Tik-76.168 Tietämystekniikan erikoistyö: ROCC, Robust Compiler Compiler

Jari Arkko
Ti IV 32701c

1989-1-31

Contents

Chapter 1

Scope of the work

The purpose of this work is to study error correction methods in parsing. These methods are mainly targeted towards programming language implementations, but they can easily be applied in e.g. command interpreters. One general method built in a LALR(1) parser generator is implemented and its properties are studied. This method extends the LR(1) parser machine for error correction and it is directed by rules specific to the languages. Also other issues related to error correction, like the generation of accurate error messages, are studied.

The properties of the selected method are studied using a parser for a structured programming language Xe. Conclusions about the usability of the selected methods are drawn from the experiences.

Chapter 2

Parsers and parser generators

Parser. *Parser is a program which takes string of tokens of some language as input and checks the syntactic correctness of the input. As a side-effect, a descriptor of the input is created. This descriptor is called the parse tree (most programming languages form recursive structures that can easily represented as trees).*

Grammar. *Every programming language has syntactic structure which is described by syntactic rules. The syntax of programming language constructs can usually be described by context-free grammars.*

For a compiler writer the use of grammars is the only suitable choice due to the following facts [2, p. 159]:

- A grammar is clear and compact specification of the syntactic structure. It also reduces problems in changing the language later.
- Grammars can be divided in to classes according to their properties. Some classes of grammars allow automatic generation of efficient parsers from the grammar. This generation is done by a parser generator.
- Possible errors in the syntax of the language are found easily when grammar is processed by a parser generator.
- The grammar of a language can be used for syntax directed translation of the input to e.g. object code.

A parser is usually constructed of two phases: in the first phase the lexical level of the input is analyzed and in the second phase the actual syntax of the input (composed of the lexical units called tokens of the input) is analyzed. The tokens usually represent elements like operators, keywords and identifiers.

The syntactic level of a programming language has usually recursive definition whereas the lexical level does not. An example is expression in a programming language: "expression is a variable, constant or two expressions separated by an operator". Most recursive languages can be modelled by context-free grammars. A context-free grammar consists of a number of terminals, a number of nonterminals, a start symbol and a number of productions.

Terminal. *Terminal is a token of the input, e.g. an identifier. Statements of the language consists of strings of these terminals.*

Nonterminal. *Nonterminal is a syntactic variable denoting a set of terminal strings. For instance, expression is a nonterminal in a typical programming language. A nonterminal is defined by a set of productions.*

Start symbol. *Start symbol is the "top-level" of the language. It is usually a nonterminal. For instance, a command language might have as its start symbol command.*

Production. *Production consists of the left hand side (LHS) and the right hand side (RHS). The LHS is a nonterminal (in context-free grammars) and the RHS is string of terminals and nonterminals. There may be more than one production with the same LHS. Informally, production represents one possible choice for a nonterminal.*

Derivation. *A derivation is a transformation of a string of terminals and nonterminals to a string of terminals and nonterminals where some nonterminal has been replaced by some RHS of a production having the the nonterminal as its LHS. Applying derivations consequtively, a string of terminals and nonterminals can be derived to a string of terminals.*

Rightmost derivation. *A derivation in which only the rightmost nonterminal of the derivation is replaced at each step [2, p. 169].*

Context-free grammar. *A context-free grammar is a grammar consisting of a set of terminals, nonterminals, productions of the form described above and a start symbol. Exactly all the statements of the language must be deriveable from the start symbol. Thus terminal strings denoted by the start symbol are exactly the statements of the language. Context-free grammars are not able to specify certain context-dependent rules in languages. For instance, they cannot check that a variable is declared when it is used [9, p. 39].*

Shift-reduce parsing. *Shift-reduce [2, p. 195] parsing attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top). We can think of this process as one of "reducing" a string w to the start symbol of a grammar. At each* reduction *step a particular substring matching the right side of a production is replaced by the symbol on the left of that production, and if the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse.*

LR parser. *LR parsing is an efficient, bottom-up syntax analysis technique that can be used to parse a large class of context-free grammars. The technique is called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" is for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions [2, p. 215].*

LALR parser. *LALR (lookahead-LR) [2, p. 236] is a variation of the LR technique. This method is often used in practice because the tables obtained by it are considerably smaller than the canonical LR tables, yet most common syntactic constructs of programming can be expressed conveniently by an LALR grammar.*

2.1 The LR parsing algorithm

LR parser consists [2, p. 216] of an input, an output, a stack, a driver program, and a parsing table that has two parts: action and goto. The driver program (parser machine) is same for all LR parsers; only the parsing tables change from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form $s_0 X_1 s_1 X_2 s_2$ ... $X_m s_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a symbol called state. Each state symbol summarizes the information contained in the stack below it, and the combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shift-reduce parsing decision. In an implementation, the grammar symbols need not appear on the stack; however, we shall include them in our discussion.

The parsing table consists of two parts: a parsing action function action and goto function goto. The program driving the LR parser behaves as follows. It determines s_m, the state currently on top of the stack, and a_i, the current input symbol. It then consults action[s_m, a_i], the parsing action table entry, which can have one of four values:

- Shift s, where s is a state. A new state is pushed on the stack.
- Reduce by a grammar production. The grammar symbols on the stack correspond to the right hand side of the production. These symbols and the states corresponding to them are taken away from the stack and the reduction is performed. A new state is pushed on the stack. This state is determined by using the goto table and the state on top of the stack after the reduction.
- Accept signals acception of the input. The grammar symbol for the parse tree is now on top of stack.
- Error signals an error.

In practice the parser also contains another stack to store the lexical values of tokens and partial results of the parse tree. These values can be thought as corresponding to the grammar symbols.

Chapter 3

Methods for error correction

The input of a parser is not always correct according to the syntactic rules of the language. This makes the task of writing parsers much harder than it would be if there were no errors.

This problem would be eased if we were allowed to terminate the parsing in the first syntax error. However, most of the time this is not possible. The user of the compiler or command interpreter may want to have

- Detailed information of the occurred error. Sometimes (e.g. in the UNIX C compiler) it is very hard to figure out the actual reason for the error. One reason for this is that one erroneous token in the input may cause the syntax error much later in the input string.
- If the error is small enough, the system should try to correct it automatically to avoid e.g. recompilation.
- The rest of the input should also be processed to check the possible errors in it.

The error handling and correction should also conform to the other requirements of the system: it should not slow down the processing of correct input significantly. Sometimes a single error generates an avalanche of errors right after the point of the original error. This is due to inappropriate recovery of the parser or simply due to that the erroneous input tried to start a new language construct which failed due to the error and this in turn led to the misunderstanding of the rest of the structure.

Possible errors in the input string can be divided e.g. in the following classes:

1. Lexical errors occur in the tokenization process before actual parsing takes place. An example of this is an unknown operator.
2. Syntactic errors occur in the parsing, such as missing operator.
3. Semantic errors, such as type errors.
4. Logical errors, where the algorithm itself is erroneous.

An example of erroneous program in a block-structured abstract datatype language (Xe) is represented in figure 1.

```
fact = proc(n:int) returns(word)
  if n == 1 then return(0)
       ^ Illegal operator (lexical)
                        ^ Should be 1 (logical)
            else return(n*fact(n-1))
                         ^ Type mismatch: word/int (semantic)
  end
end

main = proc()
  putn(:stream$primary_otput(),fact*10))
               ^ Should be primary_output (semantic)
                                 ^ Should be ( (syntactic)
end
```

Figure 3.1: An erroneous program.

3.1 No recovery

The simplest form of error recovery is abortion of the parsing at the point of an error. The input did not match the rules of the language and it was discarded. Usually no results (e.g. parse trees) can be acquired from such parse.

This kind of error recovery cannot usually be tolerated. Some command language parsers might not need to recover from an error but programming language parsers have to. The problem in aborting further parsing is that other errors will not be found before the error is corrected.

3.2 Hand-coded error recovery

The designer of the parser may have some insight of the typical errors in the input. This knowledge often extends to the frequency of these errors. It is possible to create a set of rules for the erroneous inputs. These rules will then be used as an error is seen in the input. The rules may recover from the error nicely or even correct it.

How these rules are expressed and what things are considered in the error recovery process depends largely on the type of the parser. E.g. for generated parsers the rules might include typical erroneous inputs.

However, this method applies only to typical errors and all other errors would be poorly handled even if they were simple. And it would also be difficult to ensure that the error recovery or error correction does not induce any other errors.

The problem with this approach is that it requires knowledge of the errors. Programming resources are also needed to create the code that will handle the errors.

3.3 Panic mode recovery

Panic mode recovery is a popular error recovery method since it is simple to implement even in generated parsers. When an error is detected by a parser it throws away input tokens and stack contents until an input token and the stack state permit parsing to continue. The usual approach to implementing panic mode involves defining a set of synchronizing tokens. These tokens are firm delimeters in the language. Usually, firm delimeters end statements or structured constructs and only minimally constrain the input which may follow. A typical firm delimeter in Pascal, for example, would be the semicolon.

The YACC LALR(1) parser generator implements a variant of the panic mode error recovery scheme. In this variant there are error productions in the grammar of the language. These productions are of the following form:

```
A -> error alfa
```

When an error is seen in the input the parser throws away stack contents until it finds a stack state which is associated with an error production. Then the parser acts as if it had seen the special token error in the input and continues to parse the (often empty) string of terminals and nonterminals alfa. The action associated with the error production is called at a successful parsing of the error production. Typically the action would skip the input until it finds a firm delimiter applicable in this context.

The parser writer has the option of using his intuitions concerning the likely causes of errors when including error productions in the grammar.

3.4 Corrective methods

Contrary to [8] the previously described recovery methods are not considered satisfactory here since they - by definition - only detect the error and possibly recover from it to continue parsing at some of the following parts in the input. However, they can only point out the error in the input but they cannot give any likely causes for the error. Also, they are not able to correct the error even if it would be critical for the rest of parsing or compilation process to succeed. In this section we will look closely at three different error correction methods.

A corrected input does not only mean that the parsing can continue undisturbed. It also means that the correction, if given to the user, is likely to be a clear representation of the error. Compare, for example, the error messages 'Syntax error' and 'Missing semicolon'. However, there is also the other side to this issue: if the correction made is a wrong one the user will very likely be confused. In this respect it would be unfortunate if we would make too eager corrections to the input.

3.4.1 Graham-Hayley-Joy method

The Graham-Hayley-Joy method [5] uses a two phase approach: in the first phase correction is tried and if it succeeds, parsing continues. If no correction can be made the second phase recovers from the error using the error production method.

Error correction of the Graham-Haley-Joy method is based on the Graham-Rhodes method [6]. In the Graham-Rhodes method the left context information is condensed [9, p. 389] after the error has been detected and then a forward move is made. In the Graham-Hayley-Joy method the left context condensation is automatically performed by the default reductions of a LALR parser.

As the error is detected the possible corrections are collected before the forward move. The possible corrections are insertions, exchanges and deletions in the current state or in the previous state if it performed a shift and no reductions have been made since. The set of corrections is inferred from the parsing tables. This reduces the number of possible corrections and thus the time spent in the relatively slow forward move is minimized.

The forward move is done by trying the corrections by making a correction and forcing the parser ahead for a bounded number of tokens. The success of running the parser is recorded for each correction. Only a subset of the compiler is used during this operation to avoid undoing the changes made by the semantic actions of the parser. The parsing may be aborted due to an error before the bound on the number of tokens. The costs of the corrections (given by the user supplied routines) are multiplied by the amount of progress into the right context. Using these numbers the correction is then selected and parsing continues.

Also semantic information can be used in deciding the best correction in this method.

3.4.2 Burke-Fischer method

The Burke-Fischer method [4] consists of three phases. Simple recovery phase corrects single token errors. Scope recovery recovers from errors in nested structures e.g. statement blocks. Secondary recovery discards text that preceeds, follows or surrounds the error token.

We shall look more closely only at the simple error correction method. The error token (where the error was detected) is not necessarily the token that is in error. Consider the following Pascal declaration:

```
PROCEDURE Factorial(x:Integer; VAR fact: Integer): Integer;
```

In this example the keyword "PROCEDURE" was used where "FUNCTION" seems to be intended. Whatever the intention of the programmer, simple repair from "PROCEDURE" to "FUNCTION" corrects the syntax error. However, the error token is the colon that follows the right paranthesis and the point of error detection follows the occurrence of the error by 12 tokens. This forces the error correction to back up to the actual point of error before correcting the input.

Unfortunately, maintaining general mechanism for undoing parsing is too costly. In [3] a method of deferring reductions to the point where shift would occur is presented. Constructing k parallel parsers using this technique enable the parser to defer actions of k sequences of reductions and the k-1 shifts occuring between them. (The first parser parses the input but performs no semantic actions, the second is always one token behind and so on).

Using the parse action defarral mechanism the Burke-Fischer method is able to correct errors within range of k tokens from the point of error.

3.4.3 Global correction

There is a problem with the two previous methods. It is possible that a locally good correction will not be the right one when considering the whole input. This problem often exhibits itself when there are several errors in the input. The problem gets worse as the later passes of the language processor (e.g. passes of a compiler) handle errors. Some language have a “shallow” syntax in the sense that most of the rules of the language do not concern the syntactic level. E.g. most text inputs with equal number of left and right parantheses are syntactically correct Lisp programs. If the error correction is made too eagerly the semantics of the input might change even if we could produce a syntactically correct input.

In global error correction the whole input is taken as such and all corrections are made to it. The best correction is the correction which will minimize the sum of the cost functions associated with the corrections. Unfortunately, the best currently known algorithm, globally minimum distance model [1], is $O(n^3)$ where n is the length of the input and is thus considered impractical. There have been several attempts to take a middle way between local and global error correction, e.g. [7].

We should point out that a closest correct program may not be what the programmer had in mind. Nevertheless, the notion of global correction does provide a yardstick for evaluating error recovery techniques, and it has been used for finding optimal replacement strings for other error correction methods.

Chapter 4

Methods used in ROCC

Rocc is a LALR(1) parser generator.

4.1 Error correction

Rocc tries to correct errors in its input stream. Rocc parses normally until it sees an error, i.e. a state and input which does not lead to any reduction or shift operation. If the correction method fails, user-written error productions are used.

In the correction method the action tables are examined and they determine which tokens could have been in the place of the erroneous token so that the parsing can immediately continue. This is done both in the state the parser is currently and in the state after the last shift action.

The set of possible corrections, PC, is a set of tuples (S,T) where S is either the state of the parser in the last shift action or the current state of the parser. T is the token for which the action tables indicate succesful first action.

The set PC is not good for our purposes because of two reasons: first, it may have a large number of members due to the nature of the LALR(1) parsing and we would not know which one to select. Second and also due to the nature of the LALR(1) parsing the first action might succeed in the parsing but the next ones might not. This represents the case that the correction creates a head of some syntactic construct applicable at the current place in the input but the rest of the input does not match this construct. Thus we need to look forward in the input and make the decisions based also on the input.

All these reasons lead us into a second selection phase for the error correction. First, we define all the possible corrections that can be made in ROCC:

- Deletion of a erroneous token. We simply act just as if the token would not be in the input stream. Example: illegal character in the input stream.
- Insertion of a missing token. We insert a token in the input stream, possibly accompanied by its lexical value. Example: missing paranthesis.
- Exchanging a token into another. This correction is equal to successive deletion and insertion, but since ROCC does not support several corrections for a single error, an

exchange must be provided. Also, the cost functions (see below) for exchange are probably somewhat different from the sums of delete and insert cost functions.

Only a single correction is allowed for a single error (in the same region). The correction can be made either in the current state of the parser or in the last shift action state of the parser. Ideally, it should be possible to make any number of corrections in the current state of the parser or in any of its previous states. However, implementation of this would be too inefficient (see the chapter for the internal operation of the ROCC parser).

As we now know the possible corrections that a ROCC parser can make, we can proceed in to the second phase of the selection of the correction. One of the problems encountered in the first phase was that we didn't know which one of the corrections to select. This is solved by having static cost functions for the corrections. The cost of correction is calculated by the user given routines given the tokens that should be deleted, inserted or replaced. The cost functions may also indicate an infinite cost, i.e. that the correction cannot take place at all. This, in fact, is usually the case for most corrections.

The lexical value of the token does not play a part in the selection, regrettably since e.g. changing identifier "proctpye" to keyword "proctype" would be possible. However, since the parser needs the lexical values for the inserted or replaced tokens, a user-supplied routine calculates an appropriate lexical value when given the token. This method, however, does not work well except for keyword-like tokens.

As a result of the cost function evaluation, we have an ordered set of corrections DELETE(token), INSERT(token,lexvalue) and REPLACE(from,to,lexvalue). An obvious thing to do would be to select the least costly correction of this set and correct the input with it. However, preliminary studies showed that this method would almost never succeed in the correction due to the nature of the LALR(1) parsing (the LALR(1) parsing tables usually have many reduce actions that would reduce on almost any input and most errors would generate a too large set of corrections).

Therefore, the parser does not select the best correction by the cost functions but rather the best correction by the cost functions that succesfully parses the input. This means that the corrections are tried in the ascending cost order and the first one succesfully parsing the input is selected. This is called the forward move phase. By "succesful parsing" we do not mean parsing the whole input to the end but rather parsing succesfully the next N tokens. This is for two reasons: one, parsing the input to the end would be too inefficient and two, since we only allow one error in a region (now defined to be an area in the input consisting of N tokens), a second error in the input would prevent any correction from ever occurring. In short, error correction in ROCC means testing several ordered possibilities for the correction.

If the cost functions have been written carefully, then the selected correction is typically one of least costly corrections because the most frequent corrections have the lowest cost. Thus the correction is selected during the first few forward move tries. This lowers the cost of correcting the input compared to e.g. the Graham-Hayley-Joy method where all of the possible corrections must be tried. Of course, the ROCC method may lead to too eager correction.

How do the parsing actions (parse tree creation) relate to the error correction scheme? The parsing actions are turned off during the test parsing to avoid unnecessary work. Thus the actions are not executed at all during the test parsing. This leads to a problem after a successful parsing has been verified for a correction: the last N tokens have been read but the parsing actions for the reduced syntax rules have not been done and no parse trees have been created. Thus the parser machine backs up to the state where the correction started, makes the correction and proceeds normally.

What is the efficiency of this correction scheme? As we will see later, in keeping knowledge of the the last shift state there is a small linear overhead proportional to the number of reductions. Thus, the parsing of a correct input slows down a little. When an error is seen in the input, the first phase of the correction selection mechanism has an overhead proportional to the number of actions in the current and last shift state. This overhead is large compared to parsing a single token. When the parser tries the different corrections there is an overhead proportional to the number of possible corrections times N. This overhead is the largest induced in the ROCC correction scheme. For example, if the number of possible corrections is 100 and N is 4 then the overhead is close to the time spent in parsing an input of 400 tokens long, provided that all corrections must be tried. If the correction is found at, say, from the 10th correction then the overhead would be close to the time spent in parsing an input of 40 tokens long. It is clear, however, that if the number of errors in the input is large, the parser slows down significantly. However, the normal procedure in these cases is that the parser is aborted after some number of errors.

4.2 Information

The ROCC provides information of the errors occurred during parsing. There are two kinds of events that provide information. Detection of an error forces a call to a user-specified routine. This is used for generating error messages. The token that caused the error is given as an argument to the routine, as well as other information that might be usefull. If the error is then corrected, another user-specified routine is called. This is used to notify the user of the correction (and the nature of the error). The routine will be given as an argument a description of the correction (deletion of token X, insertion of token X or exchange of token X into Y).

4.3 Controlling the error correction

The error correction mechanism has several options that the user can define. These include:

- Maximum number N of tokens the forward move will consume (this is usually between 1 and 5).
- Cost function for deleting tokens.
- Cost function for inserting tokens.
- Cost function for replacing tokens.
- When the correction replaces or inserts a new token into the input, the lexical value of that token must be supplied. Often the lexical value does not matter, as for keywords and special characters, but e.g. with identifiers a lexical value is needed. If the cost functions allow insertion of tokens requiring lexical values the user must supply a routine which will create these values.
- Routine to be called at the point of an error.
- Routine to be called when a correction is made to the input.

- Routine *fail* that can be called from the semantic actions. It informs the parser of a semantic error. This way the error correction can reject erroneous corrections. It can also be used to reject even syntactically correct input and force an error correction.
- Routine *error-ok* that can be called from the semantic actions. It informs the parser that the current correction should be accepted since a major language element has been reduced and is thus complete (e.g. in Pascal reducing a statement). This can be used to prematurely accept a correction, e.g. when the size of the forward move region is large.

4.4 Internal operation

The parser operates normally like any other LALR parser. However, the error correction scheme requires some changes.

The ROCC error correction scheme requires undoing reductions after the last shift. This is due to the nature of the LALR parser that allows the parser to make reductions even when the current input symbol eventually signals an error. It is assumed that the semantic actions are free of side effects.

The parser stores information to be able to undo the effects of reductions to the state of the parser. This information is cleared when a shift occurs. Each reduction consumes a number of lexical value and state stack elements and produces one lexical value and state stack element. The changes from the state of the parser after the shift operation are recorded. For each reduction, the consumed lexical value and state stack are saved to a list. The state and lexical value produced by the reduction are pushed to the stacks. The pushed values are not saved during the next reductions; the saving is only done for the elements in the original stacks after the shift. This is implemented by calculating the number of elements created on the stack by the reductions. The saving introduces three new variables: *nReductionResults*, *SaveStateStack* and *SaveValueStack*. For each shift operation, these variables are initialized:

```
nReductionResults := 0
SaveStateStack := null_stack()
SaveValueStack := null_stack()
```

For each reduction, where *nReductionRHSComponents* is the number of element in the right hand side of the production about to be reduced and where *StateStack* and *ValueStack* are the state and value stacks of the parser, respectively, the below code must be executed:

```
for index := nReductionResults to nReductionRHSComponents do
    push(SaveStateStack,elementfrom_top(StateStack,index))
    push(SaveValueStack,elementfrom_top(ValueStack,index))
end
nReductionResults := nReductionResults + 1 -
                     min(nReductionResults,nReductionRHSComponents)
```

The saving slows the parser during the reductions. For each reduction, at most $2n$ stack elements must be saved, where n is the number of tokens or grammar symbols in the right hand side of the production for which the reduction was performed. However, the reduction itself must reference $2(n+1)$ elements. Thus the saving does not increase the complexity of the parsing algorithm and only adds the execution time by a small constant factor.

When an error is detected, the parser enters the error correction state. In the correction state the possible corrections are first generated from the parsing tables. The corrections are ordered using the cost functions. Corrections not possible are also ruled out by the cost functions. The corrections may be done at the current state of the parser or in the state after the last shift action.

When it comes necessary to restore the state of the parser to the state it was after the shift the parser stacks *StateStack* and *ValueStack* must be restored. This is done by removing the extra elements generated by the reductions and restoring the elements removed by the reductions:

```
PopStackNElements(StateStack,nReductionResults)
PopStackNElements(ValueStack,nReductionResults)
AppendStack(StateStack,SaveStateStack)
AppendStack(ValueStack,ValueStateStack)
```

If the error correction fails then the normal panic mode recovery scheme is adopted.

ROCC has also facilities for moving in the input stream. These facilities are used in the error correction forward move phase. The tokens read are saved to a list and this list can be used for input if the forward move fails.

4.5 Selection of the cost functions

The purpose of the cost functions is to 1) indicate which corrections are possible, 2) indicate an ordering between the corrections and 3) ease the task of finding the correction using the forward move. The cost functions should be selected so that at least typical errors are covered and that the cost associated with each correction approximates the size of the change to the input. This size can be determined e.g. by visual or logical similarity.

Careful examination of the syntax of the language and the typical errors is not the only way of constructing the cost functions. It is also possible to assign every possible correction the same cost. This would only violate points 2) and 3) above. It is actually required that the typical error pattern is not followed too rigidly. This is simply because the errors are not always typical. It is a major advantage of the forward move scheme that unexpected errors can be corrected. Thus the forward move should be given enough room to operate by not restricting the cost functions too much.

Chapter 5

Application: parser for the Xe language

5.1 Properties of the Xe syntax

Xe is a block structured and strongly typed language. Its syntax differs from other block structured languages in that it relies heavily on context. There are no semicolons or other statement separators. The nature of a statement is usually clear only in a rather late phase of the parsing.

Xe uses almost all of the ASCII characters including e.g. three kinds of parantheses. These are used in different contexts of the Xe language. This adds the number of possible errors that must be dealt with, though some of them are easy to recover from.

5.2 General operation

The parser reads the input using a scanner. The scanner yields tokens. There are over one hundred different tokens. When an error occurs during parsing, the parser calls a function which will record the point of the error. The parser then tries correcting the error using the methods described in chapter 4. If the correction succeeds the parser calls another function which will be given information of the correction. Using the earlier recorded point of the error and this information an error message is generated. If the correction does not succeed a fatal error message is generated.

Fatal error leads to a recovery process and a recovery function is called. The recovery function depends on the context of the error: either definition or statement recovery is tried depending on which one is the closest enclosing syntactic structure. The recovery works by removing tokens and parse states until the erroneous construct is considered to be ended and then allowing the parser to continue parsing. An error indicator is left in the parse tree at the place of the erroneous construct.

The definition recovery searches the input for definition patterns. A definition pattern in Xe is of the form "id = proc", "id = iter", "id = rule" or "id = datatype". It is assumed that when such pattern appears, it represents the next definition in the input. The current definition is marked erroneous and skipped. The input marker is set at the point of the identifier in the definition.

The statement recovery searches for common statement terminators. As there are no semicolons in Xe other signs must be used. A statement list usually ends in an "end". This must then mark the end of a statement. Skipping up to that point means typically that several statements must be skipped. However, there are no alternatives.

When more than 10 fatal errors are detected the parsing is aborted bacause the input seems to have a major problem and the parser cannot succesfully continue. This also prevents the comparatively costly error correction process to take too much time trying to correct large number of errors.

5.3 Cost functions

The deletion, insertion and exchange operations of tokens are assigned costs. These costs are calculated by the cost functions. The cost functions were derived from typical syntax errors and properties of the Xe language syntax. The correction operations were targeted towards the following kinds of syntax errors:

- Missing or extra parantheses or **begin-end** blocks, including premature end of input without enough closing parantheses.
- Mismatching of parantheses, e.g. closing one kind of paranthesis with another kind of paranthesis.
- Extra input, e.g. control characters in the input.
- Missing or extra commas or other common syntactic separators.
- Common misunderstandings of the Xe language, e.g. a missing **do** keyword in the **while** statement.
- Exchange of visually similar tokens, e.g. commas and dots.

Thus the deletion cost function assigns costs (in increasing order of cost) to illegal tokens, parantheses, **begin** token and **end** token. Identifiers can also be deleted but with a much higher cost. The insert cost function allows insertion of the same tokens as the deletion cost function, except for the identifier.

Exchange cost function assigns costs to exchanging syntactically similar characters, e.g. commas and dots or parantheses. Also common misunderstandings like exchanging := and = are corrected. Regrettably, there are no provisions for exchanging a misspelled keyword into the correct one.

5.4 Error messages

Fatal errors produce an error message "Fatal error in X" where X is the token where the error was detected. Corrected errors produce messages which state the token where the error was detected and the correction. The messages are of the forms “Error in X - removed”, “Error in X - inserted Y” and “Error in X - replaced by Y”, where X and Y are tokens. This kind of message gives surprisingly often the correct description for the error. The most important problem with this kind of messages is that no context information can be represented. For instance, the error message “Error in id - inserted do” deals with the token structure whereas “Error in while statement - missing do keyword inserted” explains the context information to the user.

The Xe programming environment lets the user browse through the syntax and semantic errors in the editor. Using commands of the environment the cursor is moved to the point of the error.

5.5 Experiences

Below is an example of an erroneous input for the Xe parser. The error messages are shown under the erroneous lines. Notice the first fatal error message which forced the parser to skip to the start of the next routine (*fact*). Also notice the corrections to the errors with unbalanced parantheses.

```
%-*- Mode: XE -*-

hello = proc()
     @*&^%@*$#&^
%Fatal syntax error at @
     if a = 2 then
        a := 1
     end
end

fact = proc(n:word) returns(word)
        if n = 0 then return(1
        else return(n*fact(n-1)) end
%Non-fatal syntax error at ELSE - inserted )
        end
end
%Non-fatal syntax error at END - removed

fib = proc(n:word returns(word)
%Non-fatal syntax error at RETURNS - inserted )
        putl(:stream$primary_output() "Fib " || unparse(:n)
%Non-fatal syntax error at string - inserted ,
        if n < 2 then
%Non-fatal syntax error at IF - inserted )
                return(#1)
%Non-fatal syntax error at illegal - removed
        else
                return(fib(n-2)+fib(n-1))
        end
%Non-fatal syntax error at eof - inserted END
```

The ROCC correction applied to the Xe parser performs quite well with respect to the requirements of section 5.3. The main problems with the error correction are:

- Several errors force the error correction to fail if they are too close together.
- The corrections are designed with a set of typical errors in mind. These errors might be atypical for some users or situations.
- Due to the nested structure of the Xe language the panic mode recovery does not always succeed in skipping to the next definition.

Chapter 6

Conclusions

6.1 Differences from other methods

The Graham-Haley-Joy method is very similar to the ROCC method. In some cases it can insert several tokens at the point of one error. However, it can only change the current input token (or the previous if there has been no reduce actions since the last shift action) when the error is detected. ROCC can back up to the last shift regardless of the reductions.

The Burke-Fischer method deferral mechanism is more general than the ROCC method of undoing the last reductions. However, the ROCC method is simpler and serves as a generalization of the Graham-Haley-Joy method. Disadvantage of the ROCC method is that the semantic actions must be free of side effects.

6.2 Results

The main results of the ROCC parser generator are 1) implementing yacc style parser generator for Lisp, 2) implementing a working error correction method, 3) developing a method for undoing parsing actions that will serve as a middle ground between the Graham-Hayley-Joy method and the Burke-Fischer method of parse action deferral and 4) developing an interface from the semantic actions to the parser that allows to process even semantic errors using the error correction mechanism.

6.3 Usability

The ROCC method can be applied for any LR(1) parser. There is no need for modifications to the action tables of parser. The action tables can be used as such. The state machine execuiting the parser tables must, however, be changed, but it is so also for other corrective methods. Also, the ROCC method does not require special properties from the language parsed.

Cost functions and the error signaling routines are the user interface of the ROCC. The signaling routines are obviously necessary for every kind of parser. In ROCC, these routines convey information of the erroneous tokens and corrective actions. Some parsers do not convey any information at all, and ROCC is better compared with them. Some parsers generate even the error messages from the grammar and corrections. ROCC does not do this, but the generation of simple error messages is not usually hard, as could be seen from the example in the previous chapter. The quality of the error messages may suffer from this. However, it is considered that the ROCC error message mechanism fills most of the requirements for a typical programming language parser.

The cost functions can be written trivially and in this respect it is not a major task for the parser writer, if so wanted. Even writing good cost functions is not a large project, as can be seen from the example in the previous chapter. The process of writing cost functions does often takes place as the parser is used and typical errors noted. Writing the cost functions does not require knowledge of the parsing process. It is thus concluded that the cost function interface is rather good from the user's point of view.

Bibliography

[1] A. Aho, T. Peterson: *A minimun distance error correcting parser for context free languages.* SIAM Journal of Computing 1, 4, pp. 305-312 (1972).

[2] A. Aho, R. Sethi, J. Ullman: em Compilers: Principles, Techniques and Tools. Addison-Wesley 1986.

[3] M. Burke, G. Fischer: *A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery.* Proceedings of the SIGPLAN 82 Symposium on Compiler Construction (June 23-25, 1982, Boston). ACM, New York, 1982, pp. 67-78.

[4] M. Burke, G. Fischer: *A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery.* ACM Transactions on Programming Languages and Systems, Vol. 9, No. 2, April 1987, pp. 164-197.

[5] S. Graham, C. Haley, W. Joy: *Practical LR Error Recovery.* Communications of the ACM, 1979.

[6] S. Graham, S. Rhodes: *Practical Syntactic Error Recovery.* Communications of the ACM 16, 11 (Nov. 1975), pp. 639-650.

[7] J. Mauney, C. Fischer: *A Forward Move Algorithm for LL and LR parsers.* Communications of the ACM, 1982.

[8] H. Richter: *Noncorrecting Syntax Error Recovery.* ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, July 1985, Pages 478-489.

[9] J. Tremblay, P. Sorenson: *The Theory and Practice of Compiler Writing.* McGraw-Hill, 1985.

Appendix A

ROCC Manual

Appendix B

Xe Syntax and Parser