INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

A Robust Parser for the Web Query Language Xcerpt

Clemens Ley

Projektarbeit im Rahmen des Fortgeschrittenenpraktikums

Beginn der Arbeit:	September 2004
Abgabe der Arbeit:	Juli 2005
Betreuer:	Prof. Dr. François Bry
	Dr. Sebastian Schaffert

Abstract

This project thesis investigates error handling for the Web Query Language Xcerpt (cf. `http://www.xcerpt.org`). In addition, a formal definition of the Xcerpt syntax is given.

After a short introduction to Xcerpt and parsing, several error management techniques known from literature are presented. They are compared regarding their usefulness for Xcerpt, and one technique is recommended as most suitable for Xcerpt.

Unfortunately, the technology to produce a fast and easy to maintain parser for Xcerpt allows only a very limited error management. We discuss some extensions of this technology to improve its error management potentials. We also discuss the error management, possible with the available technology, which has been implemented for the Xcerpt parser.

In the last section we present a new error handling technique weakening some problems of traditional error management techniques.

Zusammenfassung

In dieser Projektarbeit wird die Fehlerbehandlung für die Web Anfragesprache Xcerpt untersucht (s. `http://www.xcerpt.org`). Des Weiteren wird eine formale Definition der Syntax von Xcerpt angegeben.

Nach einer kurzen Einführung in Xcerpt und das Parsen, werden einige aus der Literatur bekannte Fehlerbehandlungs-Techniken vorgestellt. Diese werden bezüglich ihrer Brauchbarkeit für Xcerpt verglichen und eine davon für Xcerpt empfohlen.

Ungünstigerweise erlaubt die Technologie, mit der man einen schnellen und gut wartbaren Parser für Xcerpt herstellen kann, nur sehr eingeschränkte Fehlerbehandlung. Wir diskutieren einige Erweiterungen dieser Technologie, die ihr Fehlerbehandlungspotenzial verbessern. Des Weiteren präsentieren wir die Fehlerbehandlung, die mit dieser Technologie möglich ist, und die für Xcerpt implementiert wurde.

Im letzten Abschnitt diskutieren wir eine neue Fehlerbehandlungstechnik, mit der Probleme anderer Fehlerbehandlungstechniken abgeschwächt werden können.

Contents

1 Introduction

Every programmer knows the problem that sometimes it takes a long time to find and correct a few, simple syntactical errors. This is not only extremely annoying for the programmer, it is also very expensive from an economical point of view. Highly skilled and paid workers spend a non-negligible amount of their time searching for errors. Therefore good techniques for error management have not only the potential to please the programmer, but also to make software development cheaper.

Often, the decision whether to like a new programming language or not is very emotional. If a programmer is able to implement some small programs quickly and easily, it is probable that he will like the language. If he struggles hard to get a program running, he might turn away and look for another language. As Xcerpt is new, many people have not decided whether to like or not. A parser with good error handling can help to make Xcerpt more accessible.

The problem of error management is probably as old as higher programming languages. A lot of research has been done on this field. Much of this research dates back to the end of the seventies and eighties. Algorithms exist which are able to correct almost any syntax error, but the repair might not be so natural [MYV95, Cer02]. Others correct up to three quarters of errors, like a human reader would [BF87]. But none of these techniques seems to outperform all others. They all have their advantages and disadvantages. To sum it up, one could say that there is a trade off between speed, and quality of error management.

This project thesis investigates error handling for the Web Query Language Xcerpt. In the first part 'Preliminaries', it gives a short introduction to Xcerpt. This is followed by an introduction to *LR* and *LALR* parsing, the technique used to build the Xcerpt parser. To conclude the first part, some error handling techniques known from literature are discussed.

In the second part 'The Xcerpt Parser', first some issues about the Xcerpt grammar are discussed. Next, the technologies which are used for the Xcerpt parser are described. As the available technologies showed to be insufficient for the needs of the Xcerpt parser, they need to be extended. Possible extensions as well as the way they could be used to improve error handling for Xcerpt are described. In the following section 'The Error Handling' the error handling that was implemented with the available technologies is described. In the last section we present a new idea how some general problems of some error handling techniques can be remedied.

A grammar for the Xcerpt syntax can be found in appendix A.

Part I

Preliminaries

2 The Web Query Language Xcerpt

This Introduction to Xcerpt is taken from [SFB05].

An Xcerpt [Sch04] program consists of at least one *goal* and some (possibly zero) *rules*. Rules and goals contain query and construction patterns, called *terms*. Terms represent tree-like (or graph-like) structures. The children of a node may either be *ordered*, i.e. the order of occurrence is relevant (e.g. in an XML document representing a book), or *unordered*, i.e. the order of occurrence is irrelevant and may be chosen by the storage system (as is common in database systems). In the term syntax, an *ordered term specification* is denoted by square brackets `[ ]`, an *unordered term specification* by curly braces `{ }`.

Likewise, terms may use *partial term specifications* for representing incomplete query patterns and *total term specifications* for representing complete query patterns (or data items). A term t using a partial term specification for its subterms matches with all such terms that (1) contain matching subterms for all subterms of t and that (2) might contain further subterms without corresponding subterms in t. Partial term specification is denoted by *double* square brackets `[[ ]]` or curly braces `{{ }}`. In contrast, a term t using a total term specification does not match with terms that contain additional subterms without corresponding subterms in t. Total term specification is expressed using *single* square brackets `[ ]` or curly braces `{ }`. Matching is formally defined later in this article using so-called *term simulation*.

Furthermore, terms may contain the *reference constructs* `^id` (*referring occurrence of the identifier* `id`) and `id @ t` (*defining occurrence of the identifier* `id`). Using reference constructs, terms can form cyclic (but rooted) graph structures.

2.1 Data Terms

Data terms represent XML documents and the data items of a semistructured database, and may thus only contain total term specifications (i.e. single square brackets or curly braces). They are similar to *ground* functional programming expressions and logical atoms. A *database* is a (multi-)set of data terms (e.g. the Web). A non-XML syntax has been chosen for Xcerpt to improve readability, but there is a one-to-one correspondence between an XML document and a data term. Example 1 on the following page gives an impression of the Xcerpt term syntax.

2.2 Query Terms

Query terms are (possibly incomplete) patterns matched against Web resources represented by data terms. They are similar to the latter, but may contain *partial* as well as *total* term specifications, are augmented by *variables* for selecting data items, possibly with *variable restrictions* using the $\rightarrow$ construct (read *as*), which restricts the admissible bindings to those subterms that are matched by the restriction pattern, and may contain additional query constructs like *position matching* (keyword `position`), *subterm negation* (keyword `without`), *optional subterm specification* (keyword

Example 1
The following two data terms represent a train timetable (from `http://railways.com`) and a hotel reservation offer (from `http://hotels.net`).

At site `http://railways.com`:

```
travel {
  last-changes-on { "2004-04-30" },
  currency { "EUR" },
  train {
    departure {
      station { "Munich" },
      date { "2004-05-03" },
      time { "15:25" }
    },
    arrival {
      station { "Vienna" },
      date { "2004-05-03" },
      time { "19:50" }
    },
    price { "75" }
  },
  train {
    departure {
      station { "Munich" },
      date { "2004-05-03" },
      time { "13:20" }
    },
    arrival {
      station { "Salzburg" },
      date { "2004-05-03" },
      time { "14:50" }
    },
    price { "25" }
  },
  train {
    departure {
      station { "Salzburg" },
      date { "2004-05-03" },
      time { "15:20" }
    },
    arrival {
      station { "Vienna" },
      date { "2004-05-03" },
      time { "18:10" }
    }
  }
  ...
}
```

At site `http://hotels.net`:

```
voyage {
  currency { "EUR" },
  hotels {
    city { "Vienna" },
    country { "Austria" },
    hotel {
      name { "Comfort␣Blautal" },
      category { "3␣stars" },
      price-per-room { "55" },
      phone { "+43␣1␣88␣8219␣213" },
      no-pets {}
    },
    hotel {
      name { "InterCity" },
      category { "3␣stars" },
      price-per-room { "57" },
      phone { "+43␣1␣82␣8156␣135" }
    },
    hotel {
      name { "Opera" },
      category { "4␣stars" },
      price-per-room { "106" },
      phone { "+43␣1␣77␣8123␣414" }
    },
    ...
  },
  ...
}
```

`optional`), and *descendant* (keyword `desc`).

Query terms are “matched” with data or construct terms by a non-standard unification method called *simulation unification* that is based on a relation called *simulation*. In contrast to Robinson’s unification (as e.g. used in Prolog), simulation unification is capable of determining substitutions also for incomplete and unordered query terms. Since incompleteness usually allows many different alternative bindings for the variables, the result of simulation unification is not only a single substitution, but a (finite) *set of substitutions*, each of which yielding ground instances of the unified terms such that the one ground term matches with the other. Whenever a term t_1 simulates into another term t_2, this shall be denoted by $t_1 \preceq t_2$.

2.3 Construct Terms

Construct terms serve to reassemble variables (the bindings of which are specified in query terms) so as to construct new data terms. Again, they are similar to the latter, but augmented by *variables* (acting as place holders for data selected in a query) and the *grouping construct* `all` (which serves to collect all instances that result from different variable bindings). Occurrences of `all` may be accompanied by an optional sorting specification.

Example 2
Left: A query term retrieving departure and arrival stations for a train in the train document. Partial term specifications (partial curly braces) are used since the train document might contain additional information irrelevant to the query. *Right:* A construct term creating a summarized representation of trains grouped inside a `trains` term. Note the use of the `all` construct to collect all instances of the `train` subterm that can be created from substitutions in the substitution set resulting from the query on the left.

```
travel {{                              trains {
  train {{                               all train {
    departure {{                           from { var From },
      station { var From } }},             to   { var To }
    arrival {{                           }
      station { var To }    }}         }
  }}
}}
```

2.4 Construct-Query Rules

Construct-query rules (short: rules) relate a construct term to a query consisting of AND and/or OR connected query terms. They have the form

```
CONSTRUCT Construct Term FROM Query END
```

Rules can be seen as “views” specifying how to obtain documents shaped in the form of the construct term by evaluating the query against Web resources (e.g. an XML document or a database). Queries or parts of a query may be further restricted by arithmetic constraints in a so-called condition box, beginning with the keyword `where`.

Example 3
The following Xcerpt rule is used to gather information about the hotels in Vienna where a single room costs less than 70 Euro per night and where pets are allowed (specified using the `without` construct).

```
CONSTRUCT
  answer [ all var H ordered by [ P ] ascending ]
FROM
  in {
    resource { "http://hotels.net" },
    voyage {{
      hotels {{
        city { "Vienna" },
        desc var H ? hotel {{
          price-per-room { var P },
          without no-pets {}
        }}
      }}
    }}
   } where var P < 70
END
```

An Xcerpt query may contain one or several references to *resources*. Xcerpt rules may furthermore be *chained* like active or deductive database rules to form complex query programs, i.e. rules may query the results of other rules. Recursive chaining of rules is possible (but note that the declarative semantics described here requires certain restrictions on recursion. In contrast to the inherent structural recursion used e.g. in XSLT, which is essentially limited to the tree structure of the input document, recursion in Xcerpt is always explicit and free in the sense that any kind of recursion can be implemented. Applications of recursion on the Web are manifold:

- structural recursion over the input tree (like in XSLT) is necessary to perform transformations that preserve the overall document structure and change only certain things in arbitrary documents (e.g. replacing all `em` elements in HTML documents by `strong` elements).
- recursion over the conceptual structure of the input data (e.g. over a sequence of elements) is used to iteratively compute data (e.g. create a hierarchical representation from flat structures with references).
- recursion over references to external resources (hyperlinks) is desirable in applications like Web crawlers that recursively visit Web pages.

Example 4
The following scenario illustrates the usage of a “conceptual” recursion to find train connections, including train changes, from Munich to Vienna.

The `train` relation (more precisely the XML element representing this relation) is defined as a “view” on the train database (more precisely on the XML document seen as a database on trains):

```
CONSTRUCT
  train [ from [ var From ], to [ var To ] ]
FROM
  in {
    resource { "file:travel.xml" },
    travel {{
      train {{
        departure {{ station { var From } }},
        arrival   {{ station { var To }   }}
      }}
    }}
  }
END
```

A recursive rule implements the transitive closure `train-connection` of the relation `train`. If the connection is not direct (recursive case), then all intermediate stations are collected in the subterm `via` of the result. Otherwise, `via` is empty (base case).

```
CONSTRUCT
  train-connection [
    from [ var From ],
    to   [ var To ],
    via  [ var Via, all optional var OtherVia ]
  ]
FROM
  and {
    train [ from [ var From ], to [ var Via ] ],
    train-connection [
      from [ var Via ],
      to   [ var To  ],
      via  [[ optional var OtherVia ]]
    ]
  }
END

CONSTRUCT
  train-connection [
    from [ var From ],
    to   [ var To ],
    via  [ ]
  ]
FROM
  train [ from [ var From ], to [ var To ] ]
END
```

Based on the “generic” transitive closure defined above, the following rule retrieves only connections between Munich and Vienna.

```
GOAL
  connections {
    all var Conn
  }
FROM
  var Conn ? train-connection [[ from { "Munich" } , to { "Vienna" } ]]
END
```

3 Parsing

Given a sequence of tokens w, a parser for some language L will decide the word problem for L, that is whether $w \in L$ or not. Above that the parser will also investigate the syntactical structure of w and return a syntax tree, if $w \in L$. This tree is used for the further processing of the input. The definitions of language, syntax tree and token are given in the next section.

Generally there are three types of parsers: universal parsers, top-down parsers and bottom-up parsers. Universal parsers, like the CYK-algorithm, Early's algorithm or *GLR* parsing, can parse any context free language. But because these algorithms have at least cubic time complexity in the worst case [GJ90] they are not applicable in practice.

Top-down parsers build a syntax-tree in a top-down manner: they start the analysis at the start-symbol and perform a *pre-order traversal of the syntax tree*. At each step, they make a prediction about the input and try to verify it against the actual input [SB95]. In general, top-down parsing can parse any context-free languages, but *backtracking* is required. This is less efficient than parsing techniques without backtracking (e.g. bottom-up methods). A subset of context free languages may be parsed by a top-down parser without backtracking. These languages are said to meet an *LL(1) condition*. As we use a bottom-up parser for Xcerpt top-down parsing is not addressed in this thesis.

In contrast, a *bottom-up parser* analyzes the input and builds the parse tree from the leaves towards the start symbol. It performs a *post-order left-to-right traversal of the parse tree* [SB95]. A portion of input is read *(shifted)* until a point in the input is reached, where a *reduction* can be performed. This means that the portion of input, matching the right hand side of a production is replaced by its left hand side. In this manner, the whole input is consumed until it is possible to reduce the start symbol. This method, also known as *LR(k)* or *shift/reduce parsing*, is more powerful than efficient top-down approaches. More precisely, the set of *LR* -languages is a superset of the set of *LL* -languages, but still a subset of the context free languages. Like with most programming languages, a parser is used to analyze the context free features of Xcerpt. Therefore only parsers for context free grammars will be described here. To make things more precise we will formally define grammars and languages in section 3.1 . In section 3.2 $LR(k)$ parsers are described. During the construction of such a parser some conflicts can arise. These are discussed in section 3.3. Section 3.4 describes how the syntactical structure of a word is investigated. Finally, section 3.5 describes a more general version of $LR(k)$ parsing.

3.1 Grammars and Languages

In this section some definitions are given. They mainly follow [Sch92, WM92, NS00].

Definition 5 (Language)
An *alphabet* Σ is a set of *terminal symbols* (also referred to as *terminals* and *tokens*). The Kleene star of Σ, Σ^*, is the smallest superset of Σ containing the empty word ε and closed under concatenation. An element of Σ^* is called a *word*.

A *prefix* of a word w is a sequence of terminals u with $w = uv$, $u, v, w \in \Sigma^*$ (analogously a *suffix* v is a sequence of terminals with $w = uv$). A *sub-word* is a prefix of a suffix of a word. It should be pointed out that even though $|\Sigma^*|$ might be infinite, all elements of Σ^* have finite length. A *language* L is a subset of Σ^*.

Definition 6 (Context Free Language)
A *context free grammar* (CFG) is a tuple $G = (V_G, \Sigma_G, S_G, P_G)$ where V_G and Σ_G are finite disjoint sets

of *nonterminal* and *terminal symbols*, $S_G \in V$ is the *start symbol* of G and $P_G \subseteq V \times (V \cup \Sigma)^*$ is a finite set of *productions*. An element $(A, \varphi) \in P_G$ will also be denoted as $A \rightarrow_G \varphi$.

A relation *derives* also denoted as $\Rightarrow_G \subseteq (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ is defined by $\Rightarrow_G := \{(\alpha A \omega, \alpha\varphi\omega) : A \rightarrow_G \varphi \in P_G\}$ with $\alpha, \varphi, \omega \in (V \cup \Sigma)^*$, $A \in V$. The language L produced by a Grammar G is defined by

$$L(G) := \{w \in \Sigma^* : S_G \Rightarrow^*_G w\}$$

where $\Rightarrow^*_G$ is the reflexive transitive closure of $\Rightarrow_G$. The indexed G may be omitted anywhere, if it is clear which Grammar is meant.

Definition 7 (Syntax Tree)
Let $G = (V, \Sigma, S, P)$ be a context free grammar. Let T be a labeled ordered tree whose inner nodes are labeled with symbols from V and leaves with symbols from $\Sigma \cup \varepsilon$.

T is said to be a syntax tree for some word $w \in \Sigma^*$ and grammar G if

- for every inner node of T labeled A with children labeled $C_1 \dots C_n$, there is a production $p \in P$ with $p \equiv A \rightarrow C_1 \dots C_n$ with $A \in V, C_1 \dots C_n \in (V \cup \Sigma)$
- the leaf word $word(T)$ of T, the concatenation of the labels of the leafs of T, is w
- the root of T is labeled S.

3.2 LR(k) and LALR(k)

LR(k) parsing, invented by D.E. Knuth in 1965 [Knu65] is the most powerful bottom-up parsing technique that still has linear time complexity. $LR(k)$ parsers can be constructed automatically using a parser generator. *LALR(k) parsers*, which are closely related to $LR(k)$ parsers, require less space than $LR(k)$ parsers but are less powerful.

A $LR(k)$ parser consists of a word as an input, an output, a stack of states, two tables called *action* and *goto table*, and a driver program. The *configuration* of a $LR(k)$ parser is a pair

$$q_1 \dots q_m \mid t_c \dots t_n$$

of a *state stack* $q_1 \dots q_m$ and a sequence of input tokens $t_c \dots t_n$ called the *remaining input*. The first token of the remaining input t_c is the *current token* and $t_c \dots t_{c+k}$ the *lookahead*. The state on top of the stack q_m and the lookahead are used to decide how to carry on parsing. Before parsing is started the parser is in the *start configuration*:

$$q_1 \mid t_1 \dots t_n$$

Here q_1 is the *start state* and $t_1 \dots t_n$ is *the input*. A *remaining input* is some suffix of the input.

To move from one configuration to another two transitions, called *actions*, are used: *shift* and *reduce*. A shift removes one token from the input and pushes a related state onto the stack. A reduce is always associated with a production: it pops a number of states of the stack and pushes a new one onto it. Conceptually, a reduce is the replacement of a right hand side of the production by its left hand side on top of the stack.

The information when to take which action, is stored in the *action table*. Using the topmost state and lookahead, the action table determines the next action. The *goto table* uses the topmost state and a grammar symbol (a terminal or non terminal) to determine the next state which has to be pushed onto the stack after a shift or reduce.

As the topmost state is not changed on a shift, the state to push onto the stack after a shift is usually also stored inside the action table for efficiency reasons. To keep the explanation easier this will be ignored here. The k in $LR(k)$ denotes the length of the lookahead which is used to index the action table. As there is a $LR(1)$ parser for every $LR(k)$ language [WM92] (see definition of a $LR(k)$ language below) and in practice mainly $LR(1)$ parsers are used, the case $k = 1$ will be used in the following description. The algorithm for $k > 1$ is analogous.

The driver program controls the parser. If the parser is in the configuration

$$q_1 \dots q_m \mid t_c \dots t_n$$

the driver will consult the action table entry $action[q_m, t_c]$. This determines the next step of the parse:

- If $action[q_m, t_c] = shift$, then the token t_c is deleted from the input and the state q_{m+1} is pushed onto the stack, where $q_{m+1} = goto[q_m, t_c]$ is the goto table entry for state q_m and token t_c. The configuration after the shift is:

$$q_1 \dots q_m \ q_{m+1} \mid t_{c+1} \dots t_n$$

- If $action[q_m, t_c] = reduce$ by production $A \rightarrow \beta$, then $|\beta|$ items are popped of the stack. The new topmost element of the stack q_j with $j = m - |\beta|$ and A determine the next state to push onto the stack: $q_{j+1} = goto[q_j, A]$. The next configuration is:

$$q_1 \dots q_j \ q_{j+1} \mid t_c \dots t_n$$

- If $action[q_m, t_c] = accept$ and the remaining input is empty, the parse successfully terminates. If the remaining input is not empty, error management is called.
- If $action[s_i, t_c] = error$, an error has been detected and error management is called.

Because the action and goto tables are very big, much research has been done to reduce their size. The problem was solved by merging states of the $LR(1)$ automaton, yielding a $LALR(1)$ automaton. This decreases the number of states by a factor of ten [ASU86]. As the action and the goto table are indexed by states and grammar symbols, reducing the number of states also reduces the table size.

The drawback is that by merging states information is lost. Therefore, some languages which can be parsed with a $LR(k)$ parser can not be parsed with a $LALR(k)$ parser. Anyhow, as most programming languages may be recognized by a $LALR(k)$ parser, these parsers seem to be the best trade-off between parser size and parsing power.

3.3 LR(k) and LALR(k) Conflicts

The construction of parsing tables is not discussed here. A good explanation can be found at [ASU86]. More detailed discussions can be found at [GJ90, WM92].

There are context free grammars for which it is ambiguous in some configurations which action to take next. In this case $action[q,t]$ has to have multiple entries for some state q and some lookahead t. A parser has to decide nondeterministically between the possible actions. A configuration $q_1 \dots q_m \mid t_c \dots t_n$ where $action[q_m,t_c]$ has more than one entry is called a *conflict configuration* and $action[q_m,t_c]$ a *LR conflict*. There are two kinds of *LR* conflicts: A LR conflict is called a *shift/reduce conflict*, if $action[q_m,t_c]$ has a shift and a reduce entry for the same lookahead; it is called a *reduce/reduce conflict*, if $action[q_m,t_c]$ contains two reductions under different productions for the same lookahead.

A grammar G meets the *LR(k) condition*, if the tables produced by a $LR(k)$ parser generator do not contain conflicts. A language L meets a $LR(k)$ condition, if there exists a grammar G with $L(G) = L$ and G meets the $LR(k)$ condition. Such a language is also referred to as a *LR(k) language*. $LR(k)$ denotes the set of languages which meet the $LR(k)$ condition. In cases where the amount of lookahead k is clear or not important the '(k)' will be omitted. The definitions for the *LALR* and *GLR* are analogous.

There are two possibilities two deal with these conflicts: One possibility is to use a precedence order to choose one action among the possible ones. Usually shift is favored over reduce and a reduce/reduce conflict is resolved by choosing the reduction with the longest right hand side. This approach is used by most $LR(k)$ parsers. The other approach is used by *GLR* parsing and will be discussed in the next section.

A configuration

$$q_1 \dots q_m \mid t_c \dots t_n$$

is called an *error configuration*, if $action[q_m,t_c] = error$ or $action[q_m,t_c] = accept$ and the remaining input is not empty. A configuration is called *legal*, if $action[q_m,t_c] = shift$ or $action[q_m,t_c] = accept$ and the remaining input is empty. All other configurations c have $action[q_m,t_c] = reduce$. If a sequence of reduces is applied to c and an error configuration is reached, then c is also an error configuration; if a legal configuration is reached, c is legal. It should be pointed out that if a configuration is legal, this does not mean that the remaining input is syntactically correct. It only means that it is possible to shift the current token.

3.4 Building the Syntax Tree

The second task of a parser has not been mentioned yet. The parser not only needs to accept or reject an input, it must also analyze its syntactic structure. This is usually done by building a syntax tree.

This tree can be built using an *LR* parser as follows: In addition to the state stack the parser keeps a second stack called *semantic stack*. The elements of this stack are either tokens or *semantic actions*. If the *LR* parser shifts, the current token is pushed onto the semantic stack. If the parsers reduces, the same number of elements is popped of the semantic and state stack. The element to push onto the semantic stack is a semantic action, which may be specified by the parser writer. In the input file for parser generators the semantic actions can be associated with productions. If the parser reduces by production p, then the semantic action associated with p is pushed onto the stack after a reduction. Because the values of the popped elements can be used in the semantic action, a tree-like data structure can be built.

3.5 GLR(k)

GLR parsing, proposed by Tomita [TN91], builds upon *LR* parsing, and was developed to deal with grammars which cause *LR* conflicts. As long as no *LR* conflict configuration is encountered, the *GLR* parser behaves just like a *LR* parser. If it runs into a conflict configuration, conceptually a new parser is started for every possible action. A naive implementation would have a bad time complexity, mainly due to the unnecessary duplication of parsing configurations. But several improvements can be made: All parsers are kept on the same input symbol at all times, such that a breath-first search over the parsing decisions is performed [GJ90]. Therefore, only the stacks have to be copied. Still there are two further optimizations possible:

- A new parser does not need to have its own copy of a stack. Equal prefixes of stacks can be shared.
- Different parsers might be in the same state. These states can merge their stacks, yielding one resulting parser. To deal with differences below the topmost state, a directed acyclic graph is used to represent the stack.

In the worst case this algorithm has an exponentially time complexity. In practice they generally require linear or slightly more time [GJ90].

Several further improvements have been published [AH99]. An introduction can be found at [GJ90, Lju03].

4 Error Handling

Since many programs processed by a parser contain syntax errors, an important task of parsing is error management. This is generally a difficult task: To perform optimal error handling the parser has to know the intention of the user. As this is not realistic most error handling techniques rely on heuristics.

Error Handling can be performed on different levels: *Error detection*, *error recovery* and *error correction*. To perform *error detection* a parser must reject any syntactically incorrect input. Error recovery and error corrections require to convert an error configuration into a legal configuration to enable the parser to resume parsing and possibly find additional errors. Error recovery does this by skipping some input, and error correction adjusts the incorrect input to a correct one.

Any parsing algorithm should perform error detection. In addition *LR* parsers have the *viable prefix property*: They report an error as soon as possible. To be precise, if a *LR* parser for a language L parses input $w = t_1 \ldots t_i \ldots t_n$ and $w' = t_1 \ldots t_i$ is not a prefix of a word from L then the *LR* parser rejects and halts and does so before shifting t_i [NS00]. A *LR* parser does not even reduce in an error configuration. *LALR* parsers also have the viable prefix property but might reduce in an error configuration.

This property helps to detect the position of the error but still the point where the error is detected (the *detection point*) is not necessarily the point where the error occurred (*error point*). Consider the following Xcerpt example:

```
a[ ] with default b[ ]
```

A parser with the viable prefix property will detect an error on `with`, because either the leading `optional` is forgotten or the trailing `with default` is superfluous. It is more probable that the former is the case but this is undecidable for the parser. In addition it might be the case that `a[ ]` contains arbitrary many subterms. Therefore, the distance between the error detection point and the error point could be arbitrarily big. If a parser with the viable prefix property finds an error, the only reliable information is that there has been at least one error on or before the error detection point.

The goal or *error recovery* is to find as many errors as possible in one run. Thus, the parser needs to be restated after an error is found. This means that an error configuration must be altered to a legal one. To do this either a syntactically incorrect input is changed to a syntactically correct one (see error correction) or some portion of the input is discarded until a token is found where parsing can resume.

There are mainly two problems with error recovery: The minor one is that errors within the skipped input are not found. Therefore it is necessary to recompile the input more than once to find all syntax errors.

The major problem with restarting the parser after an error is that *spurious errors* might be produced. These are errors which arise from error recovery. Avoiding spurious errors is very important. They can make the programmer mistrust the error management. He might then re-parse the input every time he has corrected the first reported error. Thus producing spurious errors might make the attempt to recognize many errors superfluous. Here a trade-off is necessary between the amount of errors to be found and the liability of reported errors.

Error correction requires to make assumptions about the intent of the user. It is clear that this might easily lead to spurious errors. On the other hand, if possible corrections are investigated, better *diagnostic messages* can be produced. Additionally, if a good correction is found, parsing can continue at the error detection point. No input has to be skipped and therefore more errors can be found.

A requirement to error handling is *efficiency*. A robust parser should not significantly slow down the processing of correct programs [ASU86]. Error handling techniques should be *language independent* [DP95] such that they are applicable to various languages.

In this section a short survey over several error handling techniques known from literature is given. The classification of these techniques differs between different authors [GJ90, DP95]. I will mainly follow Grune and Jakob. In section 4.1 ad hoc methods, which are highly language dependent, are introduced. In section 4.2 interactive recovery is presented. Global techniques (section 4.3) produce good corrections but lack efficiency. Regional error handling presented in section 4.4 tries to base the correction decision on a small context around the error token. Local strategies (section 4.5) only try to modify the input to consume at least one more token.

4.1 Ad Hoc

Ad hoc methods are the most widely used techniques. Widespread parser generators like YACC and Bison use them for error management. They are efficient and produce very good diagnostic messages for expected errors. They can also be used for error recovery. In this case they produce only poor diagnostic messages and should therefore be combined with other techniques.

Language independency is not achieved with these methods. Since they can not be automatically generated from the grammar, they are not flexible: if the language is changed, the error handling has to be adjusted.

If the most common errors of a language are known, ad hoc techniques are a good choice. As there is no knowledge about common errors in Xcerpt, the Xcerpt parser should only use them for recovery.

Error Productions

If there is knowledge about common syntax errors, the grammar may be extended by *error productions*. These are normal productions, except that they produce syntactically incorrect words. These productions are associated with semantic actions which generate error messages. Now if the parser reduces by such an error production, the error message associated with it can be reported to the user

A drawback of this method is that only anticipated errors can be handled. Furthermore, the readability of the grammar is degraded by mixing real productions with error productions. An even bigger problem is that the modified grammar might not be suitable for the parsing method used anymore [GJ90]. In an *LR* parser error productions might cause a *LR* conflict, even though the language meets the *LR* condition. This might lead to asymmetric error handling where some sort of errors are not handled because of *LR* conflicts.

Anyway, error productions are very easy to implement and may be used with any parser generator. In particular they are the best way to produce very precise error messages for expected errors. Combined with other techniques, they can be used to deal with errors on which the other technique fails. If they are not used extensively, they are a good add-on to other strategies.

As we do not have results on common syntax errors in Xcerpt, error productions should be used with caution.

Error Tokens

Parser generators like YACC and Bison support error handling with error tokens [Joh79]. They are mainly used for recovery but also to produce error messages.

A special token `error` may be used in productions as a terminal or nonterminal wildcard e.g.:

```
<A> ::= error
```

If an error occurs, states are popped until a state is found in which the error token can be shifted. If no such state is found, the recovery fails and the parser halts. Otherwise the error token `error` is inserted into the input. Now normal parsing can continue: the error token `error` is shifted and the stack is reduced by the production containing the error token (`<A> ::= error` in the example above). Finally the input needs to be adjusted. Tokens are deleted until the parser can shift the next k tokens ($k = 3$ in Yacc) without running into an error configuration. Again, if such tokens can not be found, error recovery fails. Otherwise the parser is in the state it would be in, if it had reduced by a normal production.

If the error token `error` is followed by a terminal e.g.:

```
<A> ::= error a
```

the only difference is that all token will be discarded from the input until the token `a` is found. In Johnson's description of Yacc [Joh79] no semantics are given for the case where the error token is followed by a non terminal.

This approach has much in common with error productions. Again the grammar is augmented with special productions for error recovery. Therefore many arguments from above also apply here. But this is a more general approach: unexpected errors can be handled as well, but only if they occur in expected places. Because input is skipped, errors might not be found.

A problem arises, if for example the beginnings of statements are not distinctive enough [Joh79]. Then the parser tries to restart too early and produce spurious errors.

Happy (the parser generator for Haskell) has only a limited support for error productions (see. section 6.1). This limited support is not sufficient for the requirements of the Xcerpt parser. Because error tokens are an effective method for recovery they should be used in the Xcerpt parser for this purpose. To be able to do this the parser generator Happy has to be changed. Some implementation details will be discussed in section Recovery (8.3).

4.2 Interactive

If an interactive parser encounters an error, it will not try to correct the error itself. It will rather localize and describe the error as good as possible and ask the user how to correct it.

A difficulty of this approach is the difference between the error location and detection: If the correction is made left of the detection point, changes to the stack are necessary. This is generally difficult and requires some auxiliary data structures to be kept and therefore degrades the performance of the parser, even on correct input [DP95]. Using incremental parsing techniques, which try to alter an existing parse tree instead of building a new one, can increase efficiency here [DMM88].

Another problem is the synchronization of the editor with the parser: If a correction is made by the user, it should also be made in the source file. Thus a syntax-directed editor is required [BDM$^+$93]. The advantage of this approach is that all errors can be corrected in one run and all corrections made are optimal since they reflect the intention of the user.

This approach is very promising, but requires sophisticated techniques like incremental parsing and syntax-directed editors. It is not realistic to implement such a parser in short time.

4.3 Global

Global error handling considers the whole input in order to perform a correction. These techniques produce good repairs [DP95] but have poor efficiency since they require at least cubic time [GJ90]. Thus they are too slow to be used in practice.

The most well known global error handling technique is least-error correction. This method tries to derive a syntactically correct program from the input using as few corrections as possible. A correction might be the deletion or insertion of a token as well as the replacement of one token by another. A good description of global least-cost error correction can be found at [GJ90].

Because of the bad efficiency of this approach, it is not recommended as a recovery strategy for Xcerpt.

4.4 Regional

Opposed to global error handling, *regional error handling*, often also referred to as *phrase level error handling*, considers only a limited amount of context around the error token as candidates for possible corrections. The advantage of these methods is that they can be used with any kind of bottom-up parser and that they are completely language independent.

A phrase level correction has two phases: A *condensation phase* gathers context around the error. Context is collected left of the error detection point (this is called *backward move*), or right of the detection point (*forward move*), or both. In a *correction phase* information gathered in the condensation phase is used to modify the error configuration to be able to restart the parse. Repairs on the input are usually the deletion, substitution or insertion of a token, the merging of two tokens or spelling corrections.

There is a problem with backward moves: the tokens to the left of the error token have not only been shifted onto the stack, they might even be ‘lost’ in a reduction (recall that *LALR* parsers might still perform reductions in an error configuration). Since it is not generally possible to undo reductions, additional information is required, which has to be collected before the error is detected. Thus all techniques which consider corrections to the left of the error point decrease the efficiency of the parser, also on correct input.

Validating techniques may be seen as a subclass of phrase level error handling. These techniques identify possible repairs. Then these repairs are validated by continuing to parse on trail. If the parser can shift several tokens without running into an error configuration, the repair is accepted. This leads to a second, possibly smaller set of repair candidates. The decision among these is usually taken concerning different kinds of heuristics: either *costs* are associated with different edit operations and the least cost repair is chosen [ABBS83], or the amount of input that can be parsed without yielding another error is taken into account [BF87].

The Problem with validation is that if another error is encountered, it is unclear whether the repair is not good, or another syntax error has been found. This problem is called the *spurious error problem*. Another problem is efficiency: usually several trails are necessary to decide which token to use.

In the following four sections examples of phrase level corrections are presented. Panic mode (section 4.4) is the simplest form of regional error handling. Graham and Rhodes (section 4.4) try to ignore the fact that an error occurred and continue parsing. Sippu’s and Solisalon-Sioninen’s algorithm (section 4.4) tries to replace an erroneous portion of input by a non terminal. In section 4.4 an example of a validating technique is introduced. A further example of phrase level error handling is Penello and DeRemer [PD78].

Panic Mode

This is the simplest form of phrase level corrections. No left context is used, and no correction is made. The implementation is similar to the one for error tokens. If an error is detected, the input is discarded until a synchronization token, called *beacon* is found. Beacons are usually delimiters such as ‘`,`’ or ‘`;`’. States are removed from the stack, until the state on its top permits to continue parsing with the beacon as current token. If more errors are detected soon after restarting the parse, these are either ignored or a recursive call to panic mode recovery is made. The difference to error token recovery is that it is called whenever an error is detected, not only in anticipated cases.

The drawback of this method is that no attempt is made to analyze the error. Thus no good diagnostics can be provided. In addition tokens are skipped, so that not all errors can be found.

The advantage is the fact that it does not affect the performance on correct inputs and that it is easy to implement. In addition it ensures termination. For these reasons panic mode is often used for recovery.

Graham and Rhodes

In the Graham and Rhodes [GR75] method, if an error is found, all possible reductions are performed on the stack as a backward move. Then, in the forward move, tokens are shifted, ignoring the fact that no shifts may be performed according to the action table. Shifting is continued, until either a reduction is called, that spans the error detection point or an error occurs. In the latter case error handling is called recursively. In the former case correction phase starts: To perform the found reductions, changes have to be made to the stack, the condensed tokens or both. These edit operations are related to costs, assigned by the parser writer. The edit with the least cost is chosen, the reduction is accomplished, and the parsing may continue. A more detailed description of the algorithm may be found at [GJ90].

Sippu and Solisalon-Sioninen

Seppo and Sippu [SSS83] try to identify an error phrase, which is then deleted from the input and replaced by a suitable state on the stack. If the string

$$q_1 \dots q_m \mid t_c \dots t_n$$

is an error configuration, then the substring

$$q_{i+1} \dots q_m \mid t_c \dots t_{j-1}$$

with $1 \leq i \leq m, c \leq j \leq n$ of that configuration is an error phrase, if one of these conditions hold:

- removing of the substring allows the parser to advance at least a fixed distance into the forward context
- there is a nonterminal A such that a valid action is defined in q_i on A, and after processing A the parser can advance at least a fixed distance into the forward context

Here q_i,A and t_j are the *recovery state*, *reduction goal* and *recovery symbol*.

A detailed discussion of different strategies for selecting the error phrase and state is presented in [SSS83]. Further information on this topic may be found in [Cha91].

Burke and Fisher

Burke and Fisher [BF87] use three phases of recovery: *simple repair*, *scope recovery* and *secondary recovery*.

A validation technique is used to perform *simple repair*. To deal with the problem of editing the left-context, Burke and Fisher use two parsers. The first parser simply checks for syntactical correctness. The second one, called *deferred parser*, is always *k* tokens behind. The tokens between the two parsers are stored in a *deferred token list*. If the first parser detects an syntax error, it is possible to use the second parser to validate corrections in the left context.

A *simple repair* (such as insertion, deletion, substitution or merging of tokens) is not only tried on the error token, but also on the tokens inside the deferred token list. These repairs are validated with a fixed validation length: To remain in consideration a repair must allow to parse at least a fixed number of tokens (*MIN_ADVANCE*) into the right-context, without finding an error. The repairs are grouped into sets, depending on the repair mode (insertion, deletion ...). If more than one set has a single remaining candidate, then one repair is chosen using a heuristic preference order: merge, misspelling, insertion, deletion, substitution. If none of these sets have a single candidate but the farthest parse check distance exceeds a threshold value ($MIN\_ADVANCE + 2$), the set is chosen by applying the same preference order as above. If this set has more than one candidate, one candidate is chosen arbitrarily.

If simple recovery fails, *scope recovery* is used. *Closers*, such as closing parenthesis or '`END`' have to be predefined by the parser writer. These are inserted and validated similar to simple repair candidates. The differences are: only insertion is considered as repair, repairs only need to parse trough one token beyond the error token, and recovery is called recursively, if another error occurs.

Secondary recovery is called, if scope recovery fails. It is checked, whether parsing can continue, if a suffix of the stack is deleted. If this check fails, the current token is deleted from the input and the succeeding token becomes the current token. Then secondary recovery is called recursively. This is done until either a token is found where parsing may continue, or the end of file is reached. The parse check ensures the termination of error recovery.

The problem concerning the corrections in the left-context can not be seen as satisfactorily solved by Burke and Fisher: In cases where the error point is more than *k* tokens behind the error detection point (where *k* is the length of the deferred token list) the error can not be corrected, and the processing of correct input is slowed down due to the use of the second parser.

In *statistical tests*, this technique shows to be very successful. Tree fourth of syntax errors are repaired "excellent", meaning that the repair is equal to one a human reader would make.

4.5 Local

Local error handling techniques have been developed to increase efficiency of phrase level corrections [DP95]. They try to modify the input only on the error detection point in a way such that it is possible to continue parsing for at least one token. This ensures termination. They can be seen as validating techniques with validation length one.

Due to the viable prefix property, local error handling techniques are able to repair any error. But this does not mean that the repair is good. In the worst case the whole remaining input has to be replaced by a different one.

Local error handling is based on the assumption that the error point is identical with the detection point. Errors for which this is not the case will not be repaired in a natural way. The repair might even lead to spurious errors. To avoid this, either a secondary recovery such as panic mode is used or validation.

This is acceptable because, as Ripley and Druseikis [RD78] investigated on student Pascal programs, errors are mostly infrequent, sparse and simple: 60% of the programs were syntactically and semantically correct. 80% of the erroneous statements had only one error and 90% of errors were single token errors [ASU86]. The problem is that normally, errors that are difficult to repair for a parser are also difficult to repair for a human. Therefore, especially in difficult cases where good error management is important to the user, local techniques might fail.

Anyway, since local error handling is very efficient and good results can be achieved for most errors they are the state of the art [DP95, Cer02] and a local technique should be used for the Xcerpt parser. In section 9 an idea to deal with the problems sketched above is presented.

In the next section we discuss the correction of spelling errors. In section 4.5 we present a cost based technique. A further example of local error handling can be found at [ABBS83].

Spelling Errors

The idea of spelling error recovery is to correct misspelled keywords. Whenever an error is found on an identifier this method should be used. A check is done whether there is a keyword which is 'similar' to the found identifier. A measurement for similarity of two strings could be for example the Levenshtein distance. If this distance is smaller than a certain value, the identifier can be replaced by that keyword.

This technique should be used in combination with validation. Consider the following Xcerpt example:

```
SHAPE { COLOR[``RED''] FORM[``CIRCLE''] }
```

As the Levenshtein distance between `FORM` and `FROM` is only 2 and an error is found on `FORM` an error management system with spelling error correction but without validation would try to delete `FORM` and insert `FROM`. This is obviously a very bad correction that will lead to spurious errors.

The advantage of this error correction technique is that it can be used with any other type of error handling because it is based on the difference between two strings. As the technologies used to build the Xcerpt parser do not support validation, this technique can not be used for Xcerpt error management.

McKenzie, Yeatman et al.

In the work of McKenzie Yeatman et al., if an error is found, new configurations are generated until a legal configuration is found. The way these configurations are generated ensures that the state space is investigated in a breath-first and least cost manner, and a least cost recovery will be obtained [MYV95]

Insertion and deletion costs for each token must be specified by the parser writer. Starting with the error configuration, new configurations are generated by either deleting or inserting one token. The cost related to a new configuration is the sum of the cost for the edit operation and the original configuration. A priority queue is used to store the configurations. If it is not possible to resume parsing in the configuration with the least costs, new configurations are generated by investigating all possible transitions from the least cost configuration. Since there is no upper bound to the recovery time for most grammars [MYV95], a secondary recovery is needed, in the case that the number of configurations added to the queue exceeds a certain number.

This method is both efficient and able to correct most syntax errors. Using up to 1 000 configurations added to the queue, 87.9% of the syntax errors were repaired [MYV95]. Increasing this number to 1 000 000, 98.0% of successful repairs are achieved [Cer02]. Even with a big amount of configurations the algorithm is still fast: using 1 000 000 configurations the recovery took two seconds in the worst case (no repair could be generated and secondary recovery is entered) on an Athlon 800 Mhz (The recovery time is proportional to the number of configurations queued) [Cer02].

Carl Cerecke [Cer02] presents a modified version of the algorithm to improve the repair for difficult errors. His version lowers the number of configurations needed to find a repair by a factor of 2.5 to 7. The time necessary for a correction is decreased for difficult repairs were for simple ones it is increased. As these simple errors are still repaired quickly (0.1s on the Athlon 800), this is a good trade. Further, the number of successful repairs is increased to 98.4%. It should be pointed out that these investigations do not consider the quality of repair. It can be assumed that techniques using more context would yield better corrections, tough lacking in efficiency.

Even though this approach has all the disadvantages of local error handling it gives good repair and is efficient. It is language independent and may be fine tuned by adjusting the costs. On the long term this approach would be a good idea for the Xcerpt parser.

Part II

The Xcerpt Parser

5 The Grammar

Within the scope of this work, an EBNF grammar for Xcerpt has been developed (see appendix A). Before this was done no formal description of the Xcerpt Syntax existed, apart from the rather informal description in [Sch04]. It is important that the grammar is correct in the sense that it produces all words from the language Xcerpt and no other. The syntax tree that is built must be correct in the sense that the further processing of it is correct. The *LALR*(*1*) condition must be met to allow Xcerpt programs to be parsed efficiently. These issues will be discussed in the next two sections.

5.1 Correctness

The correctness of a parser has two aspects:

- accept a program iff it is a word of the language Xcerpt
- if the program is syntactically correct, then build up the syntax tree correctly

The old parser meets the second condition but not the first. It parses a superset of Xcerpt. This is not acceptable for a robust parser because not every syntax error can be detected. It showed to be less work to write a new parser than to change the existing one.

The grammar was constructed in cooperation with Sebastian Schaffert and using the description of the Xcerpt syntax in his PhD thesis [Sch04].

The parser, built according to the grammar, was tested by parsing the Xcerpt Use Cases and a couple of test programs. The syntax tree was built equally to the old parser. Concerning the first point the new parser discovered syntactic errors which the old parser did not find, two of them in the Xcerpt Use Cases.

5.2 LALR(1) Condition

A parser corresponding to the new grammar has been implemented using the parser generator Happy. Parser generators test whether the input grammar meets the *LALR*(*1*) condition while the parser is generated. The Language Xcerpt meets this condition in all but two cases: the Xcerpt term

```
optional optional t1[ ] with default t2[ ]
```

is ambiguous and thus not *LALR*(*1*). It is unclear whether

```
optional ( optional t1[ ] ) with default t2[ ]
```

or

```
optional ( optional t1[ ] with default t2[ ] )
```

is meant.

To avoid this ambiguity, two things are done: an precedence is used such that the second version is implicit. In addition it is allowed to use parenthesis if the other version is desired.

Analogously,

```
not t{var x} where {x > 0}
```

could either be parsed as

```
( not t{var x} ) where {x > 0}
```

or

```
not ( t{var x} where {x > 0} )
```

In this case the second version is implicit. To keep the Xcerpt syntax 'symmetric', any term may be enclosed in parenthesis.

Generally enforcing the parenthesis would make the Xcerpt grammar meet the *LALR*(1) condition. Since by using this precedence Xcerpt becomes unambiguous and the version without parenthesis is nicer to read, the parenthesis may be left out. It should be pointed out that this does not affect the speed of parsing.

6 Technologies

As the current prototype implementation of Xcerpt is implemented in Haskell [Tho99, PHF99], it is desirable to also implement the parser in Haskell. There are two basic approaches to parsing in a functional programming language:

- **Parser combinator libraries:** A parser is written by hand, using a combinator library
- **Parser generators:** A parser is generated from a context free grammar by a parser generator

Generated parsers allow to model on a higher level of abstraction: instead of writing a program, a grammar may be specified. This makes parsers generated from parser generators more flexible to changes of the language. Therefore, we use a parser generator to build the Xcerpt parser.

Apart from flexibility, we wanted to ensure that the Xcerpt syntax satisfies the *LALR* condition. This ensures that Xcerpt can be parsed in linear time and that the parser size stays moderate.

There are several tool that support parser generation in Haskell. Some of them build on parser combinators (Lucky [K9797], Parsec [Lei01]) others on GLR parsing (HASDF [dJKV99], HaGLR [Fer04]). They all can parse any context free language, but have exponential time complexity in the worst case. The only tool which has linear time complexity in the worst case is the *LALR* parser generator Happy [MG01]. As the old parser was generated with Happy we use Happy to generate the new parser.

We use Alex [DJM01], a lexer generator for Haskell to generate the Xcerpt lexer.

Since Happy is implemented in Haskell, all code examples in this section are given in Haskell syntax. Apart form basic types like `Int` or `Bool` the types `Configuration`, `Token` and `ParseResult` will be used to denote a configuration, a token and the result type of the parse.

In the next section we introduce Happy. In section 6.2 we discusses its error handling facilities. It shows, that these are very limited. In section 6.3 we discuss some extensions for Happy which improve the error management support of Happy.

6.1 Happy

Happy is a *LALR*(*1*) parser generator, which takes an annotated grammar as input and produces a Haskell file implementing a parser for the grammar. It tests whether the grammar meets the *LALR*(*1*) condition while it generates the parser. It is well supported and documented. In this section we describe some features of Happy, which are used by the Xcerpt parser.

Happy allows to have multiple parsers within one file. This can be used for example to have one parser parsing Xcerpt programs and another one parsing Xcerpt terms without having to write a additional grammar for Xcerpt terms. A recent add-on to Happy is the support for *GLR* parsing. This allows Happy to parse most context free grammars. The implementation uses the same parsing tables as the *LALR* mode and differs only in the driver program.

Happy makes intensive use of *monads*. Monads originate from category theory [Obr98] and can be used to mimic impure feature such as state or exceptions in pure functional languages [Wad92]. A good introduction can be found at [Wad92].

There are different kinds of parsers which Happy can generate from a grammar. These parsers differ in performance and behavior and one can chosen among them by directives and command line flags. Using a `monad` directive the generated parser will thread a monad through the parse [MG01]. This means that the type of the syntax tree will be of monadic value. This allows to add a state to the parser, which can be used for example to keep track of line numbers or to perform some error management.

If the `lexer` directive is used, the parser will not operate on a list of tokens but call the lexer every time a new token has to be shifted. This leads to a slight performance win and allows some communication between the parser and the lexer [MG01].

6.2 Error Handling with Happy

If a syntax error is detected, a special function `happyError` is evaluated. This function may be implemented by the parser writer to perform error handling. The type of the function differs, depending on the directive used. In the standard case it is of type `[ Token ] -> a` (a mapping from a list of tokens to some type `a`). As it is polymorphic in return type, a call to the Haskell function `error` must be made and all computed results are lost.

If the `monad` directive is used, the type changes to `[ Token ] -> M a` where `M` is a monad. The monad may be implemented by the user. Furthermore, the semantic actions associated with the productions may have a monadic type. It is possible to use error productions to produce error messages or warnings and store them inside the monad. If the parser terminates successfully or an unexpected error occurs it is possible to report the generated error messages.

There is only a very limited support for error management implemented in Happy. Unlike other parser generators like YACC or Bison there is no real error token. The error token supplied by Happy was implemented to make Happy able to parse the Haskell layout rules.

To deal with errors the action table entries are changed. In all states, in which there is no shift for the error token, all entries which are labeled *error* are labeled *reduce by p* instead. The production *p* is chosen by a heuristic. If an error is detected the error token is inserted. The idea is that if an error is detected the parser reduces into a state that can deal with the error token automatically using a shift action. If the parser is in such a state parsing can continue. Unfortunately this is almost never the case. In practice, it does not make a difference whether the error token is inserted in a production or not.

The main problem with the error management of Happy is that unexpected errors cannot be handled. If such an unexpected error is found, the function `happyError :: [Token] -> ...` is called. As the stack of the parser is not passed to the error management function, all information about the input that has already been parsed is lost.

Therefore the only possible error management with Happy is to use error productions. As discussed in section 7.1 these are only useful in combination with other techniques or if there is good knowledge about the distribution of syntax errors. If reasonable error handling is required the parser generator Happy has to be extended. A possible extension is discussed in the next section. The implementation of this extention has been started, but could not be finished within the time frame of this thesis.

6.3 A Framework for Error Management with Happy

The Idea

If someone wants to implement a new error handling technique for *LR* or *LALR* parsers, usually a parser generator has to be changed. It requires a lot of time until the generator is understood before the new idea can be implemented. We propose a simple but powerful framework for Happy which gives the user full control over the error management. Building on this framework a big variety of error handling techniques can be implemented.

This approach seems reasonable since it is not (much) more work than a less general approach. The advantage is that it could be used in various projects. Currently, there are four projects on the chair of Prof. Bry using Happy. As the framework does not implement an error handling strategy but allows the user to do so, this framework could be used by other projects of the chair to implement their error management. Of course, if a language independent technique is implemented this technique could also be used for other projects. In addition, someone else might implement some fancy error handling techniques and publish them on the web. These techniques could be used for Xcerpt or other projects.

The idea derives from the observation that most error managing techniques alter an error configuration to a legal configuration: We therefore suggest to pass the error configuration to the parser writer if an error is found. If there is a possibility to alter this configuration and restart parsing, the parser writer has full control over the error managing process.

The Implementation

First the type of `happyError` has to be changed from `[Token] -> Parseresult` to `Configuration -> ParseResult`. Now if `happyError` is called the configuration of the parser may be modified in any way.

Three features have to be added to Happy: A function to restart the parser in a possibly altered configuration (`restart`), a function to reset the parser into the configuration it was in before it shifted the last *n* tokens (`unparse`), and a function to check whether it is possible to shift *k* tokens in a configuration without running into an error configuration (`parseCheck`).

The function `restart :: Configuration -> ParseResult` restarts the parser in the favored configuration. In combination with the new type of `happyError` this might seem sufficient to change the configuration and carry on parsing. This would be true for *LR* parsers, but *LALR* parsers have the unpleasant property of performing reductions if they reach an error configuration. As the reductions applied to the stack depend on the current lookahead token, the reductions for one lookahead token differ from those on other ones. Therefore, the stack has to be reset to the state it was in after the last shift. If this is not done, it might happen that even though the appropriate correction is used the parse will fail. To avoid this it would suffice to be able to reset the parser to the configuration it was in after having shifted the last token, but it is no more work to unparse further.

There are various possibilities to implement unparsing. In any case the last *k* tokens must be stored. Burke and Fisher [BF87] suggest to use two stacks. One of them, called the *deferred stack*, is always *k* tokens behind. If an error is encountered it can be used to rebuild the other one. It is possible to defer the semantic stack. This has two advantages: No additional work has to be done to be able to unparse and there is no need to unparse the semantic stack, which might slow down the parser and use lots of space.

McKenzie and Yeatman [MYV95] use a much simpler approach. They analyzed the stack sizes while parsing Modula, Pascal and C programs. These sizes showed to be small on the average (23 for Modula, 20 for Pascal and 13 for C) as well as having small maximums (38, 60, 54). Therefore, they simply save the stack after every shift. In McKenzies and Yeatmans implementation, parsing speed was slowed down from 14600 to 10300 tokens/sec when stacks were saved. As this method is quite easy to implement and the decrease of speed seems acceptable, unparsing in Happy will be implemented this way.

The last feature is not really desired, but very useful. In many cases it is necessary to measure the quality of a repair. This is usually done by determining the number of tokens the parser will shift without running into an error configuration. The function `parseCheck :: Configuration -> Int -> Bool` can be used to check if the parser will successfully parse a certain number of tokens. If the number of tokens which may be shifted before the next error occurs is of interest `restart` can be used.

Discussion and State of Work

With these features it is possible to implement most of the error handling techniques mentioned in section 4. The range goes from simple local cost based repairs to more sophisticated ones like the approach of McKenzie, Yeatman, and De Vere [MYV95]. Even error token recovery can be implemented (The algorithm will be sketched in section 8.3). The advantage of this framework is that error management will not have to be provided by the parser generator but may be implemented by the parser writer. This allows him to fine tune the error handling to fit his needs. Of course, the implementation of an error handling strategy requires a deeper knowledge about shift/reduce parsing than is needed to use a parser generator. But error recovery strategies could be implemented and added to Happy as a library.

The implementation of these features is started but is not completed due to the time frame of this thesis. A problem is the various sorts of parsers that may be generated by Happy. To ensure that these changes will be maintained in further versions of Happy, at least the three main types should be supported. The work of investigating how these features can be implemented is finished for two of the three cases. This means that they have been implemented by hand into parsers generated by Happy. The third case is in the middle of its implementation. It has also been investigated how these features may be implemented into the parser generator but no code has been written.

7 The Error Handling

Xcerpt is a very young language. This makes error handling very important. The goals of error handling should be to provide meaningful error messages and detect as many errors as possible while avoiding the production of spurious error messages. In addition the compilation of correct programs should not be affected significantly by error management.

At the moment these goals cannot be achieved. As discussed in section 6.1, Happy does not support error handling in a adequate way. Nevertheless, some error management is possible, and the features implemented for the Xcerpt parser are presented in this section.

We use error productions to give warnings, if features of Xcerpt are used that are not implemented in the current prototype.

Further, to have some error recovery we simulate a limited version of panic mode with Happy. This will be presented in section 7.3. Section 7.2 describes a little improvement to error localization of Happy implemented within this project thesis.

7.1 Error Productions

As discussed in section 7.1 error productions are only useful if there is knowledge about common errors in a language. This is not the case for Xcerpt. Instead, we use error productions to produce warnings if features of Xcerpt are used, that are not implemented in the current prototype.

The implementation requires the use of monads. For this purpose a monad was implemented for the Xcerpt parser. Warnings and other error messages are stored inside the monad during the computation. This monad is also used to implement the error recovery of the Xcerpt parser discussed in section 7.3.

7.2 Error Localization

When Happy finds an error the function `happyError :: [ Token ] -> ParseResult` is called. In version 1.14 of Happy, the list of tokens starts with the token succeeding the error detection token. As the location of the token is usually stored with the token, this is unfavorable: If the location of the error detection point is to be reported to the user, the token on which the error is detected is needed.

To avoid a workaround that slows down the parser or consumes extra space, the parser generator Happy was changed such that the list of tokens begins with the token where the error is detected. A patch was submitted to and incorporated in Happy version 1.15 as part of this thesis.

The implementation of this little patch helped understanding the working of the parser generator. This knowledge was very useful for implementing the other changes to Happy which are presented in section 6.3

7.3 Error Recovery

One of the most important tasks of a parser is to find more than one syntactic error in one run. Otherwise a programmer has to parse an incorrect program as often as it has errors. This will not only slow down the programmer significantly, it will most certainly annoy him. Therefore error recovery is very important.

Techniques known form literature perform recovery by discarding symbols from the input and popping states form the stack, until a configuration is found, where parsing may continue. The problem is that the stack is not passed to `happyError`. Because it contains all the information about the input parsed until the error point, the stack is necessary to restart parsing in the general case.

Luckily, in the case of Xcerpt there is a way to simulate a limited version of panic mode with multiple parsers (see 6.1) and monads. If an unexpected error is encountered, a message is generated and, just like in panic mode recovery, input symbols are discarded until a beacon is found. The beacons for the Xcerpt parser are the tokens `CONSTRUCT`, `GOAL` and `FROM` which separate the construct and query parts in rules. Now if `FROM` (`CONSTRUCT`, `GOAL`) is found, a new parser that recognizes a query term (construct term) starts parsing the term starting after the beacon. This term parser can parse this term to find more errors. If the term is syntactically correct, the end of the construct part (query part) is reached (recall that it only may consist of one term). At this point an error is detected, because the stack of the term parser is empty, but the remaining input is not. As this error is expected, it is not reported to the user. Instead, recovery is called again and the next construct or query term is parsed.

As Happy supports multiple parsers within one file, it is not necessary to rewrite parts of the grammar for these term parsers. A monadic parser allows passing the generated error messages from the failed parser to the new one. With this technique several unexpected errors can be detected.

This technique skips big amounts of input (almost the whole program in the worst case) but is reliable and fast. The first syntax error will be found in each construct or query part of a program. Unless the beacons `CONSTRUCT` and `FROM` are used inside construct or query terms, no spurious errors will be produced. Since `CONSTRUCT` and `FROM` are very strong symbols, it is unlikely that they will be used by mistake. It is also possible to return a syntax tree for all syntactically correct rules of a program. These could then be passed to semantic analysis to be checked for further errors.

To decrease the amount of skipped tokens, it was investigated what changes have to be made to restart the parser on the next term. This showed to be impractical, because it is necessary to have some knowledge about the input that has already been shifted, when an error occurs. On one hand it has to be known whether to start a construct term parser or a query term parser. On the other hand, the new parser needs to know how many and what kind of terms include the term it is supposed to parse. For example if the term parser has to be restarted on the input

```
d[ ]]
```

it is impossible for the parser to know whether this is an error or the left context looked for example like this:

```
a[ b ; c[ ], d[ ]]
```

(Here, the parser detects an error on the semicolon, finds the next beacon on '`,`' and a new parser is started on `d`). To deal with this, information could be gathered before an error occurs. But this would not only slow down the parser. Work would be doubled only to be able to throw the better part (the stack) away when an error occurs.

8 Future Work

The error management performed by the Xcerpt parser is very limited at the moment. The main flaws are the diagnostics and the recovery: errors are only located and not described and only one error inside a construct or query part of a rule is found. The reason for these limitations is the limited support for error management provided by Happy. If the implementation of the features described in the section 'A Framework for Error Management with Happy' (6.3) is finished, these flaws can be remedied.

In this section some ideas will be presented how these extensions can be used to improve the error management of the Xcerpt parser. They can be seen as use cases for the above framework. A three phase error management is suggested. The first phase tries to correct spelling errors (section 8.1). In the second phase (section 8.2) some other local repairs of the input are tried and diagnostic messages are produced. If the attempt to find a repair fails, recovery is entered (section 8.3). Some input will be discarded, to allow the parser to resume parsing.

8.1 Spelling Errors

Spelling errors can be corrected relatively easily and this can be done independently of any other techniques used. It should therefore always be the first thing to try.

The most simple repair would simply merge the error token with the following token. E.g.: `wit hout` could be corrected to `without`. If such a repair parse checks a few tokens it is very likely to be a good repair.

Next, if an identifier is found at a point where a keyword is expected and the keyword and the identifier are 'similar', then it is likely that a spelling error is detected. A measure for similarity could be for example the Levenshtein distance or a slight modification of it that treats the transposition of two characters as a single edit. If the distance between the identifier and the keyword is smaller than a threshold value, the identifier should be replaced by that keyword. To avoid spurious errors, the chosen correction should be validated (using `parseCheck`).

8.2 Correcting Phase

The goal of this phase is to try to correct the input if this is easy and to produce diagnostic messages. Any correcting technique could be used here. Three techniques of increasing complexity are suggested. The techniques build upon each other and could be implemented in the given order to quickly achieve better error management.

The most simple approach would be to try no corrections and only produce some diagnostic messages. This could be done by performing simple edits on the input and then checking whether this correction leads to an error within the next tokens. An edit operation could be the insertion of a token, the substitution of a token by another, and the deletion of the error token. Then, all repairs that parse checked further than a certain number of tokens can be reported to the user as diagnostics. After that, recovery is entered to find more errors.

As a next step a simple correction could be tried. If the above algorithm yields more than one candidate, a decision has to be made which repair to use. This could either be done by using the repair which parse checks the furthest into the right context or by assigning costs to the repairs and choosing the minimum cost repair. I would favor the cost based approach because it is more efficient and allows some tuning of the behavior of the error correction. Assigning the costs to edit operations is not simple and best done if there is a collection of syntactically incorrect Xcerpt programs. Then the best cost distribution can be determined experimentally. If this is not possible a heuristics could for example be to have costs which are proportional to the length of the lexeme (number of characters) of a token and using the precedence order over repair modes: insertion, deletion, substitution. Substitution should not generally be treated an a sequence of a insertion and a deletion because the costs might be different to the sum of deletion and insertion costs.

Both these techniques try to only correct single token errors. This is justified because these seem to be the big majority [RD78]. If a sequence of more than one simple repair is considered time complexity increases exponentially with the number of simple repairs. A first approach could limit the number of simple repairs to a fixed number. A very good way to increase efficiency and to ensure that the least cost repair is found is the approach of Cerecke [Cer02] who builds on McKenzie, Yeatman, et al. [MYV95]. But the work to implement this should not be underestimated. The description given in 4.5 is a strong simplification of the algorithm.

Any of these techniques will fail on certain errors. Therefore, secondary recovery is required. I would suggest to used error token recovery if the attempt to repair the error fails. In any case, the result of the analysis, e.g. possible repair candidates, can be reported as diagnostic messages.

8.3 Recovery

Using the extensions for Happy discussed in section 6.3, error token recovery can be implemented. As with normal error tokens, a token `error` has to be inserted into the grammar at all places where errors are expected. There is no difference between this token and others. The algorithm should work like this:

If error token recovery is entered, the first thing to do is to insert the error token and pop states of the stack until a shift is possible. This is not a problem since the `Configuration`, which contains the stack, is passed to `happyError`. `parseCheck` can be used to check whether the error token may be shifted. Now the search for the token to restart parsing starts. `parseCheck` can be used to check whether the current configuration allows to continue parsing. If this is not the case, a token is dismissed and `parseCheck` is called again. If a configuration is found which allows the parsing to resume, `restart` is called.

If the version of error tokens is desired where the error token is followed by a terminal more than one error token is needed. For example two error tokens could be called `error_colon` and `error_open_bracket`. These should have an equivalent semantics to a normal error token followed by a colon and a opening bracket. Now if an error is found the check whether a shift is possible must be done for all error tokens and all state on the stack. The error token that may be shifted with the least number of states removed from the stack is the error token to use. This token determines the further action. If for example `error_colon` is used all tokens are deleted from the input until a colon is found.

Again, as the implementation is left to the user, other actions can be taken. Imagine the situation where the parser is supposed to restart at the beginning of the next Xcerpt term, but not on a sub-term contained by this term. No beacon can be found to mark the end of a term because both comma and closing parenthesis can also be used inside sub-terms. Here, a more complicated search for the end of the term is required that could be performed if the indicating error token is found.

Even error token recovery might fail if the error tokens are not inserted correctly into the grammar. In this case, the version of recovery that is currently used in the Xcerpt parser can be used to recover. This recovery will always find a point in the input to restart parsing, unless the error occurs in the last query term of the program.

The advantage of this form of error recovery is that the implementation can be kept out of the code generated by the parser generator. Therefore, it can easily be tuned to exactly fit the needs of the parser writer. Additionally, it will not be called automatically, but from the function `happyError`. This allows to try some other error handling strategies before error token recovery is used. This is desirable because error token recovery normally skips some tokens and does not produce as good error messages as for example correcting techniques. Therefore, it is better to use it as secondary recovery. This could not be done with the error management features of Yacc or Bison.

9 An Idea to Deal with Context

The two main problems of regional error handling techniques have to do with the usage of context. The first problem is that if repairs are considered to the left of the error detection, it is not easy to determine the tokens on which corrections should be tried. The second problem is the *spurious error problem*: if two errors are found close to each other, it is not clear whether a second error is a real error or arises from the recovery of the first one. In the former case, a recursive call to the error handler is required and in the latter, a different correction should be tried.

Local techniques avoid these problems by ignoring left and right context. Due to the viable prefix property, they are also able to repair any syntax error, but in some cases they do not find a simple repair, because this would require context. Such unnatural repairs might produce spurious errors. As these should be avoided by any means, recovery has to be called more often than with regional techniques.

I will present an idea to address both problems. These problems cannot be solved, but I think especially in difficult cases an improvement of the quality and speed of repair can be achieved. I do not think that this idea is extremely suitable for the Xcerpt parser. This has to do with the syntax of Xcerpt and the amount of work that would be needed to implement it. As I had this idea while working on the Xcerpt parser, I will present it here anyway.

9.1 The Advice Parser

The Idea is to have two parsers. One simply scans the input from left to right. The second one, lets call it *advice parser*, comes into play only if an error is found. If this happens, the lexer scans ahead until a beacon is found. The advice parser is a parser which parses the language in reverse direction (Its construction will be given in the next section). If the beacon is reached, it starts to parse from the beacon towards to error detection point. Four things might happen: The advice parser detects an error

1. to the *right of the error detection point;* then the advice parser gives the normal parser the advice

that if it encounters a further error on this token, it should call error handling recursively instead of dismissing the token.

2. *on the same token as the forward parser;* then a deletion of this token should be a good idea.
3. *on the token preceding the error detection token;* in this case an insertion should be favored.
4. to the *left of the error detection point;* here, the advice should be given that some corrections in the left context have to be considered. Repairs should be tried on the error detection point of the advice parser as well as on the detection point of the forward parser.

This technique can not only help to locate the error and to decide which form of correction has to be made. If the advice parser performs some error handling itself, it can also help to find tokens to use for corrections. For example, if it determines the tokens which are expected when an error is found, these tokens are surely very good candidates for insertion or substitution.

The advice parser has to be a partial parser. For example, in Xcerpt it would have to accept reverse Xcerpt terms. It might be necessary to have more than one reverse parser. If this is required, the decision on which of them to use could either be taken depending on the beacon found, or using validation.

The advice parser slows down recovery of simple errors and it is possible that the advice given is bad. To avoid the former problem, the advice parser should only be used if the forward parser has difficulties to find a good correction. To deal with the latter, the forward parser should try the correction according to the advice given, but if this fails, try normal recovery. In combination with a cost based technique the advice parser could simply decrease the costs of the corrections he advises to try. If the parser always follows the least cost corrections first like in the approach of McKenzie, Yeatman, and Cerecke [MYV95, Cer02], then the advice will be followed but if it does not show to be good, normal corrections are tried.

The big disadvantage of this approach is that the size of the parser will be doubled. As a relatively big part of the space of a compiler is taken by the parser, this cannot be ignored. On the other hand today space is not such a problem any more.

The advantage of this technique is that it supports error handling in difficult cases, where other techniques might fail. As mentioned in section 4.5, error management is most important in these difficult cases. It does all this without affecting the parsing speed of correct programs.

9.2 Construction of a Reverse Parser

Fortunately, the advice parser can be constructed semi-automatically. Before giving the algorithm of its construction, we define the language decided by the advice parser more precisely. To do this and to construct the advice parser, we use a overloaded function *REV* that may be applied to words assumed to be arrays of tokens, sets of productions, and grammars. The algorithms are given in an imperative pseudo code.

If *REV* is applied an array, it has the following implementation:

```
$REV(w : [token]) : [token]$
  **if** $w = []$ **then return** $[]$
  **else return** $REV(w[1] \ldots w[lenght(w) - 1]) ++ w[0]$
```

The length of an array a is computed by $lenght(a)$ and the ith element of a is accessed by $a[i]$. The concatenation of arrays is denoted infix by $++$

Definition 8 (Reverse Language)
The *reverse language* L^R of some language L is defined as

$$L^R := \{w : \mathrm{REV}(w) \in L\}$$

A grammar G^R is called the *reverse grammar* of G if $L(G^R) = L^R$.

As the advice parser is supposed to decide L^R, we need to construct a grammar G^R, with $L(G^R) = L^R$. The grammar G^R can be constructed by reverting the productions of G. For this reason we apply *REV* to productions:

```
$REV\ (P : \{Production\}) : \{Production\}$
    $P^R := \varnothing$
    for all $A \to C_1 \ldots C_n \in P$ do
        $P^R := P^R \cup \{A \to C_n \ldots C_1\}$
    return $P^R$
```

Here, $Production$ is a short hand for $V \times (V \cup \Sigma)^*$ and an element from $Production$ is denoted $A \to S_1 \ldots S_n$ instead of $(A, S_1 \ldots S_n)$, $A \in V, S_1 \ldots S_n \in (V \cup \Sigma)$.

Finally, *REV* applied to grammars

```
$REV((V, \Sigma, S, P) : Grammar) : Grammar$
    return $(V, \Sigma, S, REV(P))$
```

yields the desired G^R with $G^R := REV(G)$.

The proof of correctness ($L(G^R) = L^R$) is shown in the next section. There is still the problem that if a language L has the *LALR* property, the reverse language L^R might not. For example the language $\{a|ba\}^*$ is in *LALR(1)* wear as $\{a|ab\}^*$ is not. But the reverse language of a *LALR* language is certainly a context free language since it has a context free grammar. This means that a parser for context free languages is needed to parse L^R. There are various parsers which parse context free languages. *GLR* parsing for example has linear time complexity on the average [GJ90].

The reverse parser which accepts L^R can be constructed semi-automatically from some parser for L. First, construct the grammar G^R from G using the above algorithm and produce a parser generator input file for G^R. Since no syntax tree has to be build while parsing from right to left, the semantic actions can be omitted. The reason this construction cannot be fully automatic is that the reverse parser has to be a partial parser. The parser writer must specify the nonterminals which should be used as start symbols for the partial parsers.

9.3 Proof of Correctness

In this section we show the correctness of the above algorithm to construct the reverse parser.

Proposition 9
Let $G=(V,\Sigma,S,P)$ be a grammar, $L=L(G)$ a language, and $L^R:=\{w\in\Sigma^*: REV(w)\in L\}$ the reverse language of L. The algorithm REV generates a grammar $REV(G)=(V,\Sigma,S,P^R)$ for that the following holds:

$$L^R=L(REV(G))$$

Proof. It holds that

$$\begin{aligned} L^R=L(REV(G)) \quad &\textit{iff} \quad \forall w\in\Sigma^*(REV(w)\in L(G)\Leftrightarrow w\in L(REV(G)))\\ &\textit{iff} \quad \forall w\in\Sigma^*(w\in L(G)\Leftrightarrow REV(w)\in L(REV(G)))\\ &\textit{iff} \quad \forall w\in\Sigma^*(S\Rightarrow_G^* w\Leftrightarrow S\Rightarrow_{REV(G)}^* REV(w)) \qquad (*)\end{aligned}$$

We show the more general proposition

$$\forall w\in(\Sigma\cup V)^*(S\Rightarrow_G^* w\Leftrightarrow S\Rightarrow_{REV(G)}^* REV(w)) \qquad (**)$$

by induction over the length n of the derivation. Obviously $(*)$ follows from $(**)$.

The base case $n=0$ is trivial. Now we show the step $n\longrightarrow n+1$:

"$\Rightarrow$" Let $S\Rightarrow_G^n w\Rightarrow_G w'$ be a derivation and $w,w'\in(V\times\Sigma)^*$. Without loss of generality we state that $A\longrightarrow\alpha\in P$ is the production used in $w\Rightarrow_G w'$ and that w' derives from w by replacing the k^{th} occurrence of A by $\alpha\in V\cup\Sigma$.

By hypothesis there is a derivation $S\Rightarrow_{REV(G)}^n REV(w)$. As $A\longrightarrow\alpha\in P$ we have $A\longrightarrow REV(\alpha)\in P^R$. Let $w'_R\in(V\times\Sigma)^*$ be the sequence of terminals and non terminals with $REV(w)=w_R\Rightarrow_{REV(G)} w'_R$ by using the production $A\longrightarrow REV(\alpha)\in P^R$ to replace the k-last occurrence of A in w_R by $REV(\alpha)$. Therefore, we have $REV(w')=w'_R$ and $S\Rightarrow_{REV(G)}^* REV(w')$.

"$\Leftarrow$" Assume $\forall w\in\Sigma^*(S\Rightarrow_{REV(G)}^* REV(w))$. We can use "$\Rightarrow$" and get $\forall w\in\Sigma^*(S\Rightarrow_{REV(REV(G))}^* REV(REV(w)))$. As $REV(REV(w))=w$ and $REV(REV(G))=G$ the proposition follows.

10 Conclusion

The three steps of error management are error detection, error recovery and error correction. Error detection is actually more a duty of the parser than of the error management. Parsers discussed in this thesis all have the viable prefix property. An error will be detected as soon as possible. This does not mean that the error can be located easily. Even with this property one or more errors may be located left of the error detection point.

The goal of error recovery is to find as many syntax errors in one run. If an error is detected there are two possibilities: either tokens are skipped until a synchronization point in the input is found where parsing can continue, or the parser tries to repair the error. Skipping some input is always a problem because errors in that portion of input will not be detected. Restarting the parse at some synchronization point is also critical: if the wrong point is chosen, spurious errors might be reported. In addition, as the error is not analyzed, only poor diagnostic messages can be produced.

Error correction produces better diagnostics but requires a bigger amount of heuristics. As it is too expensive to determine the globally best corrections, only a limited amount of context around the error detection point is considered when determining possible correction. This limited context might allow more than one correction. In this case a heuristics must be used to decide among the various corrections. Here again a wrong choice might produce spurious errors.

The idea to deal with context from section 9 tries to decrease the number of possible corrections and at the same time improve the localization of an error. This is achieved by investigating the error detection point from both left and right. A metaphor could be one scientist asking a scientist from a different discipline for help. In some cases the second scientist will not be able to help the first one or even confuse him. In other cases the different perspective of the second scientist might be of great help to the first one: he might either help the first one to find a good solution or point out that an error was made earlier and a different approach should considered. If it shows that the latter case is common, the interdisciplinary dialogue should be encouraged.

Both error recovery and correction try to alter an error configuration to a legal configuration. The various error handling techniques discussed in section 4 do this in different ways. Many parser generators support at least one of the described techniques. The most widely used techniques are error tokens. Happy only supports the most simple error handling technique: error productions. These can only be used to deal with anticipated errors. If an unexpected error occurs, no diagnostic can be reported and no recovery is performed. The parser will simply terminate and locate the error. To achieve better error handling than this, Happy has to be modified.

An idea of an extension for Happy is to let the parser writer decide how to alter the error configuration to a legal one. For this purpose the error configuration is passed to the error handling routine if an error is detected. Now the parser writer may decide how to change this configuration to achieve a legal one. This gives the parser writer a maximum of flexibility in error management.

The main focus of this work is rather exploring the possibilities of error management for Xcerpt and enabling the implementation than on the implementation itself. Several error handling techniques from literature were compared regarding their usefulness for Xcerpt. The suggested technique is relatively easy to implement but still detects many errors and gives error messages that are useful for the programmer. It may be implemented step by step and after each step a version of the parser will be obtained that can be used in practice. The extensions for the parser generator Happy were designed to enable the implementation of this technique. In addition they should be general enough to be allow several improvements of error handling for Xcerpt in the future.

A The Xcerpt Grammar in EBNF

LEXICAL STRUCTURES

```
<digit>         ::= [0-9]
<number>        ::= <digit>+
<float>         ::= <digit>+ ("." <digit>+ )?
<string>        ::= [a-zA-Z]+
<identifier>    ::= [a-zA-Z] [a-zA-Z0-9-_+]*
<regexp>        ::= as defined by POSIX
<fun>           ::= identifier.
<comp>          ::= "<" | ">" | "<=" | "=>" | "=" | "!=".
```

PROGRAMMS

Programms, Construct-Query Rules

```
<program>       ::= <construct-query-rule> +


<construct-query-rule> ::= "COSTRUCT" <ns-ct-term> "FROM" <query> "END"
                         | "COSTRUCT" <ns-ct-term> "END"
                         | "GOAL" <goal-head> "FROM" <query> "END"
                         | "GOAL" <goal-head> "END"
                         | <ns-declaration>+ .

<goal-head>     ::= <ns-ct-term>
                  | "out" "{"  <out-resource> "," <ns-ct-term> "}"
                  | "out" "["  <out-resource> "," <ns-ct-term> "]" .
```

Resources

```
<out-resource>  ::= "resource" "{" <resource> "}"
                  | "resource" "[" <resource> "]" .

<in-resource>   ::= <out-resource>
                  | ("or" | "and") "{" <out-resource> ("," <out-resource>)* "}"
                  | ("or" | "and") "[" <out-resource> ("," <out-resource>)* "]" .

<resource>      ::= ( ( <string> | <var> ) "," )? <string> .
```

Namespace Declarations, Variables

```
<ns-declaration> ::= "ns-prefix" <identifier> "=" <string>
                   | "ns-default" "=" <string> .

<var>            ::= "var" <identifier> .
```

CONSTRUCT TERMS

```
ct = construct
```

Labels, Prefixes

```
<ct-ns-label>    ::= (<ct-ns-prefix> ":")? <ct-label> .
<ct-ns-prefix>   ::= <var> | <identifier> | <string> .
<ct-label>       ::= <var> | <identifier> | <string> .
<label-list>     ::= <identifier> ("," <identifier>)* .
```

Construct Terms

```
<ns-ct-term>     ::= <ns-declaration>* <ct-term> .

<ct-term>        ::=  <ct-ns-label> (<ordered-ct-list> | <unordered-ct-list>)
                   | <var>
                   | <identifier> "@" <ct-term>
                   | <arith-expr>
                   | "optional" <ct-term> ("with default" <ct-term>)?
                   | "(" <ct-term> ")" .
```

Construct-Subterms, Construct-Subterm Lists

```
<ct-subterm>     ::= <ct-ns-label> (<ordered-ct-list> | <unordered-ct-list>)
                   | <string>
                   | <var>
                   | "^" <identifier>
                   | <identifier> "@" <ct-subterm>
                   | <arith-expr>
                   | "optional" <ct-subterm> ( "with default" <ct-subterm> )?
                   | "(" <ct-subterm> ")" .


<ct-subterm-coll> ::= "all" ( <ct-subterm>
                            | "[" <ct-subterm-list> "]"
                            | "{" <ct-subterm-list> "}" )
                        ( <ordering> | <grouping> )?
```

```
                  | "some" <amount> ( <ct-subterm>
                                    | "[" <ct-subterm-list> "]"
                                    | "{" <ct-subterm-list> "}" )
                        ( <ordering> | <grouping> )?
                  | "(" <ct-subterm-coll> ")" .

<ordering>      ::= "order by" <identifier>? "[" <label-list>  "]"
                                    ("ascending" | "descending")?

<grouping>      ::= "group by" "[" <label-list> "]"

<amount>        ::= <number>
                  | "-" <number>
                  | <number> "-" <number> .

<ordered-ct-list>   ::= "[" (<ns-declaration>* <ct-subterm-list>)? "]" .
<unordered-ct-list> ::= "{" (<ns-declaration>* <ct-subterm-list>)? "}" .

<ct-subterm-list>  ::= <ct-attributes>? "," ( <ct-subterm> | <ct-subterm-coll> )
                       ( "," ( <ct-subterm> | <ct-subterm-coll> )  )* .
```

Construct Term Attributes

```
<ct-attributes> ::= "attributes" "{" <ct-attribute> ( "," <ct-attribute> )* "}"
                  | "(" <ct-attributes> ")" .


<ct-attribute>  ::= <ct-ns-label> ( "[" (<string> | <var>) "]"
                                  | "{" (<string> | <var>) "}" )
                  | <var>
                  | "^" <identifier>
                  | <identifier> "@" <ct-attribute>
                  | "all" ( <ct-attribute>
                          | "[" <ct-attribute> ( "," <ct-attribute> )* "]")
                        ( <ordering> | <grouping> )?
                  | "some" <amount> (<ct-attribute>
                                    | "[" ( <ct-attribute> ( "," <ct-attribute> )* "]" )
                        ( <ordering> | <grouping> )?
                  | "(" <ct-attribute> ")" .
```

ARITHMETIC EXPRESSIONS, FUNCTIONS AND CONDITIONS

Arithmetic Expressions

```
<arith-expr>    ::= ( <arith-expr> ( "+" | "-" ) )? <arith-term> .
```

```
<arith-term>    ::= ( <arith-term> "*" )? <arith-factor> .

<arith-factor>  ::= <number>
                  | <float>
                  | "(" <arith-expr> ")"
                  | <function> .
```

Functions

```
<function>      ::= <identifier>  "(" <fun-param> ( "," <fun-param> )* ")"
                    ( (<ordered-ct-list> | <unordered-ct-list>) )? .

<fun-param>     ::= <ct-subterm>
                  | <ct-subterm-coll>
                  | <ct-attributes> .
```

Conditions

```
<condition>     ::= <var> ( <identifier> | <comp> ) <cond-param>
                  | <cond-param> ( <identifier> | <comp> ) <var>
                  | ( <identifier> | <comp> ) "(" <var> "," <cond-param> ")"
                  | ( <identifier> | <comp> ) "(" <cond-param> "," <var> ")"
                  | ( "and" | "or" | "not" ) "[" <condition> ( "," <condition> )* "]"
                  | ( "and" | "or" | "not" ) "{" <condition> ( "," <condition> )* "}" .

<cond-param>    ::= <qr-subterm>
                  | <arith-expr>
                  | <function> .
```

QUERY TERMS

```
qr = query
```

Labeles, Prefixes

```
<qr-ns-label>   ::= (<qr-ns-prefix> ":")? <qr-label> .
<qr-ns-prefix>  ::= <var> | <identifier> | <regexp> | <string> .
<qr-label>      ::= <var> | <identifier> | <regexp> | <string> .
```

Queries

```
<ns-query>      ::= <ns-declaration>* <query> .

<query>         ::= <qr-term>
                  | ( "and" | "or" ) "{" <ns-query> ( "," <query> )* "}"
```

```
                 | ( "and" | "or" ) "[" <ns-query> ( "," <query> )* "]"
                 | "not"  <query>
                 | <query> "where" "{" <condition> "}"
                 | "in" "{" <in-resource> "," <ns-query> "}"
                 | "in" "[" <in-resource> "," <ns-query> "]"
                 | "Fail"
                 | "(" <query> ")" .
```

Query Terms

```
<qr-term>       ::= <qr-ns-label> (<ordered-qr-list> | <unordered-qr-list>)
                  | <var> ("->" <qt-term>)?
                  | <identifier> "@" <qr-term>
                  | "optional" <qr-term>
                  | "desc" <qr-subterm>
                  | "(" <qr-term> ")" .
```

Query Subterms, Query-Subterm Lists

```
<qr-subterm>    ::= <qr-ns-label> (<ordered-qr-list> | <unordered-qr-list>)
                  | <string>
                  | <number>
                  | <float>
                  | <regexp>
                  | <var> ("->" <qr-subterm>)?
                  | "^" <identifier>
                  | <identifier> "@" <qr-subterm>
                  | ( "desc" | "optional" | "without"
                    | "position" ( <number> | <var> )  ) <qr-subterm>
                  | "(" <qr-subterm> ")" .


<ordered-qr-list> ::= "[" ( <ns-declaration>* <qr-subterm-list> )? "]"
                    | "[[" ( <ns-declaration>* <qr-subterm-list> )? "]]" .

<unordered-qr-list> ::= "{" ( <ns-declaration>* <qr-subterm-list> )? "}"
                      | "{{" ( <ns-declaration>* <qr-subterm-list> )? "}}" .

<qr-subterm-list> ::= <qr-attributes>? "," <qr-subterm> ( "," <qr-subterm> )* .
```

Query Attributes

```
<qr-attributes> ::= "attributes" "{" <qr-attribute> ( "," <qr-attribute> )* "}"
                  | "attributes" "{{" <qr-attribute> ( "," <qr-attribute> )* "}}"
                  | ( "desc" | "optional" | "without" ) <qr-attributes>
                  | "(" <qr-attributes> ")" .

<qr-attribute>  ::= <qr-ns-label> ( "[" ( <string> | <var>  | <regexp> ) "]"
                                 | "{" ( <string> | <var>  | <regexp> ) "}" )
                  | <qr-ns-label> "{{" "}}"
                  | <var> ("->" <qr-subterm>)?
                  | "^" <identifier>
                  | <identifier> "@" <qr-subterm>
                  | ( "desc" | "without" | "optional" ) <qr-attribute>
                  | "(" <qr-attribute> ")" .
```

DATA TERMS

```
dt = data
```

Labeles, Prefixes

```
<dt-ns-label>   ::= (<dt-ns-prefix> ":")? <dt-label> .
<dt-ns-prefix>  ::= <identifier> | <string> .
<dt-label>      ::= <identifier> | <string> .
```

Data Terms

```
<ns-dt-term>    ::= <ns-declaration>* <dt-term> .

<dt-term>       ::= ( <identifier> "@" )? <dt-ns-label>
                             (<ordered-dt-list> | <unordered-dt-list>).
```

Data Subterms, Data Subterm Lists

```
<dt-subterm>    ::= <dt-term>
                  | <string>
                  | <number>
                  | <float>
                  | "^" <identifier>
                  | <identifier> "@" <dt-subterm> .

<ordered-dt-list> ::= "[" <dt-subterm-list>? "]" .
<unordered-dt-list> ::= "{" <dt-subterm-list>? "}" .

<dt-subterm-list> ::= <dt-attributes>? dt-subterm> ( "," <dt-subterm> )* .
```

Data Term Attributes

```
<dt-attributes> ::= "attributes" "{" <dt-attribute> ( "," <dt-attribute> )* "}" .
```

```
<attribute>     ::= <dt-ns-label> ( "[" (<string> "]" | "{" (<string> "}" ).
```

References

[ABBS83] S. O. Anderson, R. C. Backhouse, E. H. Bugge, and C. P. Stirling. An assessment of locally least-cost error recovery. *The Computer Journal*, 26(1):15–24, 1983.

[AH99] John Aycock and R. Nigel Horspool. Faster generalized LR parsing. In *International Conference on Compiler Construction (CC 1999)*, volume 1575 of *Lecture Notes in Computer Science (LNCS)*, pages 32–46, Amsterdam, March 1999. Springer.

[ASU86] A.V. Aho, R. Sethi, and J. D. Ullman. *Compilers : principles, techniques and tools*. Addison Wesley, 1986.

[BDM$^+$93] U. Bianchi, P. Degano, S. Mannucci, S. Martini, B. Mojana, C. Priami, and E. Salvatori. Generating the analytic component parts of syntax-directed editors with efficient error recovery ,. *Journal of Systems and Software*, 23:65–79, 1993.

[BF87] Michael G. Burke and Gerald A. Fisher. A practical method for lr and ll syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems*, 9(2):164–197, April 1987.

[Cer02] Carl Cerecke. Repairing syntax errors in lr-based parsers. In *CRPITS '02: Proceedings of the twenty-fifth Australasian conference on Computer science*, pages 17–22, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[Cha91] Philppe Charles. *A Practical method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis, Department of Computer Science, New York University, May 1991.

[dJKV99] Merijn de Jonge, Tobias Kuipers, and Joost Visser. Hasdf: A generalized lr-parser generator for haskell. January 1999.

[DJM01] Chris Dornan, Isaac Jones, and Simon Marlow. *Alex User Guide*, 2001.

[DMM88] P. Degano, S. Mannucci, and B. Mojana. Efficient incremental lr parsing for syntax-directed editors. *ACM TOPLAS*, 10(3):345–373, 1988.

[DP95] P. Degano and C. Priami. Comparison of syntactic error handling in lr parsers. 1995.

[Fer04] Joao Fernandes. Generalized lr parsing in haskell. Technical Report DI-PURe-04.11.01, Departamento de Informatica da Universidade do Minho Campus de Gualtar – Braga – Portugal, October 2004.

[GJ90] Dick Grune and Cerriel J.H. Jacob. *Parsing Techniques: a practical guide*. Ellis Horwood, Chichester, 1990.

[GR75] Susan L. Graham and Steven P. Rhodes. Practical syntactic error recovery ,. *Commun. ACM*, 18(11):639–650, November 1975.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[K9797] *Lucky Manual*, July 1997.

[Knu65] Donald Ervin Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[Lei01] Daan Leijen. Parsec, a fast combinator parser. Technical report, University of Utrecht – Dept. of Computer Science, October 2001.

[Lju03] Peter Ljungl"of. An abstract view of generalized lr parsing (extended abstract). September 2003.

[MG01] Simon Marlow and Andi Gill. *Happy User Guide*, 2001.

[MYV95] Bruce J. McKenzie, Corey Yeatman, and Loraine De Vere. Error repair in shift reduce parsers. *ACM Transactions on Programming Languages and Systems*, 17(4):672–689, July 1995.

[NS00] M.-J. Nederhof and G. Satta. Left-to-right parsing and bilexical context-free grammars. *6th Applied Natural Language Processing Conference and 1st Meeting of the North American Chapter of the Association for Computational Linguistics*, 2:272–279, April 2000.

[Obr98] Davor Obradovic. Structuring functional programs by using monads, May 1998.

[PD78] Thomas J. Pennello and Frank DeRemer. A forward move algorithm for lr error recovery. *Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 241–254, 1978.

[PHF99] John Peterson Paul Hudak and Joseph Fasel. *A Gentle Introduction to Haskell 98*, October 1999.

[RD78] G. David Ripley and Frederick C. Druseikis. A statistical analysis of syntax errors. *Computer Languages*, 3(4):227–240, 1978.

[SB95] James P. Schmeiser and David T. Barnard. Producing a top-down parse order with bottom-up parsing technical report 95-378. Technical Report 95-375, Department of Computing and Information Science Queen's University, Kingston, Canada, March 1995. http://www.cs.queensu.ca/TechReports/Reports/1995-378.pdf.

[Sch92] Uwe Schöning. *Theoretische Informatik – kurzgefasst*. Spektrum Akademischer Verlag, 1992.

[Sch04] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, LMU, 2004.

[SFB05] S. Schaffert, T. Furche, and F. Bry. Initial draft of a possible declarative semantics for the language xcerpt. *REWERSE Deliverable I4-D4*, April 2005.

[SSS83] Seppo Sippu and Eljas Solisalon-Sioninen. A syntax-error-handling technique and its experimental analysis. *ACM TOPLAS*, 5(4):656–679, 1983.

[Tho99] Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison Wesley, 1999.

[TN91] M. Tomita and S.-K. Ng. The generalized lr parsing algorithm. In M. Tomita, editor, *Generalized LR Parsing*, pages 1–16. Kluwer, Boston, 1991.

[Wad92] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM Press.

[WM92] R. Wilhelm and D. Maurer. *Übersetzerbau – Theorie, Konstruktion, Generierung*. Springer-Verlag, 1992.