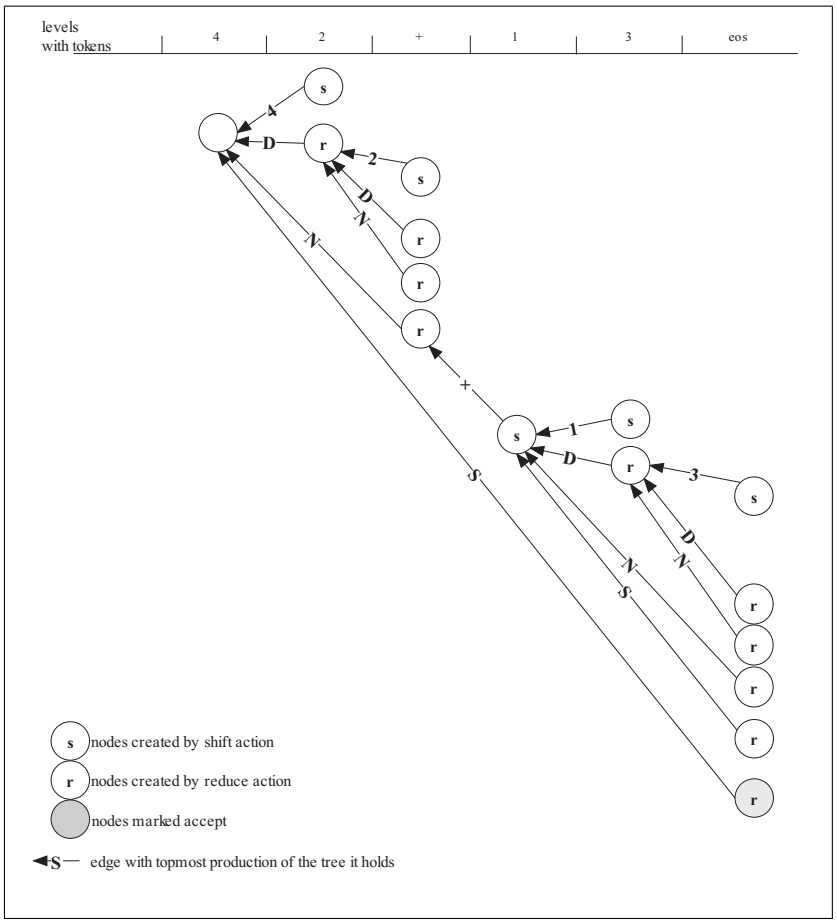

Figure 2.2: GSS while parsing "42+13". This example grammar is fully deterministic, therefore this figure shows only one parse stack.

GLR parses LR(k) grammars as efficient as the usual algorithms, and uses the same SLR[DEREMER1971] parse table generation algorithm.

The dynamic lookahead of GLR makes it possible to use the individual characters of the text as input, instead of tokens produced by a separate lexical scanner. A scanner usually uses a separate regular grammar for recognizing lexical syntax. Scannerless SGLR makes it possible to use the same context-free formalism to define the lexical syntax.

To be able to separate non-terminals, introduction of auxiliary context-free layout productions are necessary. For simplicity these and other productions added by the normalization process [VISSER1997] are not shown in the examples.

Furthermore by consuming the input string character by character all layout and comments are also stored in the parse tree. Unparsing therefore returns an exact copy of the original text.

SGLR produces a parse forest rather then a parse tree. Each stack that is not terminated provides for a tree in this forest. To this parse forest a couple of disambiguation filters could be applied. Among others, 'longest match' and 'prefer keywords' disambiguation rules are implemented using these filters.

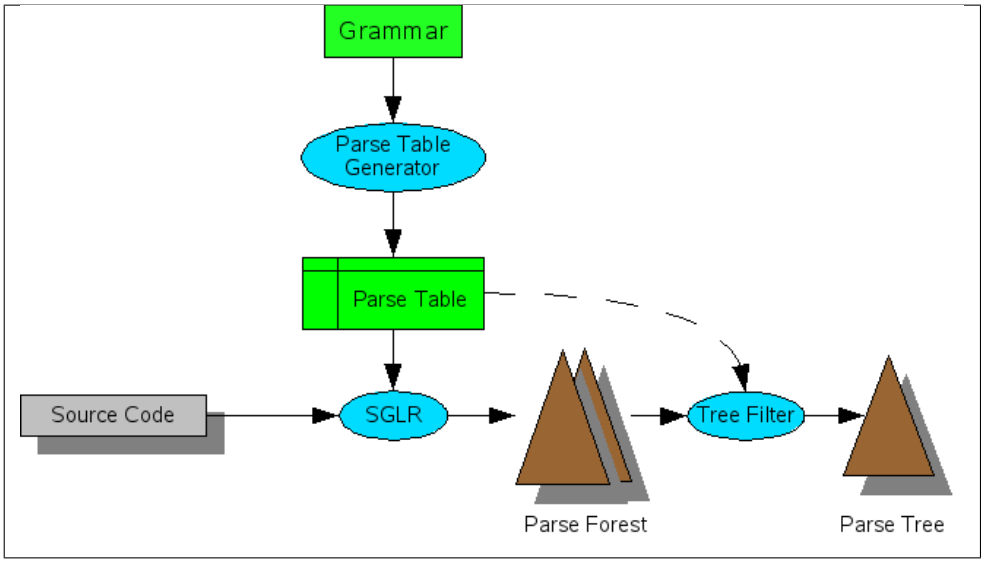

Figure 2.3: SGLR parse architecture[METAWS]

To be able to understand the error handling techniques proposed in this thesis, it is necessary to have a reasonable understanding of the GLR parsing algorithm and the vocabulary that comes with it. Next subsection will give an in depth description of this algorithm. Words that are particularly for the vocabulary of the (GLR) parsing algorithm are printed in a bold typeface near the place where they are explained.

2.1.1.1 GLR parsing algorithm

GLR parsing progresses in cycles, the **parse cycles**. Each parse cycle consumes one token from the input string. In SGLR a **token** consists of exactly one character. During parsing, the **input string** can be divided in two parts: the consumed part or **parsed prefix** and the not consumed part or **not-parsed suffix**. The token being consumed a.k.a. the **current token** is the first token of the not-parsed suffix.

Another term in connection with the input sentence is the **lookahead**. The lookahead usually has a specified length. If the lookahead has length one, then the lookahead is the same as the current token. Longer lookaheads consists of the current token extended with as many tokens from the not consumed part of the input string as needed.

There exists one special token, the end of string token (**eos**), which will be the current token when the last token of the input string is consumed.

Actions taken during each parse cycle are determined by the parse table. The **parse table** consists of two two-dimensional tables. The action table defines actions for each parser state dependent on the current token. There may be multiple actions for each **parser state** token combination. This is where SGLR differs from LR(k) parsing algorithms. The goto table defines new parser states for each parser state dependent on the action.

SGLR uses a graph structured stack (**GSS**) to hold information during the parse process. GSS is made up of a directed graph which consists of nodes and directed edges. The nodes of GSS hold a parser state. **Parser states** form an index into the parse table. Each edge of GSS holds a parse tree. This is a sub tree of the final result of the parse process.

There exists one node that has no edges leaving from it. This is the **bottom of the stack**. This node holds the **startstate**, and it is created when parsing begins.

The nodes in GSS can be divided into **levels**. The number of levels is equal to the number of parse cycles, but the switch from one level to another is made in the middle of a cycle not at the end of it. The level to which newly created nodes are added is referred to as the **current level**. From each node in the current level, there exists a path to the bottom of the stack.

The leaves of the combined trees, which are held by the edges of such a path, form exactly the consumed part of the input string.

During each parse cycle, actions are retrieved from the parse table and executed. **Actions** are retrieved for a given node by means of the state of that node and the current token. There exist four kinds of actions: shift actions, reduce actions, accept, and error.

Error is not really an action, but the absence of other actions.

Accept is committed to eos. It marks a node as accepting. Typically this node has a direct edge to the bottom of the stack. That edge holds the resulting parse tree.

Reduce actions consist of a tuple $\langle production, length \rangle$. Where production designates the production rule and *length* denotes the number of tokens in the right hand side of *production*.

When reduce actions are executed, they replace a path with length *length* with a new edge. The tree that this new edge holds has *production* as its root. This root has as its branches the trees which were held by the edges of the path that this new edge replaces. The head of the new edge is the head of the path it replaces. If *length* is zero, then the head of this new edge is the node with the state that induces the reduce action. The state of the tail of the new edge is looked up in the parse table by means of *production* and the state that is held by the head of the new edge.

Shift actions consist of a goto state. When executed they add a new edge. The head of this new edge is the node with the state that induces the shift action. The state of the tail of the new edge is the goto state of the shift action. The new edge holds a tree that consists only of the current token.

Shift actions cannot be induced by eos and no other token than eos can induce accept. Therefore accept and shift actions are mutually exclusive for a given token.

Each parse cycle consists of two phases, the **reduce phase** and the **shift phase**. It starts with the reduce phase. During the reduce phase, for each node in the current level the actions are retrieved. With the start of the first cycle, the current level consists of the bottom of the stack.

If the action is error then it is ignored. If the action is accept, then the node is marked as **accepting stack**.

Reduce actions are executed immediately. New nodes created as a result of a reduce action are added to the current level. And actions for those new nodes are also retrieved.

Shift actions are stored in the shift queue. This **shift queue** consists of tuples $\langle node, gotostate \rangle$.

If all possible reduce actions are executed, then the shift phase is started.

If, at the start of the shift phase, the shift queue is empty, then parsing halts. When parsing halts, there are two possibilities. Either there exists an accepting stack in the current level, which indicates a **successful parse**, or no accepting stack exists which indicates a **syntax error**. The latter does not mean that no reduce actions has been executed during the reduce phase of the same parse cycle. I will call the current token for the parse cycle in which an error is detected the **error token**.

If the shift queue is not empty, then parsing commences with execution of the shift actions. At the start of the shift phase a new level in GSS is started. During the shift phase all shift actions in the shift queue are executed. The nodes created during this phase form the new current level.

When all shift actions are executed, then the current token is added to the consumed tokens and the next token from the input string becomes the current token.

After this all nodes from the lower levels that have no inedges, can be removed from GSS. Together with the edges they are the tail of. Removal of a node brings about other nodes with no inedges, which could also be removed and so on. This process is referred to as **garbage collection**.

There are two important restrictions to the process described above. Firstly no level in GSS can contain two nodes with the same parser state. Instead any node can be the tail of multiple edges. Secondly no nodes in GSS are connected through more than one edge. Instead of inserting edges with the same head and tail, the trees that those edges hold, are stored as **ambiguity clusters** in the **ambiguity table**. The **filter process**, which follows parsing, decides how to handle these ambiguities. Depending on the filters used, trees from the ambiguity cluster are either inserted into the parse tree or discarded.

These two restrictions make GSS memory efficient.

The nodes in GSS could be classified in two ways. Firstly, as a result of the action that created them, nodes could be divided in **shift nodes** and **reduce nodes**. Secondly they could be classified as a result of the action or actions they induced.

After the reduce phase four kinds of nodes exist in the current level:

1. The nodes in the shift queue or the node marked accepting stack. These are the **active stacks**.

2. The nodes which induced a reduce action of zero length. These are part of a path from an active stack to the bottom of the stack.

3. The nodes which induced no action, that is induced error. These are not usually distinguished from the nodes in category four, but as this thesis is about error handling, I will call them the **error stacks**.

4. All other nodes not in the above categories. These are the nodes that induced no other actions than reduce actions with length greater than zero.

The nodes in the third and fourth categories are subjected to garbage collection.

For a complete discussion of the SGLR algorithm together with all restrictions and exceptions see [VISSER1997]. For simplicity these restrictions and exceptions have been left out of the above explanations. However one exception has to be mentioned. This is the existence of reject productions.

Reject productions are used to achieve a prefer keyword disambiguation rule. Reject productions can introduce **reject nodes** into GSS. Reject nodes are nodes on which no action is possible, although their parser state tells otherwise. Also the shift queue might contain reject nodes. Therefore, whenever an empty shift queue is mentioned in this thesis, one should read: an empty shiftqueue, or a shiftqueue containing only reject nodes.

2.2 Error handling

The error handling process consists of three steps: error detection, error reporting, and error management.[DEGANO1995] This section will provide background for each step from the literature about compilers and type inference tools, which use a LR(k) parser. Specific literature on error handling by GLR parsers has not been found.

2.2.1 Error detection

The grammar specifies whether an input string is acceptable or not. Disagreements with the syntactic rules specified by the grammar are called syntax errors. An error is not necessarily detected where it is made. In the remainder of this thesis, I will refer to the current token at the point of error detection as the error token.

An erroneous part of the input string is a part that when it is replaced by something else, then the input string will be correct. The smallest possible erroneous part is the part that gets replaced by the minimum distance technique. (see section 2.2.3) The error token is not necessarily contained in, or directly following, the smallest possible erroneous part. Nor is it certain that replacing the smallest possible erroneous part will render the structure the user intended.

Two things are certain however. The consumed part at the point of error detection is a prefix of some sentence of the language, and the consumed part extended with the error token is not a prefix of any sentence of the language.

An error has one of two possible causes. Either the end of the input string is reached while the stack is not in an accepting state. Or, when all reductions for the current token are done, there exists no shift action for the current token. Each time a parser reaches an error condition, it has to do two things: First compose a message to inform the user about the error. And secondly recover from the error state as good as possible.

2.2.2 Error reporting

[HORNING1974] described the characteristics of good error messages for compilers. [YANG2000] defined a manifesto for good type error reporting, which addressed about the same issues, basically leaving out only the demand for politeness. The same characteristics could be used to test the soundness of syntax errors reported by a parser. Combined together and adapted to suit syntax error reports, we can derive the following list of desired properties of good error reports:

1. Correct. This implies both correct detection; all errors must be found, as well as found errors must be reported correctly.
2. Precise. The cause of the error must be located in the smallest part of the input string such that everything that contributed to the error is included.
3. Succinct. Each error report should maximise useful information, but minimise non-useful information.
4. Source-based and A-mechanical. An error report should not be based on parser states or actions, but must be expressed in terms of the defining grammar and input sentence.
5. Unbiased. An error report should not take for granted that everything to the left is correct because it is successfully parsed, nor should the report assume that the defining grammar is correct.
6. Readable.
7. Polite and Restrained.

2.2.2.1 Multi-stage error reports

While the parser state in itself will not help the user much, (point 4 above more or less forbids it to be reported to the user), [JEFFERY2003] describes a method in which the parser state is used to generate messages that are tailored to the language at hand.

This method assumes that there is a direct relation between the parser state and the type of the error. Therefore it maps parser states, possibly refined with the current input token, to descriptive messages.

Small changes in the grammar can completely change the meaning of all parser states. Therefore the error message table needs to be adapted each time the grammar is changed. This can be a tedious job when it has to be done by hand.

To automate the building of the error message table, Jeffery uses a table that maps small example strings that expose the error, to the relevant messages. This table can then be used to generate the error message table. Usually, changes in the grammar will affect only a limited number of examples.

2.2.3 Error recovery

[HORNING1974] distinguishes six possible reactions on errors:

1. Crash or loop
2. Produce invalid output
3. Quit
4. Recover and continue checking
5. Recover and continue compilation
6. Correct

The first three are called less desirable. The last one could theoretically be achieved with the minimum distance technique [AHO1972], but is considered unwanted because it is too time consuming.

Syntax errors can be recovered from by altering the stack and or altering the input string. Altering the stack is thought of as impossible or at least very dangerous. Thus several techniques are discussed that aim to alter the input string in a way that is acceptable. Preferably in a way the user meant the input string to be.

The simplest way is called 'panic mode recovery', which just skips input tokens until an acceptable token is reached. This could be done in a language independent way, but usually a set of synchronization tokens is used (for instance a statement delimiter like ;), from which the parse could restart in any situation. Main disadvantage of this approach is that a (possible large) part of the source is not checked. Later research is mainly aimed at improving recovering techniques by reducing the length of the unchecked part.

[DEGANO1995] describes and classifies those techniques for LR parsers. The distinguished classes are:

Non correcting recovery; A non correcting recovery technique, as for instance the one proposed by [RICHTER1985], tries to detect syntax errors without any correction. It only locates them. Contrary to panic mode recovery, which is also a kind of non correcting recovery technique, those techniques don't discard parts of the input string. Instead they try to demonstrate that the part after the error is a substring of the language.

An advantage of this approach is that no spurious errors are introduced, as a consequence of earlier corrections. Introduction of spurious errors is a problem with all correcting recovery strategies.

Disadvantages are that sometimes an error masks another error, and most time the resulting parse tree is not suitable for further semantic analysis.

Phrase level recovery; Phrase level recovering techniques are characterized by a two phase approach: The condensation phase collects the part of the input string that contains the error. It is followed by the correction phase which returns an error description together with the action taken to resume parsing.

The condensation phase consists of a forward move in which it continues to parse, without left context, the unread portion of the input string, thus collecting right context. It does so until an attempt is made to reduce over the error token.

Some of the earlier phrase level techniques as for instance [GRAHAM1975] started with a backward move, in which as many reductions were made as possible, in order to collect left context. The gathered information can then be used to select a correction.

For this class two subclasses could be distinguished. One is called the **error production recovery** class, which uses special error productions and a special synchronization token to guide the synchronization between stack and corrected input string.

Another subclass is called the **validation recovery** class. This class attempts to validate a correction by parsing ahead. If a certain number of tokens are shifted without an error then the repair is accepted, otherwise a different repair is tried.

The major problem of this method is to determine whether the found second error is a spurious error introduced by the repair or just another actual error. In the first case another repair to the first error should be tried, and in the latter case a new error recovery process should be started which should try to repair the second error.

Experimental results have shown that phrase level recovery does not work well with productions that have a right hand side consisting of both terminals and non-terminals. [SIPPU1983] introduced a way to reduce these effects.

Local recovery; Local recovery techniques differ from the phrase level techniques in that they limit the forward move to one token. Thus they improve the efficiency of the error recovery. Considered modifications are insertion, deletion, substitution, fusion of two adjacent tokens, and spelling corrections.

The main properties of local recovery techniques are: language independency, usage of standard parse tables, easily interfacing with semantic processing, and no augmentation of original grammar. As with phrase level recovery, it is not guaranteed that a correction is found. Therefore a secondary recovery technique like panic mode recovery should be in place.

Global recovery; Global recovery techniques use a large portion of the input string. Therefore better corrections are obtained. They are inefficient because they spend equal efforts on correct and incorrect parts of the program.

To make the handling time of an error independent of the length of the input string, regional recovery methods have been proposed. These consider only a fixed limited part of the context. To that purpose, particular symbols are selected that uniquely identify the context where they must appear.

Interactive recovery; A method is called interactive when it asks the user to correct a detected error and restarts the analysis from the point which is

as near as possible from the modified tokens.

An interactive technique provides the user with information about the error and suggests possible corrections. Because the location of detection of an error is not always the exact location for the repair of the error, a user should be allowed to make corrections everywhere. This raises the problem of deciding which part has to be reanalyzed. To be able to do that, auxiliary information must be kept during parsing; this decreases performance even when an input string is correct.

A major advantage is that all errors are detected in a single pass and that no spurious errors are introduced. These methods are particularly suitable for use with syntax directed editors and incremental parsers.

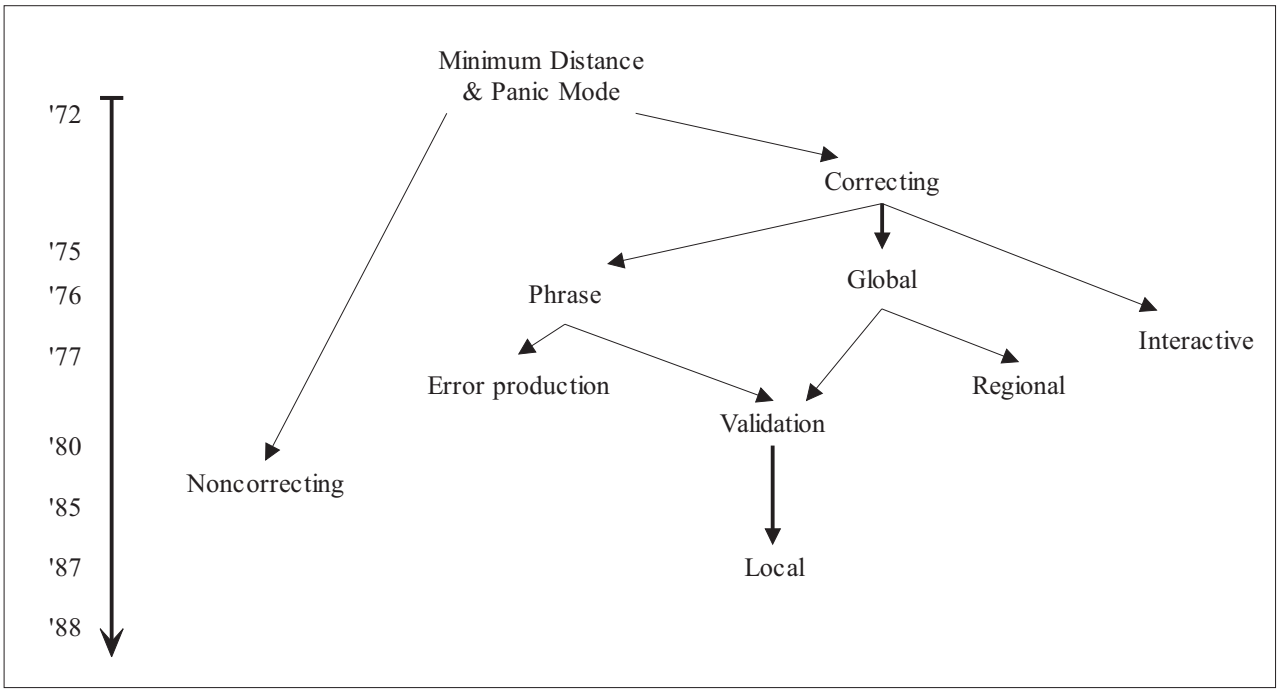

Figure 2.4: Relations among error recovery classes.[DEGANO1995]

[DEGANO1995] mentions two dimensions by which the efficiency of error management techniques could be measured. Firstly normal non-error execution time and storage space must not be incremented considerably. And secondly error processing time must be independent of input string length. They compare the various techniques on: quality of correction, language independency this is the possibility of changing the language without changing the error handling routine, flexibility this is the possibility of tuning the error handling routine, and performance degradation.

2.3 Summary

The literature about error handling by LR(k) parsers is focussed on finding as many errors as possible, thereby minimizing the not interpreted part of the input sentence. Contrary to SGLR, a parse tree is always returned. This tree can be further interpreted, and mostly will be, by type inference tools, type checkers, and compilers. Their error messages will provide additional information.

Also, with LR(k) parsers, the output of the scanner can be used for syntax highlighting and pretty printers. Being scannerless, with SGLR this is only possible with the full parse tree.

Next chapters will describe some approaches to achieve an interpretable output with SGLR, together with information about the nature of the error.

Chapter 3

Structure of consumed part

3.1 Current implementation and proposed improvement.

One important thing that can be reported to the user is how the input string has been interpreted so far. In the Meta-Environment there exists already three means of communicating the structure of the parse tree: highlighting, showing the non-terminal under the cursor, and visualizing (parts of) the parse tree. If a part of the error report exists of a parse tree in some form, then these existing aids can be used to help the user understand why an error is detected in the given place.

The current implementation makes no effort to inform the user of any interpretation of the already consumed part of the input string. This information, existent in GSS, is simply lost.

It is in the nature of SGLR that there exists more than one possible interpretation of the input string up to the token that caused the error. (see section 2.1.1) Therefore the ultimate interpretation cannot be taken from GSS.

GLR uses a non-deterministic parse table. To solve this non-determinism it tries all possible interpretations, and continues parsing until no more possible interpretations exist. It is possible that the intended interpretation encountered an error while parsing commences because also a not intended interpretation exists.

This looks like the kind of error masking with non-corrective recovery as described in chapter 7. This is inherent to the GLR parsing algorithm, which won't detect an error until no more possible interpretations exist.

Presenting the structure of the parsed prefix could be done in the form of a partial parse tree. This tree is partial in two respects. It does not reflect the whole of the input sentence. And because its root is not the grammar's start symbol, it is not an acceptable tree for the grammar, although the partial tree itself complies to the structure defined by the grammar. If the consumed part of the input sentence would be a prefix for some sentence of the language, then the partial tree would be a sub tree from the parse tree for that sentence.

The next sections describe an algorithm with which one or more interpretations

in the form of a partial parse tree can be derived from GSS. For this algorithm three versions exist, which differ in the number of interpretations they give:

1. The "excluding" algorithm which produces interpretations excluding the last consumed token.
2. The "including" algorithm which produces interpretations including the last consumed token.
3. The "omniscient" algorithm which produces all possible interpretations of the consumed part of the input string included those already discarded before the point of error detection.

3.2 Algorithms

The algorithm used to derive a partial parse tree produces, like SGLR, a parse forest rather than a parse tree. (see figure 2.3) This parse forest has to be filtered with the same filters that are used for a parse forest resulting from an error free input string. (see section 2.1.1)

The filter process combines the parse tree that is the result of the parse process with the ambiguity clusters in the ambiguity table. Also, depending on the filters used, the filter process removes trees from the forest that don't comply to grammar independent disambiguation rules like longest match and prefer literals.

The algorithm used to produce this parse forest can be divided in two parts. The main part consists of a series of reduce actions that compresses GSS to two nodes connected by one edge. This edge holds the desired parse forest. Before the reduce actions can be started, the nodes that will form the source for these reduce actions has to be selected. The three versions of the algorithm differ in the selection procedure for these source nodes.

3.2.1 Reducing GSS to one edge.

GSS could be viewed of as a directed graph where each node holds a parse state. Each edge holds a tree that has tokens of the input string at its leaves. All edges are directed towards the (single) node that holds the start state a.k.a. the bottom of the stack. The combined trees from each path, from the nodes in the current level to the node that holds the start state, hold exactly the consumed part of the input string.

When an error is detected GSS will look something like figure 3.1(A), with at the left the bottom of the stack and at the right the nodes in the current level. Figures 3.1(B) to 3.1(D) show the consecutive reduce actions which must be done to produce a useful parse tree.

For each path from a node with no in-edges to a node with two or more in-edges the trees are collected into an auxiliary treenode. If two nodes get connected by more than one edge, then only one of them is actually present in the stack. Instead of inserting edges with the same head and tail, the trees that those edges hold, are stored as ambiguity clusters in the ambiguity table. The filter

process, which follows parsing, decides how to handle these trees. Depending on the filters used, trees are inserted into the parse tree or discarded.

This algorithm will not work when there exists cycles in GSS. Therefore the algorithm has to start by removing all cycles from GSS.

These reduce actions are repeated until GSS consists of only two nodes connected by one edge. This edge then holds a tree with exactly the consumed part of the input string at its leaves. It is however highly likely that the resulting tree contains one or more ambiguities that could not be filtered away. Figure 3.2 shows this algorithm in pseudo code.

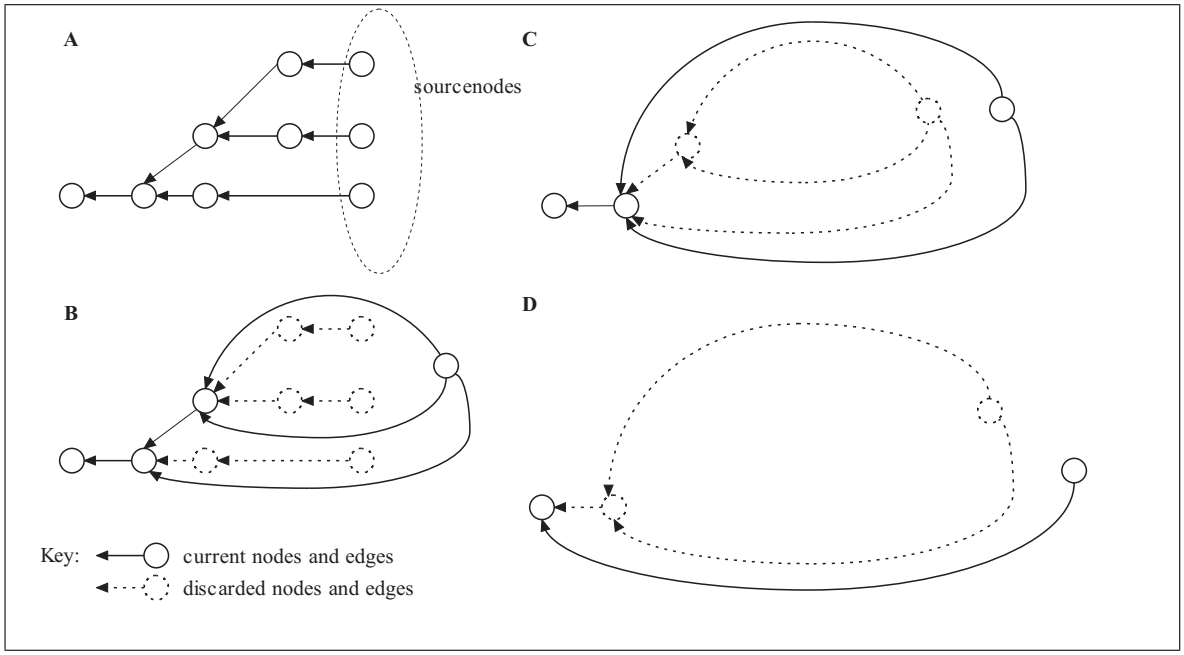

Figure 3.1: Reducing GSS

```
define ERRORGOTOSTATE := -2
define ERRORPROD := "<possibility>"
define PARSEDPROD := "<parsed>"

reduceParsedPart(StacknodeSet sourcenodes){
  remove all cycles from GSS
  while number of paths from sourcenodes to startnode > 1 do
    sourcenodes := handleAmbiguity(sourcenodes)
  TreeSet kids := the trees of the edges that form the single
                                  path from sourcenodes to startnode
  StackNode newStack := new stacknode with state ERRORGOTOSTATE
  reducer(startnode, newStack, kids, PARSEDPROD)
}

StackNodeSet handleAmbiguity(StackNodeSet sourcenodes){
  StackNode newStack := new stacknode with state ERRORGOTOSTATE
  for each StackNode source ∈ sourcenodes do
    for each Path path from stacknode source to a StackNode target that
                                              has more then one inEdge do
      kids := the trees of the edges that form path
      reducer(target, newStack, kids, ERRORPROD)
  StackNodeSet newsourcenodes := {newStack}
  return newsourcenodes
}

reducer(StackNode target, StackNode newStack,
         TreeSet kids, Production prod){
  Tree newTree := application of prod to kids
  if an edge from newStack to target exists then
    add newTree to ambiguitytable
  else
    create edge from newStack to target holding newTree
}
```

Figure 3.2: Pseudo code for the reduce phase of creating a partial tree.
startnode is the bottom of the stack. It is the only node in GSS that has no edges leaving from it.
ambiguitytable is used to collect all duplicate trees for the same part of the input string. During filtering, these trees are either rejected, or inserted into the resulting parse tree.

3.2.2 Selecting source nodes

An error condition arises when there are no nodes left in GSS that have a shift action. This means that all stacks have been discarded. Running the garbage collector at this point would render GSS empty. So which nodes should be used as source nodes to apply the above procedure to? There are three sets of nodes to be considered. And this is where the three versions of the algorithm differ.

A level in GSS consists of two sets of nodes: set A the nodes created while shifting the last consumed token, the shift nodes, and set B the nodes created while doing the reduce actions induced by the lookahead, the reduce nodes. The current level of GSS at the point of error detection consists of the shift nodes of the token before the error token and the reduce nodes of the error token. (see figure 3.3)

Both sets are possible candidates to act as source nodes for the reduce phase described in section 3.2.1. However both sets have their drawbacks.

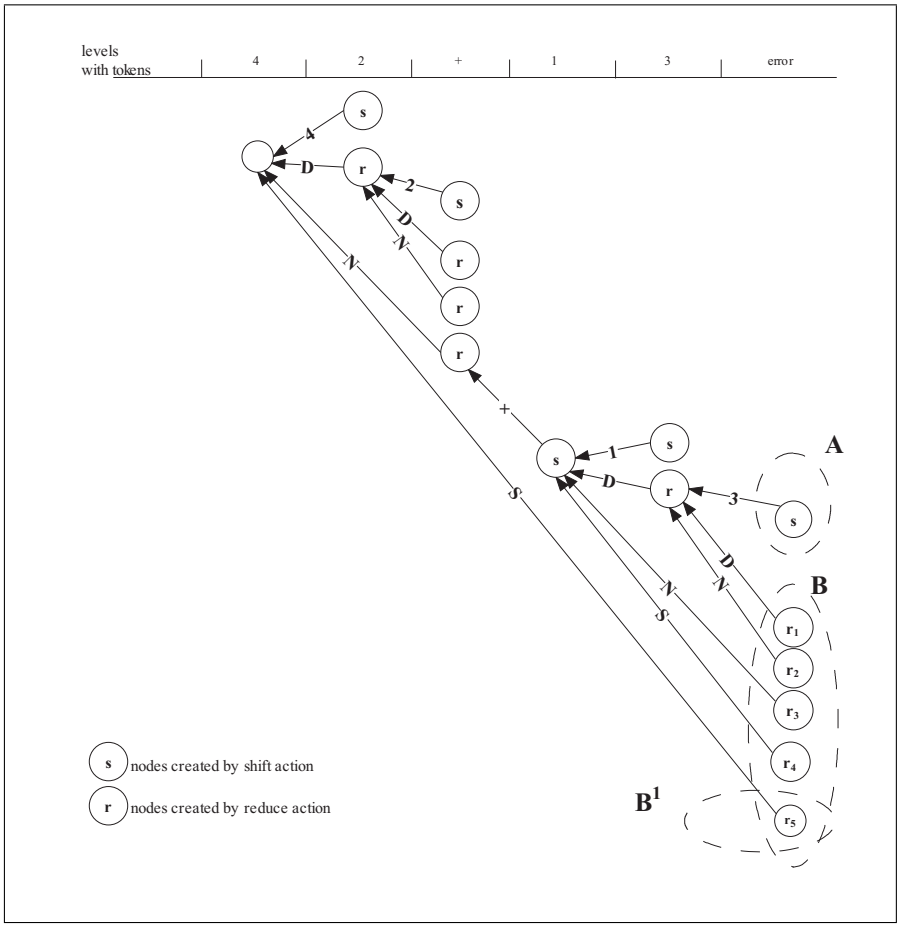

Figure 3.3: GSS upon detection of an error

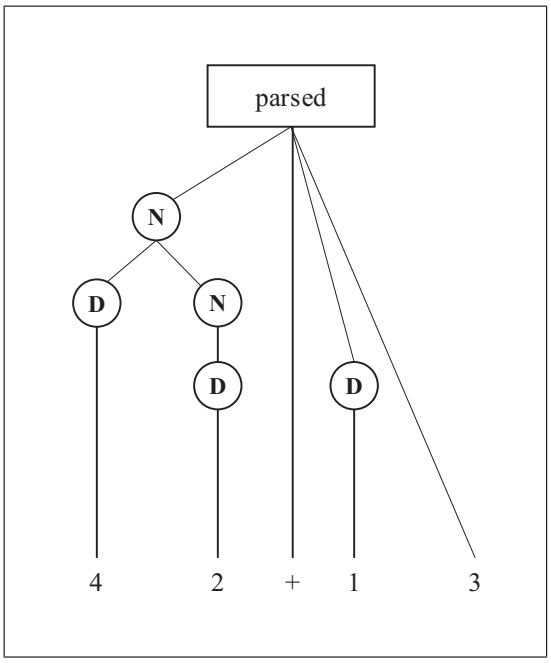

Figure 3.4: Partial tree for sentence "42+13error" as a result of the excluding algorithm

3.2.2.1 Excluding algorithm

The excluding algorithm uses set A as source nodes for its reduce phase. These nodes can simply be derived from the current level. But using these nodes means that the token just before the error token, the last consumed token, cannot be a part of any production, and will be left dangling on its own in the resulting tree. (see figure 3.4)

3.2.2.2 Including algorithm

The nodes of set B (figure 3.3) do lead to productions that incorporate the last consumed token. However the reduce actions that created those nodes depend on the token that caused the error. It is clear that, from the sheer fact that we have an error situation, these reduce actions are not all the possible ones. Therefore the nodes of set B cannot be used as a source for the including algorithm just like that. A set of nodes that does incorporate the last consumed token and includes all possibilities is computed with the following algorithm. That set contains the nodes that the including algorithm uses as source nodes for its reduce phase.

If we want to be complete, we first should look at all reduce actions for the nodes in set A independent of the lookahead. This can be done by performing all reduce actions possible for the nodes in set A. The nodes, created by those actions, that are only intermediate results can be discarded. Only those nodes that are possible shift targets should be used as source nodes for the reduce phase of partial tree creation.

```
StackNodeSet preparesource nodes(StackNodeSet shiftNodes,
                                 TokenSet reduceTokens){
  StackNodeSet sourceNodes := {}
  StackNodeSet usedNodes := {}
  for each Token token ∈ reduceTokens do
    parseToken(shiftNodes, token)
    sourceNodes := sourceNodes − currentLevel
    sourceNodes:=sourceNodes ∪ (shiftQueueNodes − usedNodes
    usedNodes:=usedNodes ∪ (currentLevel − shiftQueueNodes)
    currentLevel := {}
    shiftQueue := {}
  return source nodes
}

parseToken(stackNodeSet actives, Token currentToken)
This function is essentially the function PARSE-CHARACTER from [VISSER1997]
it performs the reduce phase of the parse cycle. Reduce actions
are executed. Newly created nodes as a consequence of these
reduce actions are added to the currentLevel.
Shift actions are collected in the shift queue. The set actives is
extended with the newly created nodes and actions for those are
performed also.
```

Figure 3.5: Pseudo code for computing the set of source nodes for the inclusive algorithm.

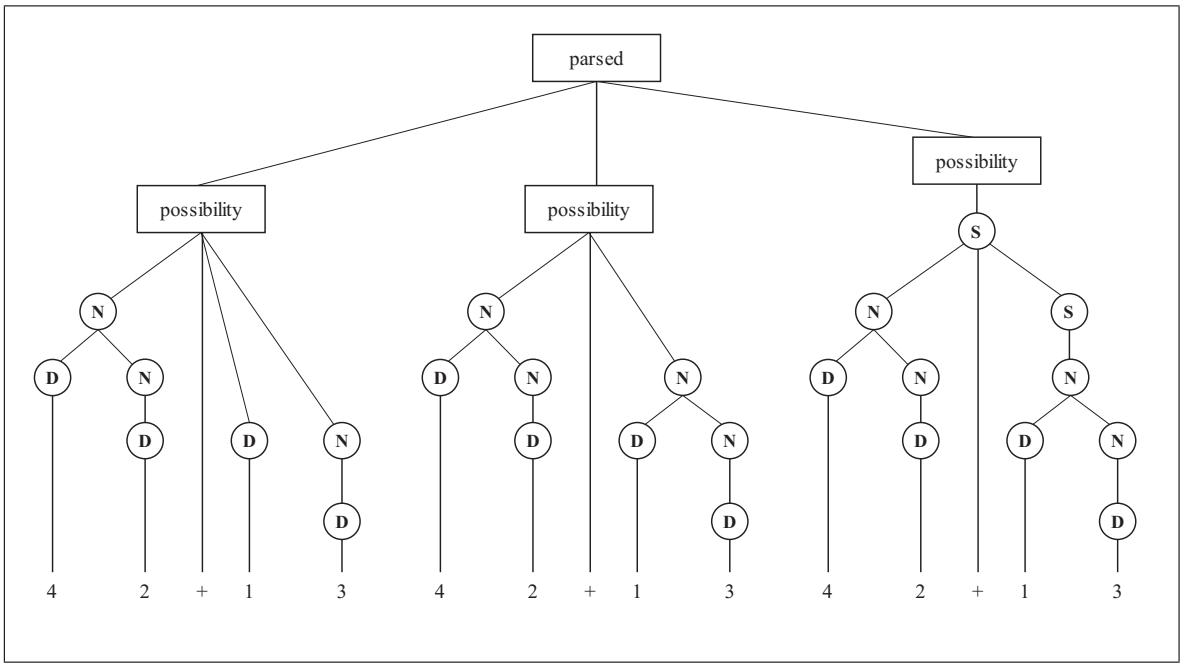

Figure 3.6: Partial tree for sentence "42+13error" as a result of the including algorithm with maximum redundancy. Each possible tree is an extension of the tree to its left. The leftmost tree is a result when the lookahead is a digit, the second one when the lookahead is a + sign and the last one when the lookahead is the end of the string. By means of removing redundancy the including algorithm will only produce the rightmost tree.

However, this introduces redundancies. These redundancies will be shown in the resulting parse tree as ambiguities. In the example of figure 3.3, if the lookahead is a digit, then node r_1 and $r_2$will be created and $r_2$will be a shift target. If the lookahead is the + sign, then nodes r_1 to r_3 will be created, but only r_3 will be a shift target. The eos token will create $r_1$to r_5, none of which are shift targets, but r_5 will be designated as accepting stack. (figure 2.2) Therefore, when we had used only the eos token as lookahead, we would have had all information we wanted without any redundancy. Figure 3.6 shows all those trees.

This redundancy could be decreased by removing all those nodes that are an intermediate result in the creation of another. This will produce set B^1 Unfortunately because of the way context-free layout is handled, not all of this redundancy could be removed that way. Figure 3.5 shows this algorithm in pseudo code.

3.2.2.3 Omniscient algorithm

It is possible that the intended interpretation encountered an error somewhere to the left of the error token. In that case the intended interpretation won't be revealed by the including algorithm.

If we want to be sure that among all possible trees, the intended tree is detected, we have to consider all error stacks that are produced by the parse process so far. (section 2.1.1) Collecting these is possible by maintaining an extra stack to which every token is shifted. This stack forms a direct path to the bottom of the stack.

?

Each edge on this path holds an one-node tree which leaf contains one token from the input string. The nodes on that path each have edges to the error stacks of the previous cycle, i.e. the nodes for which no shift or reduce action existed. (figure 3.7)

To be perfectly clear no nodes outside sets A and B in figure 3.3 fall in this category. Those nodes that have no in-edges actually had a shift or reduce action and are not terminated parse stacks, but leftovers from a non-terminated parse stack.

Alternatively, to minimize memory usage the stack that has to be kept a reference to, could be reduced to one edge by the process described in section 3.2.1.

The omniscient algorithm adds this extra stack to the set of source nodes for the including algorithm and uses the result for its reduce phase.

3.2.3 Implementation

During the course of this project, the creation of the partially tree as discussed in section 3.2 is fully implemented and incorporated into the current implementation of SGLR. SGLR uses the 'universal parse tree representation and manipulation format' (UPTR) to express the parse trees it produces.

A proper expression of a partial tree in UPTR would use auxiliary nodes for holding the parsed and not-parsed parts and to distinguish between different possible interpretations. However no such nodes currently exist in UPTR.

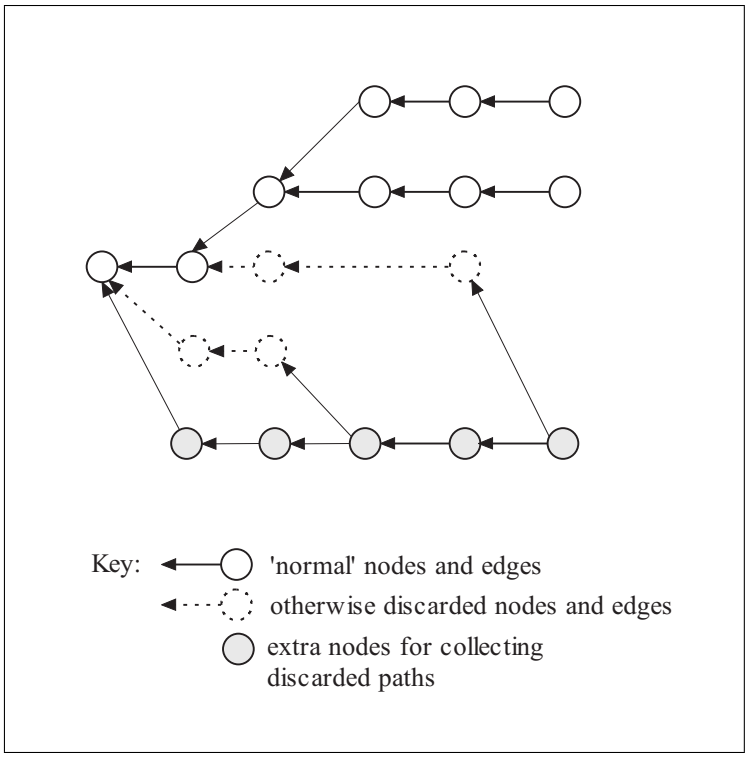

Figure 3.7: Collecting all error stacks. Those enable the omniscient algorithm to present all possible interpretations.

No additions are made to UPTR so far. Instead of the needed auxiliary nodes, ordinary productions nodes are used. Instead of ordinary productions from the grammar, these nodes show productions with an undefined left-hand-side and a descriptive message as the right-hand-side.

In order to know the number of in-edges for a particular node in GSS, the method of garbage collecting has been changed in such a way that this number is updated when nodes are deleted. The number of in-edges is needed to determine the length of the reduction paths.

Implementing the inclusive algorithm turned out to be rather awkward. In order to let the ambiguity table do its work, multiple tokens need to be parsed on the same level of GSS. This means that when the reduce phase of a parse cycle is executed for a specific token (parseToken() in figure 4.1), the current level contains nodes that normally would not be there. Therefore some nodes would already exist which normally would be created. Special provision are needed to process nodes, as with normal parsing.

After each token the nodes used by that token (those that would normally form the current level) are checked. The following sets are distinguished:

- Nodes that didn't exist before and either have the accepting state, or the given token can be shifted to i.e. are in the shift queue. Those nodes are added to the set of source nodes. Due to handling of context-free layout, a little alteration is necessary: When a node's only edge has as its tree a layout tree that is not reflected in the input string, then the target node of this edge should be added to the set of source nodes instead.
- Nodes that existed already in the current level and for which the given token is a useful token i.e. doesn't lead to an error stack. Those nodes are removed from the set of source nodes.
- Nodes that didn't exist before and for which the given token is not a useful token. Those nodes are removed from the current level.

3.3 Analysis

This section evaluates the quality of reporting the structure of the consumed part. It does this on the basis of the list of requirements formulated in section 2.2.2. The polite requirement has been left out because this report does not adress the user directly. The list is completed with the efficiency and flexibility criteria of [DEGANO1995].

The other criteria that Degano uses, correction quality, and language independency are either not applicable for any of the techniques proposed in this thesis or already covered by the other requirements.

3.3.1 Correct

It is proved by [REKERS1991] that the GLR algorithm correctly rejects strings that are not part of the given language. Therefore no accepted input sentence will contain syntax errors.

The stack of a LR parser always consists of a correct interpretation for a prefix of some sentence of the language.[DAIN1994] GSS is a combination of one or more stacks of LR parsers, therefore it holds one or more interpretations of a prefix of some sentence of the language. The partial tree created by the process described in section 3.2.1 is derived from GSS and will always be a correct interpretation for a prefix of some sentence of the language.

3.3.2 Precise

This report takes the whole consumed part of the input string into account, and it uses exactly what is needed to generate the report. Also the exact location where the error is detected is part of the report. No measures are taken to communicate which part of the input string does not contribute to the error.

3.3.3 Succinct

There is a trade-off between being complete and decreasing the amount of non-useful information. The excluding algorithm won't introduce any redundancy, but will provide for the least amount of information. The including algorithm increases the amount of useful information, but may introduce redundancy. The omniscient algorithm, finally, will guarantee completeness, but introduces lots of non-useful information.

3.3.4 Source-based

The partial tree is based on the defining grammar just like the parse tree that would have been the result when no error was encountered. However some nodes specific for erroneous trees have to be inserted.

3.3.5 Unbiased

The partial tree is biased as it reports one or more possible interpretations for the consumed part of the input sentence according to the given grammar. It assumes that both the interpreted part of the input sentence as the grammar is right. No effort is made to locate the error anywhere else than on the point of detection.

3.3.6 Readable

The Meta-Environment has various ways to show the structure of a parsed sentence, such as highlighting, pretty printing, showing the right-hand-side of a selected production, and visualisation of the parse tree. Except from pretty printing, these can all be used for erroneous parse trees as created by the error handling process.

However redundancies introduced during creation of a partial tree and multiple interpretations within a partial tree are shown as ambiguities in the resulting parse tree.

Unfortunately, the Meta-Environment offers no easy way for discovering differences between two or more interpretations of (a part of) the parsed sentence. Additional work has to be done there, which will also benefit the grammar writer who wants to write an un-ambiguous grammar.

3.3.7 Efficiency

One of the efficiency criteria for error handling is that it must not increase time and memory usage for parsing an error-free input string considerably [DEGANO1995]. For creation of a partial tree this differs with the algorithm used. The excluding algorithm will increase memory usage slightly, because shift nodes needs to be distinguished from reduce nodes. The including algorithm also starts of with the shift nodes and adds no costs to those for the including algorithm. The omniscient algorithm however places a big burden on memory usage which could be reduced somewhat, but that will increase time consumption for parsing an error-free input string.

The other efficiency criterion is that error handling time must be independent of the length of the input string. Computing time of the reduce actions is linear with the number of possible paths through GSS. This number is not directly related to the length of the input string, but depends on how the grammar has been constructed and on where in the input string the error is detected. Computing the source nodes for the including algorithm depends also on the nature of the grammar (O($\mathit{reduce\text{-}actions} \times \mathit{number\text{-}of\text{-}tokens}$)) and is also independent of the length of the input string.

3.3.8 Flexibility

There is a trade-off between completeness and the amount of (non-useful) information. This provides possibilities for fine tuning the error handling. The preferred or default algorithm for creation of a partial tree would probably be the including algorithm, but the excluding algorithm (no redundant information) or the omniscient algorithm (complete, but only feasible for small sentences) could be offered as an option.

3.4 Conclusion

A partial tree can inform the user about how the consumed part of the input sentence is interpreted by the parse process. This is an advantage over the current implementation which provides no information. Although it is biased on the consumed part of the input sentence, the user should be able to, by comparison of the presented interpretation(s) with the intended interpretation, to determine the nature of the error.

For syntax errors this bias on the parsed prefix of the input sentence is less an issue then for instance for type inference tools. In that case, it is naive to assume that the first type assignment will be correct when it is the only one that is deviant. With syntax errors usually no correction solution has preference over another. For instance when bracket pairs are out of sync, the error won't

be detected until there really is a closing bracket missing or superfluous. This could be corrected by inserting or deleting a bracket anywhere left of the error detection point. (But of course right of the point where they were in sync last.) Any of these solutions is as good as another. By the way, the case of a superfluous closing bracket is an example where the current error message is very accurate.

The including algorithm is in most cases the algorithm that provides the best possible feedback. For grammars where the interpretations, produced by the including algorithm, are to blurred to be useful, the excluding algorithm can be used as an alternative.

The omniscient algorithm is neither succinct nor efficient, but it is very complete. It provides all possible interpretations of all possible prefixes of the consumed part of the input sentence. Therefore it may be useful for situations where the error in the intended interpretation gets concealed by other interpretations.

Presenting this information, gives the user an opportunity to select the track he intended, and follow it to the point where it went wrong. However the amount of information would quickly explode. For instance in C, a simple assignment would already provide for at least four error stacks, each of which will bring about a separate, ambiguous branch in the resulting partial parse tree. Also collecting these discarded paths places considerably costs on errorfree parsing.

Therefore the omniscient algorithm is only feasible for small sentences of languages where it is likely that the actual error is somewhere left of the point of error detection.

3.5 Summary

This chapter presented three algorithms to create interpretations of the consumed part of the input sentence in the form of a partial parse tree. The algorithms differ in the number of interpretations they give. Although every presented interpretation is a correct one for the consumed part of the input sentence, there is a tradeoff between completeness and the amount of redundant useless information.

Next chapter present a method to extract a list of symbols that are an acceptable continuation of the consumed part of the input sentence.

Chapter 4

Expected symbols

4.1 Current implementation and proposed improvement.

The Meta-Environment already provides support for error reporting. Each error report consists of one verbal message, followed by one or more sub messages. To each sub message, an exact location (URL + location in the file) can be attached. The current error report tells the user what is not expected, namely the token for which there were no shift actions available.

This can be extended with the tokens that are expected. The expected tokens are those tokens that when they were in the place of the error token would induce a shift action. If one of the expected tokens was in the place of the error token, then an error would not occur, at least not at that location. Daniel Jackson uses this approach for the parser used by his Alloy Analyzer.[ALLOY]

However SGLR uses tokens of one character each. Those would not provide for very readable and succinct error reports. The aim is to extend those tokens to what they are the prefix of.

Next sections describe an algorithm that selects the expected tokens, and extends them, if applicable to context-free literals.

Context-free literals are the quoted words or characters in the lefthandsides of production rules expressed in SDF. As for instance in the following production rule from the grammar of the pico example language:

```
"if" EXP "then" {STATEMENT";"}* "else" {STATEMENT";"}* "fi" -> STATEMENT
```

This is the most used method to define keywords in SDF.

The algorithm with which single tokens could be extended to character classes or literals is based on the fact that the normalization process of SDF replaces context-free literals by a production that absorbs a series of tokens into the desired literal.[VISSER1997]

For instance, for the SDF production rule ”begin” StatementList ”end” -> BlockStatement the following production rules are added by the normalisation process:

[b][e][g][i][n] -> ”begin”

[e][n][d] -> ”end”

The square brackets denote characterclasses, the sets of characters to choose from. In this case each characterclass consists of exactly one character. In the parse table the production rules of this type can be distinguished by means of the existence of a direct shift action from one characterclass to another. For instance if the last consumed token is 'e' then there exists a shift action when the current token is 'g' for the "begin" rule and when the current token is 'n' for the "end" rule.

Separate tokens could be extended to character classes or literals, by looking at right-hand-side of the production that absorbs them. (In SDF this is the right-hand-side. In [DEREMER1971], BNF, and many other grammar notations this will be the left-hand-side.)

The right-hand-sides of the found productions, may contain some already accepted tokens. For instance: In the above example If `begin` was misspelled as `beg!n` and `beg!` could not be interpreted in any way, but `beg` could be interpreted, for instance as the prefix of an identifier, then the error will be detected with the `!`, but the expected symbol found will be `begin` not `in`. Probably also the character class [a-z] will be reported as expected, when these characters can be used to provide for a valid identifier.

To be able to determine what tokens and symbols are really expected the whole of GSS at the moment of error detection is necessary. This cannot be derived from the parse states of the last active stacks alone. Only the whole of GSS can provide the necessary context.

4.2 Algorithm

4.2.1 Selecting what is expected

4.2.1.1 Selection of tokens

As discussed in section 3.2.2 the current level of GSS at the point of error detection consists of the shift nodes of the token before the error token and the reduce nodes of the error token. Sets A and B in figure 3.3. The nodes in set B owe their existence to the error token and are not very interesting. The tokens we are looking for, the expected tokens, are the tokens that, when they were the current token, would leave the current level with one or more active stacks after the reduce phase of the current parse cycle.

Because the current parse cycle starts with set A as the current level, the search for expected tokens also starts with set A. From the parse table we can retrieve all tokens that induce an action other than error for any of the nodes in set A. According to their actions, these nodes could be divided in shift tokens and reduce tokens.

The shift tokens induce either a shift action or accept for any of the nodes in set A. The reduce tokens induce a reduce action for any of the nodes in set A. The set of shift tokens is not necessarily disjoint from the set of reduce tokens. These sets can be made disjoint by removing all shift tokens from the set of reduce tokens. that leaves the set of reduce-only tokens. Now all possible tokens are divided into three disjoint sets, all with respect to the nodes in set A.

–The tokens that induce no action at all are certainly not expected.

```
TokenSet expectedTokens(StackNodeSet shiftnodes){
  TokenSet shiftTokens :={}
  TokenSet reduceTokens :={}
  for each StackNode stack ∈ shiftnodes do
    shiftTokens := shiftTokens ∪ lookupShiftTokens(stack)
    reduceTokens := reduceTokens ∪ lookupReduceTokens(stack)
  TokenSet reduceOnlyTokens := reduceTokens − shiftTokens
  TokenSet usefulTokens :={}
  for each Token token ∈ reduceOnlyTokens do
    if isUsefulToken(shiftNodes, token) then
       usefulTokens := usefulTokens ∪ token
  return shiftTokens ∪ usefulTokens
}
TokenSet lookupShiftTokens(StackNode stack){
  retrieve from the parse table all tokens that induce a shift
  or accept action for the parse state of the given StackNode
}
TokenSet lookupReduceTokens(StackNode stack){
  retrieve from the parse table all tokens that induce a
  reduce action for the parse state of the given StackNode
}
Boolean isUsefulToken(StackNodeSet nodes, Token token){
  currentlevel := {}
  parseToken (nodes, token)
  Boolean returnValue := shift queue != empty or acceptingStack exists
  deleteshift queue
  return returnValue
}
parseToken() see figure 3.5
```

Figure 4.1: Pseudo code for selecting the expected tokens.

–The shift tokens are expected, as their actions immediately render an active stack for the shift phase.

–The reduce only tokens might or might not be expected. That depends on whether any of their reduce actions leads to an active stack eventually. To determine whether a reduce only token is expected or not, the reduce phase of a parse cycle has to be executed with that token as the current token. If after that the shift queue is not empty, or an accepting stack exists, the token is expected. Otherwise, the token is not expected. Figure 4.1 shows pseudo code for this algorithm.

4.2.1.2 Extending tokens to symbols

Figure 4.2 shows the pseudo code for an algorithm that finds the productions that absorb the expected tokens. It first performs the reduce and shift actions for each expected token to the nodes of set an of figure 3.3. Per token we then have a new set of nodes from which we can derive the desired production by looking at the reduce actions for each expected token in that situation. These are the productions that absorbs the expected tokens found in section 4.2.1.1.

The productions denoted by such a reduce action may only be added to the set of expected symbols when its consecutive reduce actions lead to an active stack.

Shift actions that are induced by a single token rather then by a set of tokens indicate that the tokens parsed so far, form the prefix of a literal. The process will have to be repeated for the tokens that induce those actions.

4.2.2 Implementation

During the course of this project both algorithms described in the previous section are fully implemented in the way described. These algorithms bring about repetitive extensions to GSS. Two things has to be observed here.

First the whole of GSS is needed to be able to decide whether a given token or symbol is really expected or not. This cannot be decided by the parse states of the shift nodes from the current level alone. Only the whole of GSS can provide the necessary context.

Secondly before processing each token and after the whole procedure, before creation of the partial tree described in chapter 3, the state of GSS should be an exact copy of the state at the moment of error detection.

To be able to fulfil both requirements, the methods that implement the reduce phase of the parse cycle are adapted. In the current implementation objects like current token and active stacks are defined globally. These have been parameterized where necessary.

Although GSS is implemented as a separate module, for its garbage collection procedure it depends on information from outside the module. Only nodes in the shift queue and in the current level can be addressed from inside GSS. This has been changed by means of a different implementation of the StackNodeSets.

In the Meta-Environment the sets of expected tokens and expected symbols are reported by attaching two sub messages to the existing error report.

```
ProductionSet findAllSymbolsRecursive(ProductionSet exptdSymbols,
                                      StackNodeSet nodes,
                                      TokenSet exptdTokens){
  storeCurrentLevel
  for each Token token ∈ exptdTokens do
    parseToken(nodes, token)
    TokenSet shiftTokens := lookupShiftTokens(nodes∪currentLevel,1)
    TokenSet reduceTokens := lookupReduceTokens(nodes∪currentLevel)
    expctdSymbols:= expctdSymbols ∪findAllSymbolsRecursive(exptdSymbols,
                                                           nodes ∪ currentLevel,
                                                           shiftTokens)
    expctdSymbols:= expctdSymbols ∪ appendSymbols(exptdSymbols,
                                                  nodes ∪ currentLevel,
                                                  reduceTokens)
  resetCurrentLevel
  return expctdSymbols
}
ProductionSet appendSymbols(ProductionSet exptdSymbols,
                            StackNodeSet nodes,
                            TokenSet reduceTokens){
   for each StackNode stack ∈ nodes do
    for each Token token ∈ reduceTokens do
      for each reduce action action ⟨stack, token⟩ do
        Production prod := production denoted by action
        for each path from stack with length denoted by action do
          storeCurrentLevel
          StackNode target := head of path
          StackNode newStack := new StackNode with state denoted by ⟨target, action⟩
          reducer(target, newStack, {}, prod)
          if isUsefulToken({newStack},token) then
            exptdSymbols := exptdSymbols ∪ prod
          resetCurrentLevel
return exptdSymbols
TokenSet lookupShiftTokens(StackNode stack, int max){
  retrieves from the parse table all tokens that induce an accept or
  a restricted shift action for the parse state of the given StackNode.
  For shift actions there is a restriction that no more than max tokens
  induce the same action.
}
lookupReduceTokens() see figure 4.1
isUsefulToken() see figure 4.1
parseToken() see figure 3.5
reducer() see figure 3.2
```

Figure 4.2: Pseudo code for extending expected symbols to expected symbols. The first call to findAllSymbolsRecursive has as parameters: an empty ProductionSet, the shift nodes of the current level, and the expected tokens as computed with the code in figure 4.1

4.3 Analysis

This section evaluates the quality of reporting the list of expected symbols. It does this on the basis of the list of requirements formulated in section 2.2.2, as well as on the efficiency and flexibility criteria of [DEGANO1995].

The other criteria that Degano uses, correction quality, and language independency are either not applicable for any of the techniques proposed in this thesis or already covered by the other requirements.

4.3.1 Correct

The list of expected tokens which is used as a starting point for the list of expected symbols, consists of the tokens that won't cause an error when they had been in the place of the error token. This list can be derived directly from the parse table, and will be both correct and complete.

The algorithm that extends those tokens to symbols collects only those symbols that possess a valid suffix when combined with the given prefix. Therefore those symbols are correctly identified as expected. However this list is not necessary complete. Only those symbols that are defined as context-free literals will be found.

4.3.2 Precise

This report takes the whole consumed part of the input string into account, and uses exactly what is needed to generate the report. Also the exact location where the error is detected is part of the report. No measures are taken to communicate which part of the input string does not contribute to the suggested acceptable continuation.

4.3.3 Succinct

The algorithm used to compute the list of expected symbols, also detects which character classes could be used as a correct continuation. Presenting those to the user is however seldom useful, as most of the time it is not precisely clear to what language constructs these character classes contribute. Also certain character classes as for instance layout characters are almost always acceptable. Therefore this list is limited to the symbols marked as literal.

Acceptable character classes are reflected in the list of expected characters.

Another issue here, is that often this list consists of a possible large number of different items which are essentially the same. For instance when, in C, an operator between expressions is appropriate, then this will produce a list of 32 possible operators. Other options in such a list will easily be overlooked.

4.3.4 Source-based

Only constructions of the defining grammar are used for these reports.

4.3.5 Unbiased

These reports assume that the consumed part of the input sentence is right and suggest a correct continuation. Therefore they are biased on the consumed part of the input sentence.

4.3.6 Readable

Generally lists, especially longer lists, do not make up very readable reports. However in the Meta-Environment they are presented in such a way that they only need to be looked at when the user thinks they will be helpful.

4.3.7 Polite and restrained

The politeness of verbal messages goes without saying, but is very much up to the implementer. However, it must by no means be forgotten, therefore it still appears in this list.

4.3.8 Efficiency

Computing the set of expected symbols starts off with the same state of GSS as creating the partial tree described in chapter 3. Therefore it imposes no additional strain on parsing errorfree sentences.

Computing the sets could be a costly operation, but it is unrelated to the length of the input string. It depends mainly on the number of tokens. For the current implementation of SDF this is limited to the 256 (extended) ASCII characters. Should this be expanded to for instance Unicode, then the proposed algorithm would not be satisfactory. In that case some other way to handle character classes should be used.

4.3.9 Flexibility

As a way of fine tuning to special circumstances, for instance with grammars that makes a very distinctive use of characterclasses, the list of expected symbols might be extended with expected character classes.

4.4 Conclusion

The usefulness of the list of expected symbols depends very much on the length of this list. At most one symbol of this list could be the intended symbol; therefore it is most useful when it contains only one symbol. And this applies only when one assumes that the consumed part of the input sentence is right. Unfortunately computing time also increases with the length of the list, and is therefore inversely proportional with usability. However as it does not influence parsing time of error-free sentences, and it does not strain handling time too much, this could be considered as a reasonable extension of the error report.

4.5 Summary

This chapter presented an algorithm to derive, from the current state of GSS, a list of expected tokens. Because the tokens in SGLR consist of only one character, this will be a list of expected characters. The expected characters are those characters that are a correct continuation of the consumed part of the input sentence.

Where appropriate, the characters in this list can be extended to the literal they are a part of. These lists are both correct and complete.

Usefulness decreases with the length of the list, where computing time increases with the length of the list.

Next chapter describes a way of supplying language specific error messages without alterations to either the grammar or the generated parsers.

Chapter 5

Language specific error messages

5.1 Current implementation and proposed improvement.

Currently SGLR has no provisions for the grammar writer to influence the generated error reports. Neither by making additions to the grammar, nor by adding specific functions to the generated parser.

According to [JEFFERY2003], each type of error together with a limited amount of context will result in the same parser state, independent of the larger context in which it appears. Therefore the reported parser state could be used to look up specific messages in a table.

Such a table could be constructed by parsing small examples of input strings that contain a specific error. That way relieving the grammar writer of the task of putting specific error messages into the parser implementation. A task which otherwise should be redone each time the grammar changes. For this purpose, Jeffery has built a tool called Merr that does this for parsers generated with Yacc.

If it is possible to infer from GSS a parser state that designates rather exact the grammar constructs currently parsed, then, in case of an error, this parser state can be used to look up language specific messages from a table.

A parser state is an index into the parse table. With LR(k) parsers such a parser state can be clearly distinguished between two consecutive tokens, looking at the top of the stack. The top of the stack always contains a parser state. In SGLR these states are blurred in two ways.

Firstly the error could occur in the middle of a lexical construct. Therefore misspellings of for instance `then` by `ten` , `tehn`, or `the` will each produce its own parser state. This is due to the fact that SGLR is scannerless.

Secondly, most of the times GSS will consist of more than one parse stack. The states of all these stacks should be reflected in the general parser state. To distinguish this general parser state from the indexes into the parse table which

are also called parser state, I will refer to the general parser state as the GSS parser state.

Because the GSS parser state is a combination of the parser states of all active stacks, a bigger part of the context is reflected in the GSS parser state than with ordinary LR(k) parsing. For instance, a parser state for a missing semicolon in a semicolon-separated statement list will also reflect the context of the statement list e.g. a while-loop or an if-then clause. This could be an advantage, because more specific messages could be used, but it increases the number of examples needed.

The Merr tool developed by Jeffery offers the possibility to also use the error token to distinguish between different reports for the same parser state. The single character tokens of SGLR are not very useful for this purpose. However, by means of the stacks that are used for the GSS parser state, also the error token is reflected in the GSS parser state, and with that also grammar constructs for which the error token is a prefix. This adds extra context information to the GSS parser state.

Next sections describe an algorithm that retains the GSS parser state and a prototype that uses this GSS parser state the way the Merr tool developed by Jeffery does with parsers generated by Yacc.

5.2 Algorithm

The GSS parser state is formed by the set of parser states that are held by certain nodes in GSS. The GSS parser state changes after the reduce phase of a parse cycle. That is the same point in the parse cycle as where an error is detected. The GSS parser state consists of the parser states of all error stacks. These error stacks are created by reduce actions inflicted by the error token. Contrary to when creating a partial tree, in this case the reduce actions induced by the error token provide valuable context information.

If the set of error stacks is empty, then we have the situation that the error is in the middle of a literal. In that case the active stacks of the last parse cycle that had reduce actions is used. This reflects the GSS parser state at the beginning of the literal, rather than at the point of error detection.

The error stacks cannot be distinguished from other nodes subjected to garbage collection. Therefore the parser states of the nodes that has no action are collected in a set of parser states during the function that retrieves the actions. After the reduce phase of each parse cycle, the function setParserState uses either that set or the shift queue to compose the GSS parser state. Figure 5.1 shows the algorithm used for setParserState.

5.2.1 Implementation

To communicate the GSS parser state, another sub message is added to the error message. The text of this message ends with a semicolon separated list of individual parser states.

To check whether the parser state could be used for any meaningful purpose, a little prototype java application has been made. This application processes a

```
define global: ParserStateSet GSSParserState, errorParserStates
setParserState(){
  if currentLevel contains reduce nodes then
    if shift queue is empty then
      GSSParserState := errorParserStates
    else
      GSSParserState :={}
      for each StackNode stack ∈ errorParserStates do
        GSSParserState := GSSParserState ∪ {ParserState from stack}
  errorParserStates :={}
}
errorParserStates contains the ParserStates from the error stacks
in the current level
```

Figure 5.1: Pseudo code for function setParserState. This function is called after the reduce phase of each parse cycle.

XML formatted file with example message pairs, and outputs a java hashtable, which stores the messages using the parser state as a key. This hashtable can then be used by the GUI to translate the parser state from the error report to the message stored in the hashtable.

From implementing and using the prototype, some issues could be derived, and a proper implementation should handle these issues:

- A possibility must exist to provide a common context for more examples. In the prototype one can specify the correct version of the example. This will then be used to start every subsequent example with.
- It should be possible that the same message is defined by more than one example.
- A warning should be generated when two examples, leading to the same error message, define the same state and are thus redundant.
- A warning should be generated when two examples define the same state, but lead to different messages. In this case only one can actually be used, thus blocking the other intended message.

The prototype has been used to create specific syntax error messages for the example programming language Pico. For 17 common errors specific message have been written. 20 examples were needed to cover all GSS parser states identifying those errors. In the used examples, all these errors were correctly identified.

5.3 Analysis

This section evaluates the quality of using the GSS parser state for grammar specific messages. It does this on the basis of the list of requirements formulated in section 2.2.2, as well as on the efficiency and flexibility criteria of [DEGANO1995].

The other criteria that Degano uses, correction quality, and language independency are either not applicable for any of the techniques proposed in this thesis or already covered by the other requirements.

5.3.1 Correct

The GSS parser state itself can neither be correct nor incorrect, it is just a property of the parser. Whether the reports generated with the aid of the GSS parser state are appropriate can not be guaranteed. It depends on the set of examples used, but this set will probably never be exhaustively. There will always be GSS parser states for which no translation is available. And probably also sometimes translations will be used for errors for which they were not intended.

5.3.2 Precise

As the other error reports do, the GSS parser state reflects the whole consumed part of the input string. But often it will be possible to derive from the GSS parser state the context of the error and communicate this in the report. From this the user can discover what part of the input string contributed to the error.

5.3.3 Succinct

Based on The GSS parser state very specific error reports are possible.

5.3.4 Source-based

The GSS parser state can be used to compose grammar specific messages. This is the main purpose to use this method.

5.3.5 Unbiased

As the other reports this reports is biased on the parsed prefix of the input string.

5.3.6 Readable

Messages that are used for these reports are specially written, and not composed from some generic literals completed with specific variables, like "<token> unexpected". The combined methods proposed in this chapter should therefore produce very readable reports.

5.3.7 Polite and restrained

The politeness of verbal messages that reach the user as a result of this error reports depends on the grammar writer whom writes those messages.

5.3.8 Efficiency

Keeping the parser state imposes no strain on error free parsing time. (see section 8.1.1) Including the parser state in the error report is almost free. Translation to a pre defined error message depends on the number of those messages.

5.3.9 Flexibility

No methods exist to adapt the methods proposed in this chapter. Except for whether the GSS parser state is actually shown or not shown. For the common user, this is useless information. But a grammar writer will need the GSS parser state to be able to find the right examples.

5.4 Conclusion

It has been investigated whether Jeffery's approach to use small example sentences for creating descriptive, language dependent error reports is feasible for SGLR. First thing needed for this method is a parser state. It turns out to be possible that, although GSS could exist of many parser states, a state can be derived from GSS that is unique for certain kinds of errors.

As a proof of concept a prototype has been developed such that the Meta-Environment could translate parser states to error messages. For the example language Pico this worked reasonably well. Parser states for spelling errors as well as omissions of keywords could easily be recognized. Parser states that uniquely identified errors related to uneven or tangled begin end pairs have not been found.

It cannot be guaranteed that found messages are appropriate in all situations. Also, it will prove to be a tedious job to provide enough examples as to cover a reasonable number of errors. Probably this method will only be useful for some domain specific languages that have a rather large group of users who are unaware of the language development.

However, this translation scheme will come on top of other error reports. Therefore even if only one peculiar error situation can be clarified, this will be an advantage.

5.5 Summary

This chapter presented an algorithm to derive a parser state from GSS. This GSS parser state can be used to, following the method described in [JEFFERY2003], firstly create a table of message indexed by those parser states and then use this table to provide these messages with the error reports for the actual errors.

The last three chapters presented methods to retrieve and communicate information about the consumed part of the input sentence. The next two chapters present methods to handle the not consumed part of the input string.

Chapter 6

Halting after an error

6.1 Current implementation and proposed improvement.

After an error is detected, and a report is created, a decision must be made on how to go on. Apart from halting after the first error, this could be a correcting or a non-correcting approach. [DEGANO1995] Contrary to literature in this field I think halting should be looked in to as a serious option.

In the literature about LR(k) parsing, halting after the first error detected is simply considered not done. In fact most literature about error handling concentrates on algorithms that detect as many errors as possible. This is partly because panic mode recovery was used rather soon after the introduction of LR parsing. Much research on error handling was aimed at improving this technique.

Furthermore in those days compiling, of which parsing was the first step could be a time-consuming operation. It would be very disappointing, if, after a couple hours, the whole process halted on one forgotten semicolon. And finding the second missing semicolon took another two hours, redoing most of the work already done the first time. Nowadays parsing hardly leaves you time to take a sip of your coffee. Therefore the time does not play an important part anymore.

Being a highly generic tool there exists no panic mode recovery technique for SGLR, because panic mode recovery uses certain language dependent synchronization tokens to provide a starting point and parser state to resume parsing. Therefore halting gracefully should be considered as an option.

The current implementation just halts and returns an error report. In chapter 3 it is proposed to return the structure of the consumed part of the input sentence in the form of a partial parse tree. This tree will only reflect the parsed prefix of the input sentence.

The next section will present an algorithm to extend that partial tree. The resulting tree reflects the whole of the input sentence, with a clear distinction between the parsed prefix and the not-parsed suffix. Unparsing this tree will produce the original input sentence.

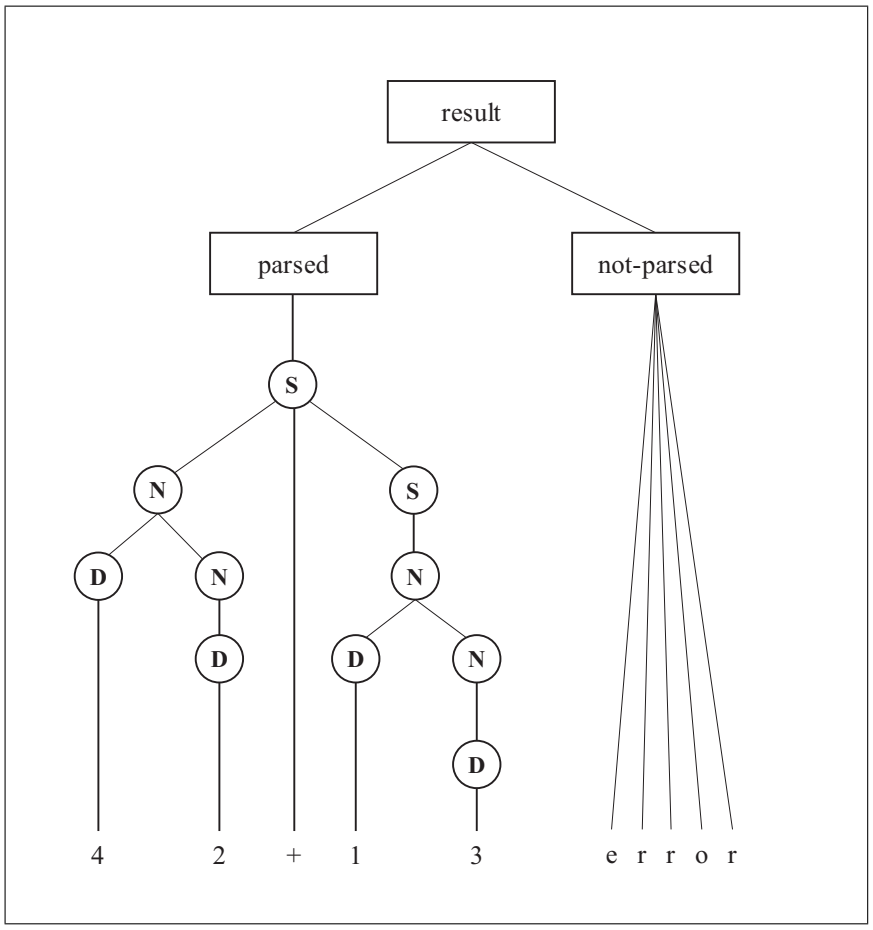

Figure 6.1: Completed result tree with not parsed part.

6.2 Algorithm

Processing the not-parsed suffix is pretty straight forward, All remaining tokens from the input string are shifted to the remaining stack of the partial tree, while counting the number of actual shifts. If eos is reached, then two reduce actions are performed, the first with a length of the number of shifts and the second with length two. The resulting stack is marked as the accepting stack. The resulting parse tree will look like figure 6.1. Figure 6.2 shows pseudo code for this algorithm.

6.2.1 Implementation

The halting procedure has been fully implemented during the course of this project according to the method described above. Halting is executed when an error is encountered while the predefined maximum number of errors has been found. The maximum number of errors is connected with the non-correcting recovery method that is described in chapter 7. If the maximum number of errors is set to one, then parsing halts after the first error as described above.

```
define RESULTPROD := "<result>"
define NOTPARSEDPROD := "<not-parsed>"

processNotParsedPart(){
  Token currentToken := next token from input string
  StackNode target := last new stack from reduceParsedPart()
  int numberOfShifts := 0

  while currentToken != eos do
    StackNode newStack := new stacknode with state ERRORGOTOSTATE
    create edge from newStack to target holding "currentToken"
    numberOfShifts++
    currentToken:= next token from input string
    target := newStack

  StackNode active reduceNotParsed(target, numberOfShifts)
  reduceResult(active)
}

StackNode reduceNotParsed(StackNode source, Integer length){
  TreeSet kids := the trees of the edges that form the path with
                              length length from source
  StackNode newStack := new stacknode with state ERRORGOTOSTATE
  reducer (head of path, newStack, kids, NOTPARSEDPROD)
}

reduceResult(StackNode source){
  TreeSet kids := the trees of the edges that form the path
                              from source to startnode
  StackNode newStack := new stacknode with state ERRORGOTOSTATE
  reducer (startnode, newStack, kids, RESULTPROD)

  mark newStack acceptingStack
}


reducer() see figure 3.2
```

Figure 6.2: Pseudo code for processing the not-parsed suffix.

6.3 Analysis

This section evaluates the quality of the halting procedure. From the list of requirements used in the previous chapters, source-based, unbiased, readable, and polite are not applicable for the halting procedure. The other requirements form the basis for this evaluation.

6.3.1 Correct

The halting procedure moves exactly the not-parsed suffix from the input sentence into the parse tree. If combined with the partial tree as created by the methods described in chapter 3, the result is an erroneous parse tree that exactly reflects the input sentence, and that can be unparsed to an exact copy of the original input sentence. It is still an erroneous parse tree however, because its structure does not comply to the grammar and its root is not the grammars start symbol either.

6.3.2 Precise

The halting procedure uses exactly the not-parsed suffix of the input string, no more no less.

6.3.3 Succinct

The halting procedure just identifies the not-parsed suffix of the input string and designates it as not parsed. No additional information is added.

6.3.4 Readable

In the Meta-Environment, after putting the cursor somewhere inside it, the not parsed part will be shaded yellow and named not-parsed. Also in the visualisation of the parse tree the not parsed suffix is clearly recognizable by a special node.

6.3.5 Efficiency

The halting procedure puts no additional strain on parsing error free sentences. The handling time is linear with the length of the not parsed suffix of the input sentence. However this time will always be less than the time needed for parsing when no error would have been encountered.

6.3.6 Flexibility

The halting procedure cannot be adapted in any way.

6.4 Conclusion

The halting procedure provides for an elegant way to halt parsing. It makes it possible to always create a partial tree for the not consumed part of the input sentence. It can therefore be used as a back up for other, correcting or non correcting error recovery schemes. In this sense it can be viewed of as the panic mode recovery for SGLR. Unlike panic mode recovery in LR(k) the halting procedure has no dependencies on any parts of the grammar. However it does not try to put much structure into the not parsed suffix either.

6.5 Summary

This chapter described the most basic way of error recovery that can be combined with returning a parse tree that reflects the whole of the input sentence, although this will be an erroneous parse tree. It boils down to halting the parse process after inserting the not parsed suffix of the input sentence into the parse tree. It does not try to make any corrections, nor does it supply any grammar defined structure for this suffix.

The next chapter describes a, also non-correcting, recovery scheme that does transform the not parsed suffix into a structure that is defined by the grammar.

Chapter 7

Continuing after an error as a substring parser

7.1 Current implementation and proposed improvement.

[REKERS1991] describes a derivation from the GLR algorithm for parsing substrings. It uses the same parse table and follows greatly the main GLR algorithm. This algorithm can be used to recognize if a sentence is a valid substring of the language at hand, and produce a parse tree if so.

The substring parse algorithm offers an opportunity to process the not parsed suffix into a structure defined by the grammar. Just like like the structure it would get when it was a suffix from a correct sentence of the language.

The next sections describe how Rekers' algorithm can be adapted to do that.

It is possible that the error token cannot be a prefix for a valid substring of the language. If that is the case substring parsing should start with the first token that is a prefix for a valid substring of the language. Also substring parsing should not start with layout characters, because usually layout characters will form a prefix for every substring of the language and therefore they won't limit the number of possible continuations. Also chances are that the first non layout character does not form a prefix for a valid substring of the language. In this sense layout characters should not be read as the usual row of tab, space, newline, etc, but as layout characters like they are defined by the grammar.

Two facts diminish the usefulness of this method: 1) subsequent errors may be concealed and 2) the amount of possibilities can easily explode.

Concealed errors As already mentioned by [RICHTER1985], with a non correcting error recovery method an error can mask subsequent errors. For instance `if exp)) {` contains two errors: the opening parenthesis after `if` is missing and the expression either lacks an opening parenthesis or has a closing parenthesis too many. Because there exists a valid prefix, `exp)) {` will be accepted as a valid substring, where `if (exp)) {` will produce an error on the second closing parenthesis.

In SGLR an error is only detected when there are no nodes in GSS to shift the current token to. Substring parsing starts off with an active stack for every possible shift action of the error token. This can introduce active stacks that accept almost every token repeatedly, and thereby mask all possible errors. For instance existence of multiple line comments can conceal all errors until an end-of-comment sign is found. Or the string literal conceals all errors until the end of the line.

A possible solution to this problem is to exclude certain productions from being a not completed result from substring parsing. A not completed result from substring parsing is illustrated by the following example: The substring "(exp)" will be interpreted by the production rule `"(" exp ")" -> exp`. Also the substrings "exp)" and ")" will be interpreted by the same rule, but those interpretations will not be complete, because one respectively two parts of the left hand side of the production rule are missing.

A (simplified) production rule for comments is: `"/*" [\0-\41\43-\255]* "*/" -> comment`. If such a production is not allowed to be a not complete result from substring parsing, then also in substrings a comment could only start with a start-of-comment sign. This can be achieved by annotation of the productions concerned.

Information explosion Any prefix from the string starting with the error token could be a valid suffix for a number of productions. Substring parsing finds all those possible interpretations. This increases the number of trees in the parse forest. Most of these trees won't be filtered out, but will appear in the parse tree as ambiguities.

A way to prune this parse forest is to prefer the tree with the lowest level of incompleteness. The level of incompleteness could be determined by annotating the not completed results from substring parsing with the number of parts in the left hand side they are missing. In the above example one and two for substrings starting with "exp)" and ")" respectively. These results are nodes in the resulting parse tree. The total of the annotations for a given (sub) tree amounts to the level of incompleteness.

Another issue is that substring parsing will be more time consuming then ordinary parsing, especially at the beginning of the substring. [REKERS1991] This is caused by the necessity to try multiple interpretations, adding an active stack for each of them. As Rekers calls it, substring parsing needs some time to get on its way. If there exists multiple errors close together, then total parsing time may grow disproportional. Therefore the number of errors found should be limited to a predefined maximum.

7.2 Algorithm

According to Rekers' algorithm, an active stack must be created in GSS for each possible shift action for the first token of the substring. For our purpose this will be the error token.

```
define SUBSTRINGSTARTSTATE := -3
initialiseSubstring(){
  IntegerSet shiftGotos:= lookupShiftGotos(currentToken)
  while (shiftGotos == {} or currentToken ∈ LayoutCharacters)
         and currentToken != eos do
    StackNode newStack := new stacknode with state ERRORGOTOSTATE
    create edge from newStack to target holding "currentToken"
    numberOfShifts++
    currentToken:= next token from input string
    target := newStack
    shiftGotos:= lookupShiftGotos(currentToken)
  reduceNotParsed(target, numberOfShifts)

  shiftQueue:= {}
  StackNode substringStartnode:= new StackNode with state SUBSTRINGSTARTSTATE
  for each Integer gotoState ∈ shiftGotos do
    shiftQueue := shiftQueue ∪ {⟨substringStartnode, gotoState⟩}
}
IntegerSet lookupShiftGotos(Token token){
  retrieve from the parse table a set that contains the gotoState
  for every shift action tokencan induce irrespective of the parser
  state.
}
reduceNotParsed() see figure 7.1
```

Figure 7.1: Pseudo code for initialising GSS at the start of substring parsing

```
handleNotCompleteProduction(Production prod, ReductionPath path){
  annotate prod with length of path - intended length of path
  IntegerSet gotos := lookupAllGotos(prod)
  For each Integer gotoState ∈ gotos do
    StackNode newStack := new StackNode with state gotoState
    kids := the trees of the edges that form path
    reducer(head of path, newStack, kids, prod)
}
IntegerSet lookupAllGotos(Production prod){
  retrieve from the parse table a set that contains all
  gotoStates for prod irrespective of the parser state.
}
reducer() see figure 3.2
```

Figure 7.2: Pseudo code for handleNotComplete(). This function is called each time the head of the path is the substring startnode.

This is done by creating a node, the substring startnode, which will function as the bottom of the stack. Instead of the normal startstate, this node will hold an auxiliary substring start state. This new bottom of the stack will be the target for all shift actions induced by the error token. Or more precisely the first token from the unparsed suffix that is not a layout character and that can induce shift actions. For this purpose the substring startnode is added to the shift queue once for each possible shift action of the error token. Normal parsing can then be resumed, with special care for reduce actions that try to reduce on to or beyond the substring startnode.

When, while substring parsing, a reduce action has a length that would require a target node that lies before the substring startnode then for this interpretation of the substring a completion is needed. Parsing could be resumed, as described by Rekers, by using the substring startnode as a target for all possible goto's for the given reduce action. Rekers suggests providing a possible completion. I suggest leaving it to the user how the part before the error could provide a completion. And only mark the production as not-complete.http://www.windguru.cz/int/index.php?sc=48299

If a reduce action has the substring startnode as its target node then no completion is needed, but the goto could not be determined, because the proper target state is not known. Therefore in this case also all possible goto's should be used.

[REKERS1991] suggests a method for completion of a substring when end-of-string is reached without an accepting state. For this purpose this could just be handled as another error.

7.2.1 Implementation

The existing functions are altered such that they can handle substring parsing also. This mainly boils down to a check whether the target node for reductions

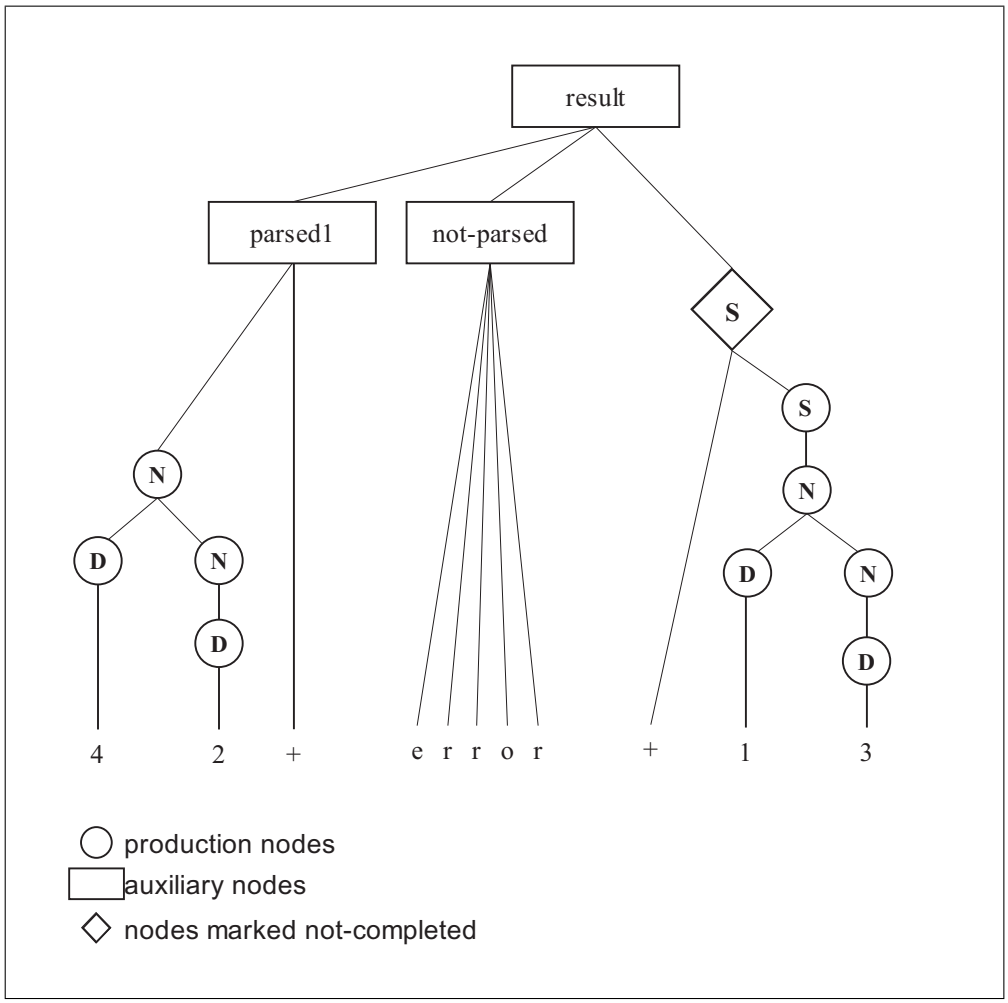

Figure 7.3: Result tree for the sentence "42+error+13" after proceeding with substring parsing

holds the special substring start state. That is where the main difference with the common parse process is. Also the information about the reduction path is augmented with the number of steps that could not be fulfilled.

As with the partial tree, no alterations are made to UPTR. Instead of special nodes for not-completed productions, extra ordinary production nodes are inserted that designate the tree they hold as not-completed and indicate the level of incompleteness.

Leaving pruning of the parse forest, by means of prefer lowest level of incompleteness, entirely to the filter process, proved to be to time consuming. Some measures are taken to reduce the tension on the filtering process.

Firstly, a tree is not added to the ambiguity table when its level of incompleteness is higher then the corresponding tree in GSS.

Secondly, not-completed production nodes propagate their level of incompleteness to their parents. This makes it possible to compare the levels of two trees without a complete search of the trees.

No provisions exist to read trees in the ambiguity table during parsing. Therefore, to be able to correctly determine the level of incompleteness for a new tree, it is essential that there exist no trees in the ambiguity table with a lower level than the corresponding tree in GSS. This means that an edge of GSS is replaced with another edge if necessary.

No trees are removed from the ambiguity table during parsing. Those trees already in the ambiguity table with an higher level of incompleteness than the tree in GSS are dealt with by the filter process.

In section 7.1 it is suggested to annotate certain productions such that it becomes impossible that they form incomplete productions for substrings. This is not implemented. The intended behaviour is simulated by the following process. Substring parsing starts with looking up all shift actions for the first token of the substring. Instead of using all those shift actions, actions that are also triggered by a layout character, are excluded. Layout characters are those characters that appear in the left hand side of productions that have as their right hand side a symbol annotated as layout. These exclusions have the effect that language constructions that accept also layout characters, like comments or string literals are excluded.

7.3 Analysis

This section evaluates the quality of this non-correcting error recovery by means of substring parsing. It does this on the basis of the same list of requirements as for the partial tree described in chapter 7.1. But with substring parsing there is one extra issue that has to be taken into account, namely its influence on the error reporting techniques from the previous chapters.

Discussions in those chapters assumed that the parsed prefix was parsed with the original SGLR algorithm. When the parsed prefix is parsed with the substring parse algorithm then handling time or result for the three error reporting techniques may differ.

Substring parsing starts with a surplus of stack nodes in GSS, each representing a possible prefix completion for the substring. As parsing progresses the number

of these possibilities gradually decreases, but boost up each time a reduction onto or beyond the substring startnode is made. When, while substring parsing, an error is encountered, then these extra stack nodes influence reporting of the error. This influence is greater when an error is encountered rather quickly after starting substring parsing. This shows in two ways.

The GSS parser state could be altered such that the message can not be translated as it would be while normal parsing. An example for this is shown in figure 8.4. Secondly, computing time for both the partial tree as the expected symbol list starts of with set A (see figure 3.3) and could therefore be increased considerably as the extra stack nodes are also present in this set. An example for this is shown in table 8.2 with parsing the C example with all options on.

7.3.1 Correct

Let x, y, and z be strings of non-terminals of a grammar G. If from a string xyz y is accepted as a substring of a language L defined by G. In other words substring parsing starts with the first token of y and halts after the last token of y, either with an error or with an accepting stack. Then in such a case it is certain that there exists x and z also substrings of L, such that xyz is a sentence of L.

In this sense substring parsing is correct, an accepted substring is always a substring from some sentence of the language.

It is not certain however that $x'y$ is a prefix for some sentence of L, when x' extended with some prefix from y is a prefix for some sentence of L. This means that possibly an error gets concealed, and it looks like substring parsing incorrectly accepted y as a substring. This is a known issue with non corrective error recovery schemes[RICHTER1985].

It is also not certain that when substring parsing of y halts with an error, no x' exists such that $x'yz$ is a sentence of the language. This means that a spurious error can be introduced, in that case an error is detected that is not really there.

7.3.2 Precise

The structure generated by substring parsing is based on the whole of the substring starting with the error token. No measures are taken to communicate which part of the input string does not contribute to the error.

7.3.3 Succinct

As with the partial tree from chapter 3, substring parsing cannot give the ultimate structure for the substring starting with the error token. The most useful interpretation will show where and how the intended interpretation deviates from the found interpretation(s). The number of possible interpretations determines how the most useful interpretation is blurred by the less useful interpretations. The number of possibilities can even be disproportional; therefore it is necessary to limit them. The proposed method for this limitation, preferring the interpretation with the lowest level of incompleteness, does not guarantee that the most useful interpretation will not be removed.

7.3.4 Source-based

The result of substring parsing is entirely based on the grammar and the input sentence.

7.3.5 Unbiased

The method, to prefer the least level of incompleteness while substring parsing, emphasizes the part to the right of the error token.

7.3.6 Readable

The analysis for the readability of the partial tree in chapter 3 applies just as much to the results of substring parsing. The Meta-Environment has various ways to show the structure of a parsed sentence, such as highlighting, pretty printing, showing the right-hand-side of a selected production, and visualisation of the parse tree. Except from pretty printing, these can all be used for erroneous parse trees.

However multiple interpretations introduced by substring parsing are shown as ambiguities in the resulting parse tree.

Unfortunately, the Meta-Environment offers no easy way for discovering differences between two or more interpretations of (a part of) the parsed sentence.

7.3.7 Efficiency

Substring parsing does not strain normal parsing of error free sentences.

Substring parsing depends on the length of the remaining input string. The extra time will mostly be used at the beginning of the substring. If multiple errors exist close together, then total parsing time can grow disproportional. Therefore a maximum is set for the number of errors found.

7.3.8 Flexibility

For this non-correcting recovery method the amount of errors found before parsing halts, could be used to fine tune the behaviour.

7.4 Conclusion

This non-correcting recovery strategy, which uses substring parsing following the algorithm devised by [REKERS1991], works well for some grammars. But it is useless for grammars with productions that accept almost any character like string literals or comments. To overcome this inadequacy certain productions need to be annotated such that they can not be used as an incomplete production while substring parsing. This means alterations to the grammar. Although the heuristic chosen for the implementation, exclusion of certain productions based on layout characters, might offer a reasonable alternative.

Also, the choice to prefer interpretations based on the level of incompleteness is arbitrary; as it is not certain that the interpretation with the lowest level will be the most informative.

Substring parsing as a means of non-correcting error recovering has as it main advantage that it gives an interpretation of the part of the input sentence to the right of the error. This is of course as useful as the interpretation of the part to the left. However it can strain parsing time of erroneous sentences considerably.

7.5 Summary

This chapter presented a non-correcting error recovery strategy. This strategy gives an interpretation for the part of the input sentence to the right of the error. For that purpose an adaptation of the substring parsing algorithm described by [REKERS1991] is used. The adaptations include that no attempts are made to provide extensions to the left or the right of the substring.

Furthermore the number of possible interpretations is limited by preferring interpretations that need the least grammar constructs, terminals or non-terminals from the left of the error.

Also certain productions are excluded from appearing in the structure for the substring when they need certain terminals or non-terminals from the left of the error.

This chapter concludes the descriptions of proposed strategies for error reporting and error recovery. Next chapter describes how these strategies work together. Next chapter also presents some practical examples.

Chapter 8

Experimental results

This chapter describes the results of the implementation of the techniques described in the previous chapters. The first section provides some performance and quantity measurements about the implementation.

The second section gives some practical examples in the form of screenshots of the Meta-Environment. They explain how the Meta-Environment uses the newly derived information to inform the user.

Most of this thesis assumes that the five proposed methods for error handling are separated processes. But of course it is intended that they work together. Also they influence each other. Especially substring parsing, used for the non-correcting recovery technique, may influence the others considerably.

All techniques have been implemented together. When an error is encountered, firstly the lists of expected characters and symbols are created, and then a message is composed consisting of those lists and the GSS parser state. Thirdly a partial tree is created. After that, parsing commences with either the halting procedure or substring parsing. The choice for one technique or the other depends on whether the number of errors found has reached the pre defined maximum.

Finally all intermediate results like partial trees or not parsed parts are joined together to one final erroneous parse tree. This tree is erroneous, because it does not comply with the structure defined by the grammar. However this tree can be used to visualize the structure of the parsed parts of the input sentence.

8.1 Measurements

8.1.1 Performance

Next sections indicate parsing times for both syntax error free input sentences and input sentences containing syntax errors. These times should not be viewed as absolute, because they will be greatly dependent of the hardware that is used. All measurements were done on a machine with the following specifications:

Processor: Intel (R) Pentium (R) mobile M 1.73 GHz

Memory: 1.5 GB

Operating system: Ubuntu 7.04

	file size		current SGLR	new SGLR			
			reference	partialtree	+ parserstate[1]	+ substring[1]	
	LOC	kB	sec.	sec.	sec	sec	%
ATerm.java	7823	232	2.22	2.55	2.54	2.52	14
SDF10	4054	130	1.01	1.12	1.13	1.13	12
SDF100	40504	1300	9.74	11.27	11.18	11.16	15
SDF300	121502	3901	29.57	33.60	33.74	33.77	13
SDF500	202501	6498	49.26	56.00	56.37	55.93	14
SDF600	243000	7803	59.45	67.39	67.89	67.47	14
SDF1000	404998	13005	99	113.04	113.08	112.07	14
SDF1500	607495	19507	149.28	169.53	170.04	168.95	14

[1] this technique in addition to the technique in the previous column

Table 8.1: parsing time for large error-free sentences. Column % gives additional parsing time as a percentage of the reference time

8.1.1.1 Error free parsing

To be able to check the performance of (different versions of) SGLR, Rob Economopoulos has composed a set of very large input sentences. A concatenation into one file of the ATerm java implementation and a series of concatenations of the SDF definitions. These are used to measure how much parsing time increases due to new error handling routines. Table 8.1 shows the result of these measurements. Column % shows additional parsing time as a percentage of the reference time. Times are measured by means of the Linux time utility.

Computing the list of selected symbols does not require additional measures above those for the partial tree. Time for the GSS parser state is measured with both partial tree and parser state active, time for substring parsing includes all other approaches. As a reference parsing times for the current implementation of SGLR are used.

Main conclusion from table 8.1 is that keeping the parser state and preparations for substring parsing does not seem to add much overhead.

Most extra time is caused by measures to make partial tree creation possible. This is probably due to the changed implementation of GSS with build-in garbage collection.

Probably the execution time of this new implementation can be optimized. A rough remedy would be to separate the code for error handling from the normal parsing code completely. That way error handling routines cannot put additional time strains on error free parsing, at the cost of duplicating code.

8.1.1.2 Error handling

Table 8.2 shows some measurements done while parsing the examples from section 8.2. These are total parsing times measured with the Linux time utility. These are rather short time intervals, and they should not be viewed as absolute. In fact these figures reflect an average of several measurements.

Although these measurements do not cover everything possible, some trends could be derived from them.

		halting		substring	
		-exp[1]	+exp[2]	-exp[1]	+exp[2]
pico example	error-free	0.028	0.028	0.028	0.028
	all	n/a	n/a	0.048	0.058
	first	0.028	0.028	0.36	0.038
	second	0.032	0.044	0.38	0.044
	third	0.028	0.044	0.36	0.044
C example	error-free	0.108	0.108	0.108	0.108
	all	n/a	n/a	1.564	5.396
	first	0.104	0.120	0.188	0.204
	second	0.109	0.125	0.280	0.324

[1] computing of expected symbols excluded

[2] computing of expected symbols included

Table 8.2: Errorhandling, parsing times in seconds

8.1.2 Lines of code

To the original 4300 lines of C code which formed SGLR, 1200 were added to implement the error handling routines discussed in this thesis. The original error handling routine was made up of 20 lines of code.

The implementation of SGLR is modularized. The main modules are: parser, gss, parseTable, and parseForest. Module parseForest processes the parse forest, which is produced by module parser into a parse tree. Actually when an accepting state is reached, GSS won't hold a forest, but a tree with exactly the input string as its leaves. Whenever an attempt is made to connect two nodes of GSS with a second edge, then instead of inserting this edge in GSS, the tree represented by the new edge is stored in the ambiguity table. The filter process retrieves all trees from the ambiguity table, and either discards them or inserts them in the parse tree.

Table 8.3 shows the division of lines of code over these modules.

module	current implementation	added	new implementation
sglr	906	134	1040
sglr/parser	4?14	632	1046
sglr/gss	612	359	971
sglr/parseForest	1189	5	1194
sglr/parseTable	892	136	1028
sglr/utils	287	34	321

Table 8.3: physical lines of C code in SGLR (counted with SLOCCOUNT by David A. Wheeler)

8.2 Examples

This section contains some screen shots of the Meta-Environment. They illustrate how the information, which is revealed by means of the methods discussed in this thesis, can be used to help the user to discover the nature of the syntax errors he made.

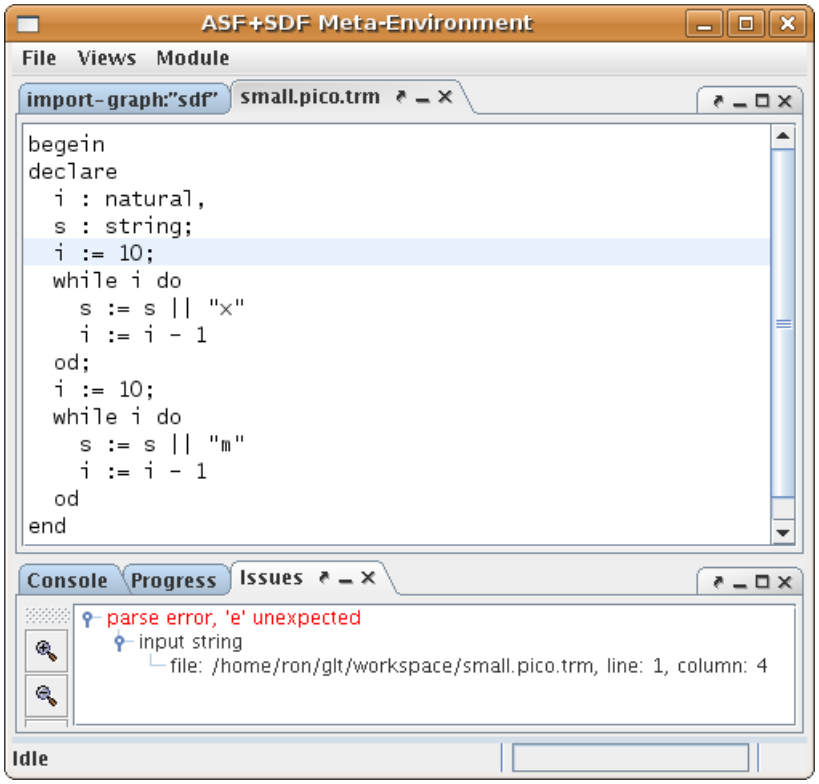

Figure 8.1: Example of a small program written in the Pico example language. It contains three syntax errors. It is shown after parsing with the current version of SGLR.

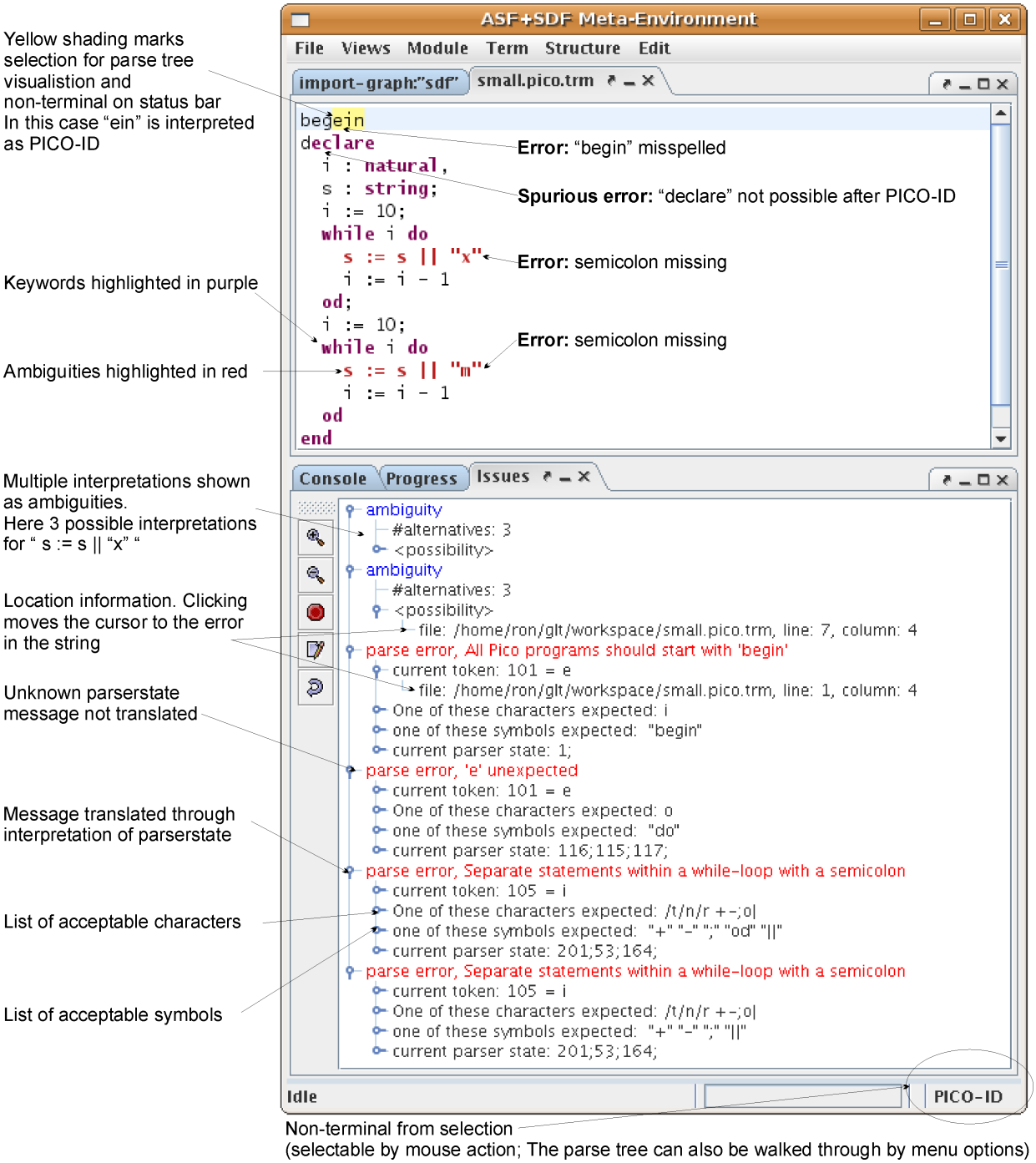

Figure 8.2: The same Pico program as figure 8.1, but shown after parsing with the improved version of SGLR. Four instead of three syntax errors are found. Recovery from the first error introduces a spurious error.

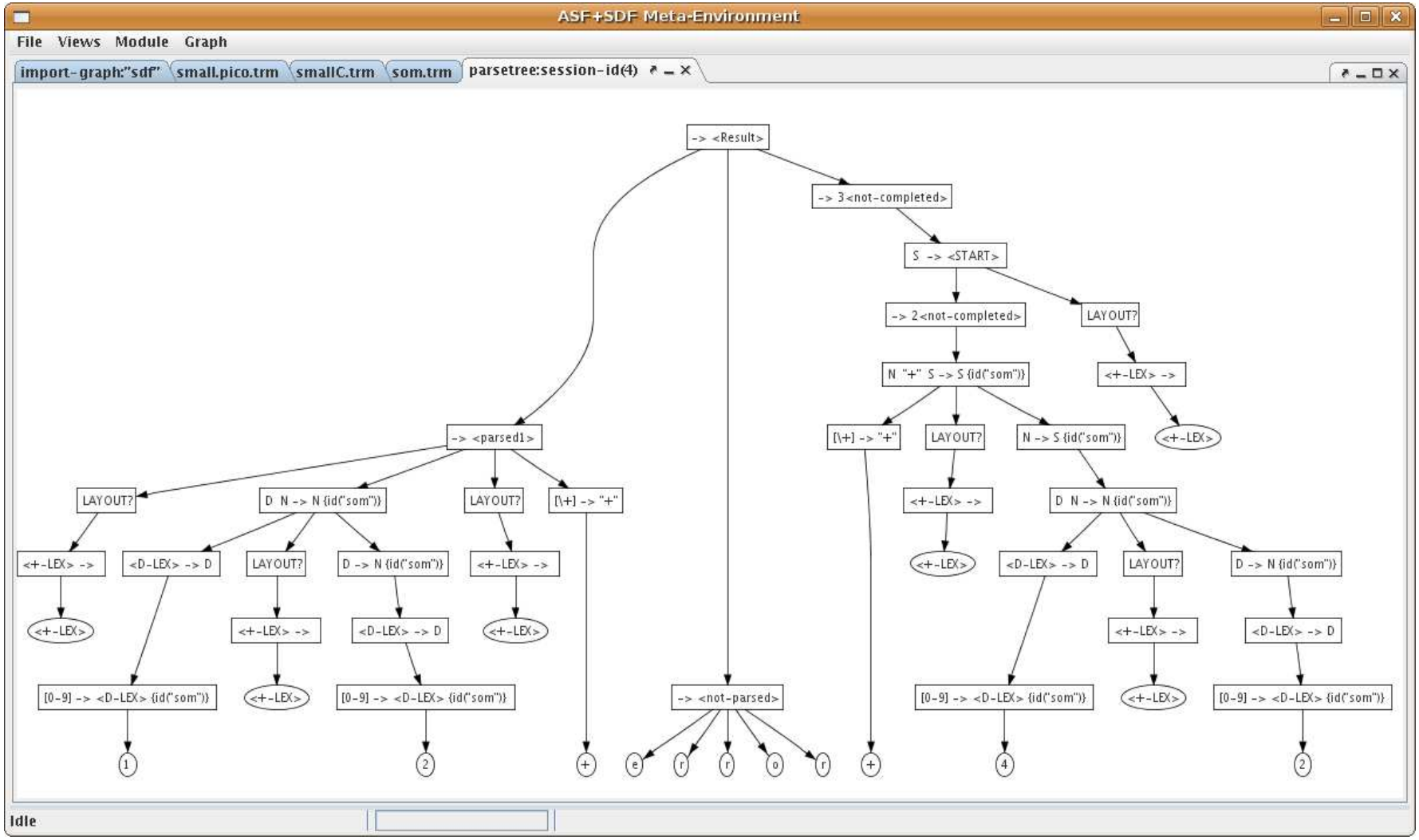

Figure 8.3: Visualization of the parse tree as produced for the sentence "12+error+42" with the `Sum` example grammar described in section 2.1. First part of the result tree is derived by normal parsing, second part cannot be interpreteted, and third part is derived by substring parsing.

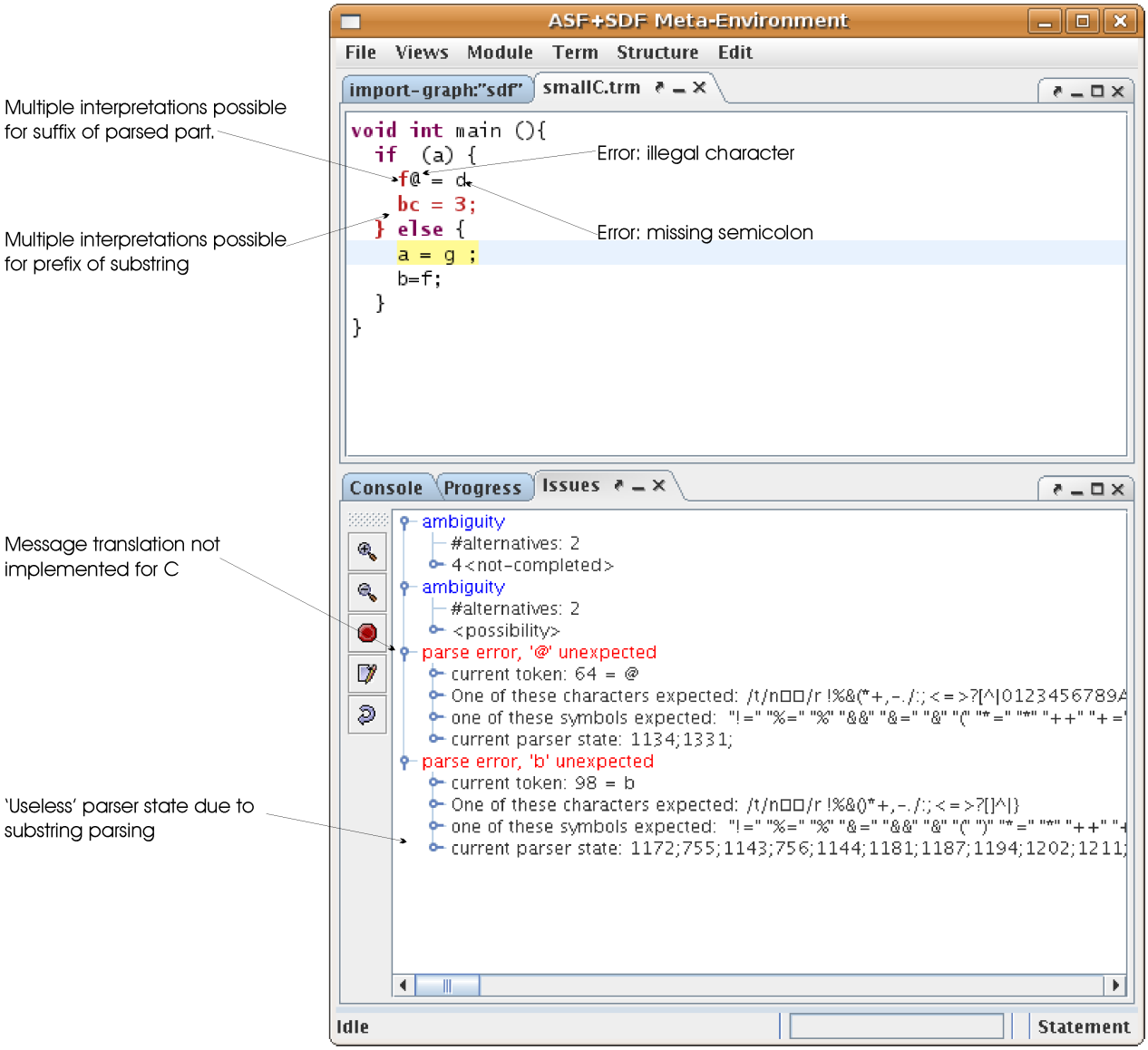

Figure 8.4: C fragment containing two syntax errors. Among other things this shows that the list of expected symbols can grow considerably, here 41 and 40 respectively. And that substring parsing may obscure the parser state when errors are close together. Next figures show trees of the interpretations directly before, in between and directly after the errors.

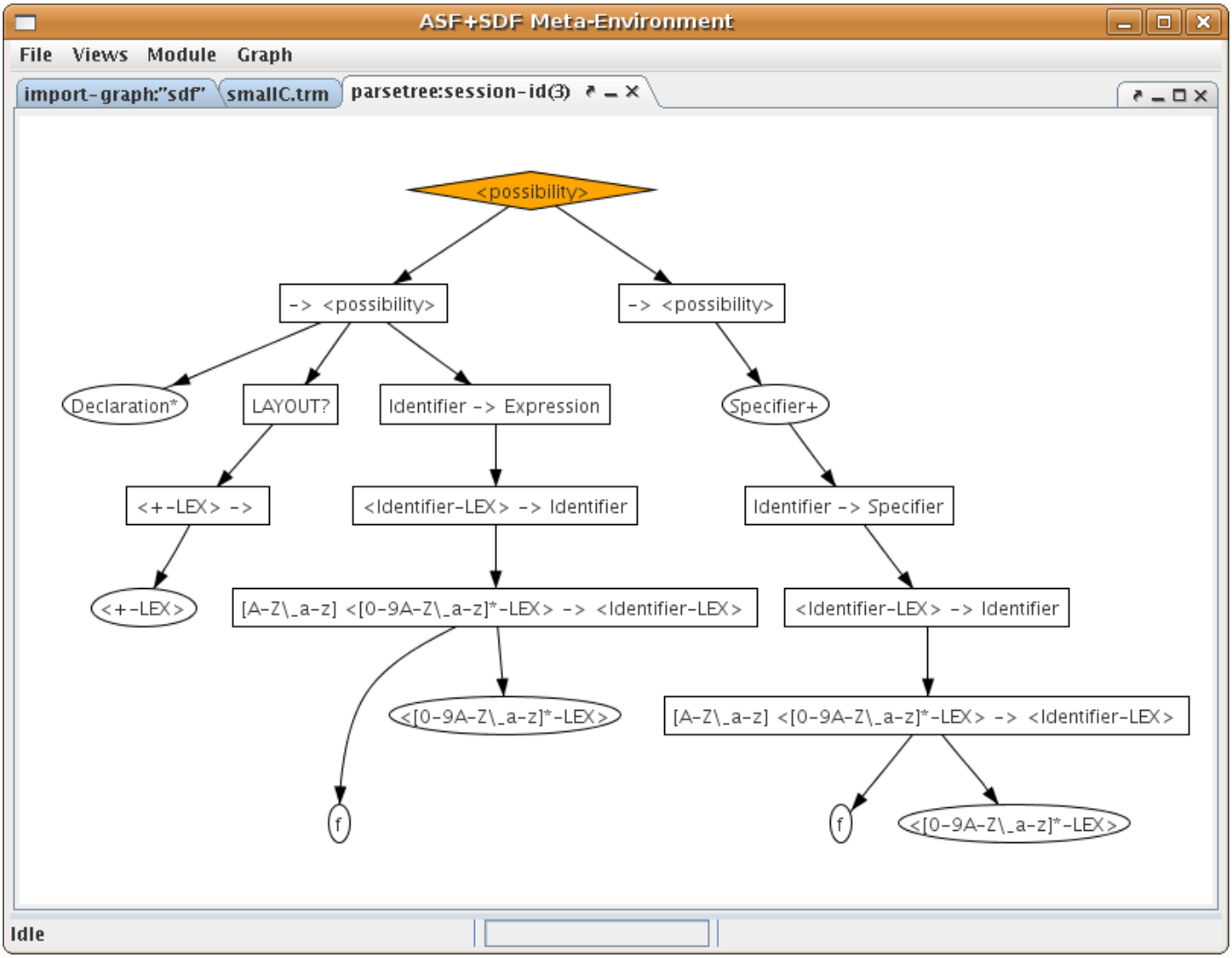

Figure 8.5: tree showing the two possible interpretations for "`f`" in the fragment of figure 8.4

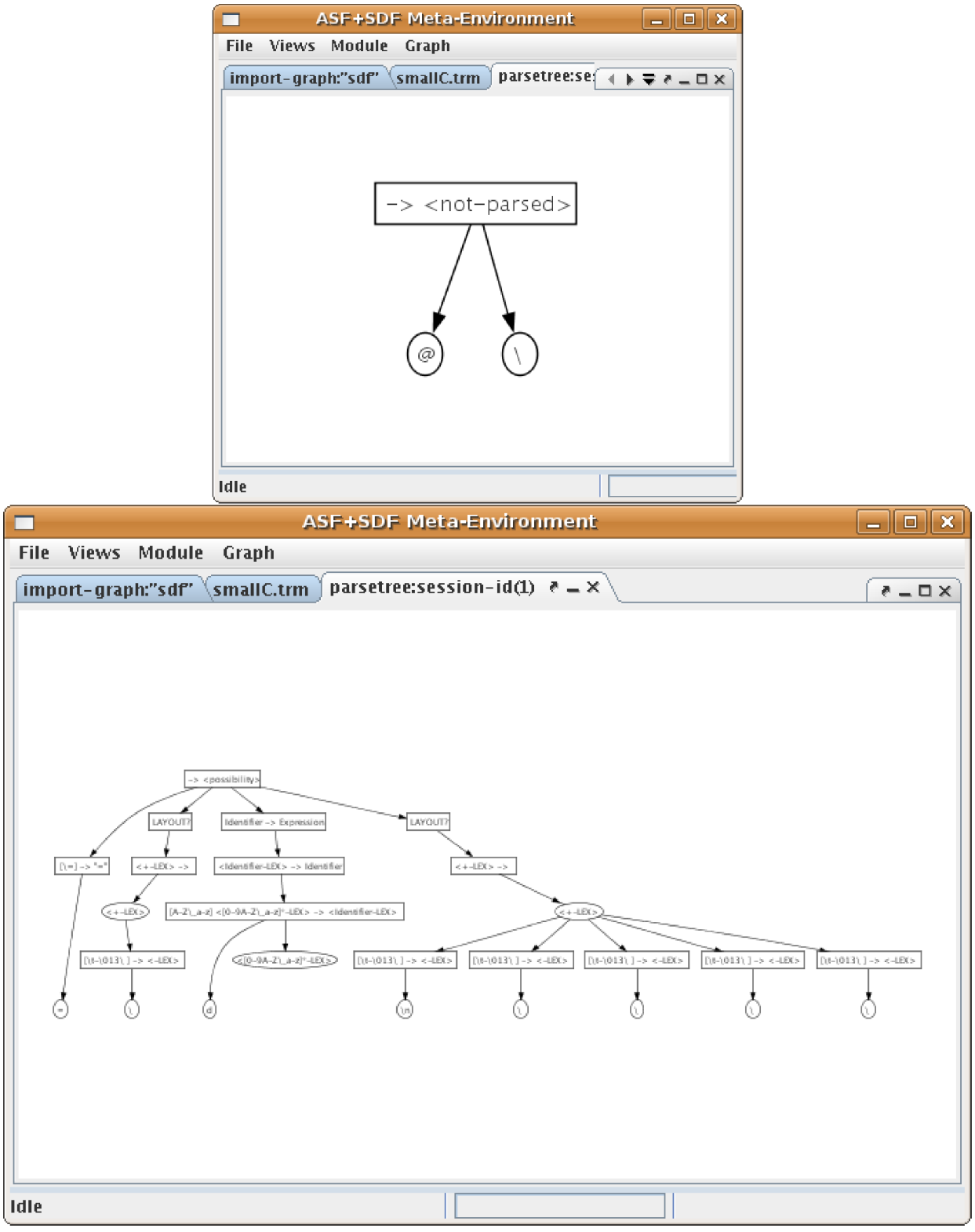

Figure 8.6: trees for "@ = d" in the fragment of figure 8.4

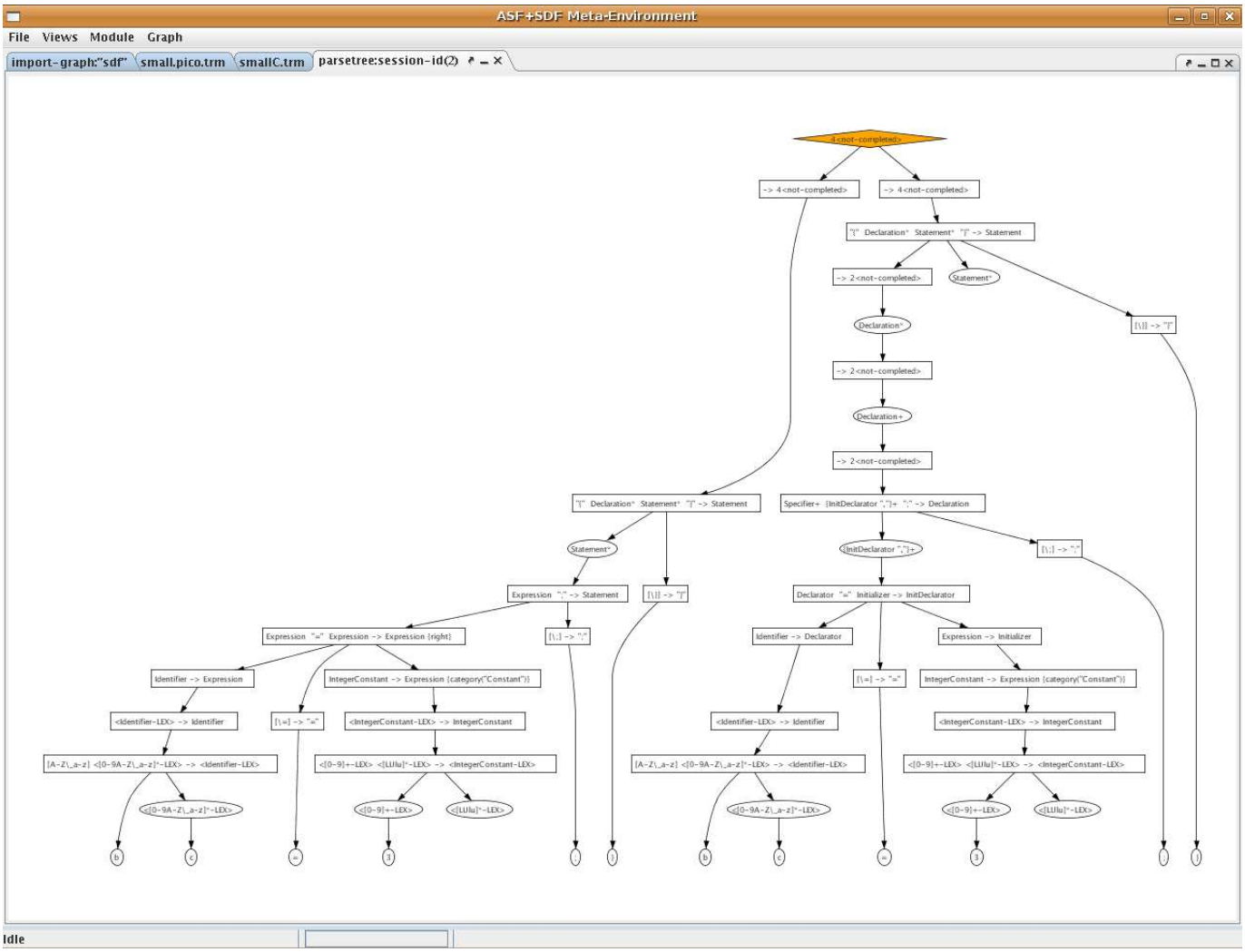

Figure 8.7: Tree for "`bc = 3; }`" in the fragment of figure 8.4, not showing layout. It shows two possible interpretations that are found by substring parsing (chapter 7). Both options needed four non-terminals to their left-hand-side to complete, and therefore they are both shown. Orange coloured diamond-shaped nodes indicate special inserted ambiguity nodes. This tree has been scaled to fit on the page. In the Meta-Environment, those trees are both scalable and scrollable.

Chapter 9

Conclusions and future work

9.1 Conclusions

This thesis has introduced five techniques to improve the error reports of the SGLR parsing algorithm. The improvements aim to help the user in finding a repair such that the erroneous input sentence becomes acceptable for the grammar, and that it reflects the interpretation that the user intended.

To be able to asses the proposed improvements for their usefulness, a list of requirements has been made up. This list has been put together by requirements from literature about compilers and LR parsers. A technique that scores well for all those requirements will be a useful one. This is not the case for any of the proposed techniques, but that does not mean that they are useless. The list of requirements helps to identify the weak and strong points of each technique. Table 9.1 summarizes the results of these analyses for each technique.

A starting point for this thesis was that the error handling should not depend on specific production rules of the grammar. None of the proposed techniques do rely on such productions. However a grammar can contain production rules that make substring parsing useless, therefore substring parsing is not fully grammar independent.

For the language specific error messages a grammar specific table is needed. This table is used to translate parser states into messages. Such a table is not a part of the grammar, but an addition to the grammar. It is doubtful whether the improvements in readability compensate for the extra efforts needed to create such a table.

A weakness with all techniques is that they are biased on either the prefix ending at the point of error detection or on the suffix starting at the point of error detection. This makes them less useful in situations where the error in the intended interpretation gets concealed by other possible interpretations. This is typical for SGLR, which continues parsing until no more possible interpretations are left.

The only technique that offers a solution in those situations is the partial tree created with the omniscient algorithm. But this algorithm is neither succinct nor efficient enough to be a feasible solution.

	Partial tree	Expected symbol list	Language specific messages	Halting	Substring parsing
Correct	++	+	−	++	±
Precise	+	+	−	++	±
Succinct	+[1]	±	++	+	−
Source-based	++	++	++	n/a	++
Unbiased	−	−−	−−	n/a	−
Readable	+	+	++	+	+
Polite	n/a	+	+	n/a	n/a
Efficiency	+[1]	+	+	+	−
Flexibility	+	±	++	±	±

Key: ++ very high; + high; ± intermediate; − low; −− very low;
[1]For the omniscient version of the algorithm, as described in section 3.2.2, this will be −−

Table 9.1: Summary of the analyses of the error handling techniques

In my opinion three of the five proposed techniques are good enough to be used by the Meta-Environment. These are the partial tree combined with the halting procedure and the list of expected symbols.

The list of expected symbols might offer possible solutions for users whom have little experience with the language. Its only disadvantage is a slightly increased error handling time.

The partial tree will in many cases reveal the structure of the input sentence up until the point of error detection. I think this will be much help in finding a repair for the error.

To commence parsing with substring parsing is only feasible when the user can adjust the maximum number of errors found. If this number is set to one, then substring parsing will be never invoked.

Lots of work will have to be done to arrive at a proper implementation for translating the parser state into messages. Presenting the parser state to the user without translation is useless. I can think of other more useful improvements to the Meta-Environment.

9.2 Future work

Three directions can be distinguished for future work on error handling of SGLR parsers: improvements to error detecting, error reporting, and error recovering.

9.2.1 Detecting

A problem with all techniques discussed in this thesis is that they don't work well when errors get concealed by other interpretations. Many substrings of a language will have more than one possible interpretation. It depends on the context which interpretation is chosen. If a user sees that an interpretation is chosen that was not the interpretation he intended, then mostly the user will know how to change the context to get the desired interpretation.

It is more difficult when an interpretation exists that is applicable for almost every substring of the language. In that case an error against the intended interpretation will get concealed, but also the context of the intended interpretation will get concealed. In that case the found interpretations in the partial tree won't offer much help in finding a repair for the error.

As a theoretical example, let `[\0-\9\11-\36\38-\255]* "%%" -> comment` be a production rule from some grammar. In that case any line that cannot be interpretated by other production rules of the grammar and does not end with `%%` will cause an error. However, the partial tree that gets created at that point will only show the characterclass `[\0-\9\11-\36\38-\255]*` as a possible interpretation for that line. With that all other possible interpretations, which might have been tried during parsing get concealed.

Unfortunately the algebraic specification formalism (ASF), which the Meta-Environment uses for code analysis and code transformations, is a language whose grammar contains such constructs. The basis of the grammar with which a certain ASF-module will be parsed, is the grammar for the language on which the analyses are done. These problems for ASF are caused by the same language constructs that causes the problems with concealed errors while substring parsing. Probably there is no solution for this without alterations to the grammar.

9.2.2 Reporting

It is inevitable that erroneous result trees contain multiple interpretations for the consumed part of the input sentence. These are shown as ambiguities in the Meta-Environment. Enabling quick comparisons, pinpointing the differences, of these ambiguities would mean a big improvement. Not only for end users debugging a program, but also for language developers trying to disambiguate their grammar.

9.2.3 Recovering

A way to improve the recovering scheme for grammars that allow substring parsing (if necessary with annotated productions), is to complete the not completed productions with (sub) trees from the parse stacks from before the point of error. This will probably mean that creation of a partial tree as proposed in this thesis will have to wait until the rest of the input sentence is parsed. This partial tree would then be created from the stacks not used by the substring parser.

The partial tree would then reflect a suffix from the parsed part of the input sentence instead of the whole parsed part. The prefix of the parsed part will be reflected by the resulting parse tree from substring parsing. From this resulting tree, some sub tree(s) might still be missing.

We then have two trees, and from that, it should be possible to compose messages like: "found this tree, with at its root <non-terminalA>, for substring "error", but expected a tree with root <non-terminalB> in that place".

I do not know whether an algorithm to do this will be both time and memory efficient enough to be useful, but I do know that all necessary information is there.

Bibliography

[AHO1972] Aho, A. V., and Peterson, T. G, 1972. A minimum distance error-correcting parser for context free languages. Siam J. Comput. 1, 4 (Dec.), p305–312.

[ALLOY] Alloy Homepage, Retrieved August, 2007, from: http://alloy.mit.edu

[BERGSTRA1994] Bergstra, J.A., P. Klint. 1994, The Discrete Time ToolBus. Technical Report P9502, Programming Research Group, University of Amsterdam.

[BRAND2000] Brand, M.G.J. van den, H.A. de Jong, P. Klint, and P.A. Olivier, 2000. "Efficient Annotated Terms." Software – Practice & Experience 30, p259–291.

[BRAVENBOER2006] Bravenboer, M., E. Tanter, and E. Visser, 2006. Declarative, formal, and extensible syntax definition for aspectJ. In Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Portland, Oregon, USA, October 22 - 26). OOPSLA '06. ACM Press, New York, NY, p209–228

[DAIN1994] Dain, J. A. 1994. A practical minimum distance method for syntax error handling. Comput. Lang. 20, 4 (Nov.), 239-252.

[DEREMER1971] DeRemer, F. L., 1971. Simple LR(k) grammars. Commun. ACM 14, 7 (July), 453-460.

[DEGANO1995] Degano, P., and C. Priami, 1995. Comparison of syntactic error handling in LR parsers. Software-Practice and Experience, 25(6) (June) p657–679.

[FTAT] Formele Talen en Automatentheorie / [cursusteam: F.J. Wester et. al.]. –Heerlen: Open Universiteit Nederland, 1990.

[GRAHAM1975] Graham, S. L. and Rhodes, S. P. 1975. Practical syntactic error recovery. Commun. ACM 18, 11 (Nov.), p639–650.

[HEERING1989] Heering, J., P.R. Hendriks, P. Klint, P., and J. Rekers, 1989. The syntax definition formalism SDF–reference manual–. SIGPLAN Not. 24, 11 (Nov.), p43–75.

[HORNING1974] Horning, J., 1974. What the compiler should tell the user. In Compiler Construction: an Advanced Course. Springer-Verlag, Berlin, Germany, p525–548.

[JEFFERY2003] Jeffery, C. L., 2003. Generating LR syntax error messages from examples. ACM Trans. Program. Lang. Syst. 25, 5 (Sept.), p631–640.

[METAWS] The Meta-Environment webhome, Retrieved May, 2007, from: http://Meta-Environment.org

[REKERS1991] Rekers, J., and W. Koorn, 1991. Substring parsing for arbitrary context-free grammars, ACM SIGPLAN Notices, volume 26, No. 5 (May), p59–66.

[REKERS1992] Rekers, J., 1992. Parser Generation for Interactive Environments . Ph.D. thesis, University of Amsterdam.

[RICHTER1985] Richter, H., 1985. Noncorrecting syntax error recovery. ACM Trans. Program. Lang. Syst. 7, 3 (July), p478–489.

[SIPPU1983] Sippu, S., and E. Soisalon-Soininen, 1983. A Syntax-Error-Handling Technique and Its Experimental Analysis. ACM Trans. Program. Lang. Syst. 5, 4 (Oct.), p656–679.

[TOMITA1987] Tomita, M., 1987, An efficient augmented-context-free parsing algorithm, Computational Linguistics, v.13 n.1-2 (January-June), p.31–46.

[VISSER1997] Visser, E., 1997. Scannerless generalized-LR parsing, Technical Report P9707, Programming Research Group, University of Amsterdam.

[YANG2000] Yang, J., G. Michaelson, P. Trinder, and J. B. Wells, 2000. Improved type error reporting. In M. Mohnen and P. Koopman, editors, IFL'00: Proceedings of the 12th International Workshop on Implementation of Functional Languages, volume 2011 of LNCS (Sept.), p71–86. RWTH Aachen, Springer Verlag.