Excel@FIT 2015
http://excel.fit.vutbr.cz

Error Recovery during Top-Down Parsing: Acceptable-sets derived from continuation

Alena Obluková*

Abstract
Parser is one of the most important parts of compiler. Syntax-Directed Translation is often used, that means that parser controls: semantic actions and generation of syntax tree. When the input contains an error, parser cannot continue and the whole compiler has to stop. Therefore, it is important that parser supports good error recovery. Error recovery requires parser to be modified with heuristic, so syntax analyzer can continue parsing even though an error is detected. There are several error-recovery strategies and methods. This paper describes one of them - acceptable-sets derived from continuations - continuation in LL parsers. This method is not so well known, however it is easy to explain and implement. We have implemented parser that uses this error recovery method. This method has been modified to be more encapsulated and clearer. Implemented parser can be used in lectures to demonstrate error recovery in top-down parser.

Keywords: syntax analysis — LL grammars — error recovery

Supplementary Material: Code attached

*xobluk00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

There will always be an error in program code. Missing semicolon, extra or missing bracket. Without error recovery, when a first error in the input occurs, the parser stops. In that case, programmer cannot see all errors in the program, only the first one. The purpose of error recovery methods is to put parser into a state, in which it can continue parsing, at the same time reporting error comprehensibly to help the programmer. Even with error recovery in place, the input can't be completely fixed, since parser cannot know, what the programmer wanted to write. None of error recovery method can fix the input, it only enables parser to continue parsing. There are four types of programming errors:

- *Lexical errors* - include misspelling of keywords, identifiers (e.g. whille instead of while)
- *Semantic errors* - include mismatch between operator and operands (e.g. integer to float)
- *Logical errors* - the program syntax may be correct, however, it may not reflect, what the programmer wanted to achieve
- *Syntactic errors* - when parser doesn't have any grammatical rule, which it can use (e.g. use of semicolon instead of comma, missing operator) [1].

In this paper, only syntactic errors are discussed, so the term error means syntactic error [1]. Goals of error recovery are:

- report the error clearly and accurately
- recover quickly enough to detect subsequent errors
- not to report non-existent errors [1]

There are two approaches: top-down parsing and bottom-up parsing. Not every method can be used in both approaches. Methods can be divided into several classes, depending on the level on which they handle the error. *Global error handling methods* use global context, the complete input. The most known method is *least-error-correction*, which is efficient, nevertheless too complicated to implement. *Phase-level recovery methods* use only part of context. Methods, which use only the actual parser state, are called *local error handling methods*. This class is the largest; *Panic mode*, *Follow-set error recovery* and *Acceptable-sets derived from continuation* all belong here. *Suffix methods* don't use any context. Lastly, *Ad Hoc methods* do not really form a class, they also do not use any context. Sum and substance of these methods is to change grammar in such a way that the parser does not even recognize that the input contains an error [2].

Method discussed in this paper is called Acceptable-sets derived from continuations. It combines quick error recovery, small memory needs, easy implementation with quick and accurate reporting of errors. This method has been used because it is suitable for demonstration of error recovery in Top-Down parser. Each step of the method has been illustrated on a simple grammar.

2. Predictive Parsing

There are two approaches how to implement Top-Down parser. Recursive-Descent Parsing and Predictive Parsing. Predictive parsing is not so commonly used, but is easier to demonstrate error recovery in Top-Down parser. Predictive parsing uses LL table. Terms, such as LL table, First Set, grammar, are used in this paper. All thess terms are described in book Formal Languages and Computation [3]. Implemented parser use simple grammar. This grammar is not Turing complete, but can be used to illustrate common errors. Whole grammar consists of 15 rules. Only 9 rules from grammar are written here.

1. $\langle PROGRAM\rangle \rightarrow begin\langle BODY\rangle end$
2. $\langle BODY\rangle \rightarrow \varepsilon$
3. $\langle BODY\rangle \rightarrow \langle STATEMENT\rangle;\langle BODY\rangle$
4. $\langle TERM\rangle \rightarrow int$
5. $\langle TERM\rangle \rightarrow id$
6. $\langle STATEMENT\rangle \rightarrow id = \langle EXPRESSION\rangle$
7. $\langle EXPRESSION\rangle \rightarrow \langle TERM\rangle\langle EXP\rangle$
8. $\langle EXP\rangle \rightarrow -\langle EXPRESSION\rangle$
9. $\langle EXP\rangle \rightarrow \varepsilon$

Non-terminals in this grammar are *PROGRAM*, *BODY*, *STATEMENT*, *TYPE*, *TERM*, *EXPRESSION* and *EXP*. Terminals are *begin*, *end*, *;*, *read*, *id*, *write*, *string*, *int*, =, *(*,*)*,+ and -. Each input has to has its end. In literature the $ sign is used, in this grammar the terminal *end* represent the ending symbol. There are two ways how to implement LL parser. Recursive-Descent Parsing and Predictive Parsing. In this paper, Predictive Parsing is used, since it is easier to understand. LL-table is needed in Predictive Parsing. It is constructed using Empty set, First set, Follow set and Predict set. The construction of LL table can be found in Formal Languages and Computation [3].

3. Error recovery

In this paper local error handling methods are mentioned. Most of local error handling methods use *Acceptable set*.

3.1 Acceptable set

Acceptable set is a set that is calculated from the parser state, when an error occur. After detecting error, symbols from the input are skipped until a symbol from the input is found as a member of Acceptable set. Local error handling methods differ in how they calculate Acceptable set.

The most known local error handling method is *Follow-set error recovery* method. At Faculty of Information Technology this method is also known as *Hartmanova metoda zotavení z chyb*. The method is easy to explain and easy to implement. It uses the First and Follow sets that together makes the Acceptable set.

3.2 Acceptable-sets derived from continuation

Acceptable-sets derived from continuation is a local error handling method. It uses very interesting and effective approach. The biggest difference between this method and other local error handling methods is that this method uses two grammars. Supposed that when error is detected, part of the input has already been processed. This part is a prefix u of a sentence uv. Uv is a sentence in language, so the only thing that has to be done, is to find string v, that is continuation of u. The continuation computes as follows:

Continuation is a sentence, which has to be produced using the fewest productions steps. The trick is to find the quickest end for each nonterminal, in other words to find which right-hand side leads the fastest to string composed only of terminals. It is possible to compute it in advance, using a number called step count. Each terminal has step count of 0. Nonterminals have step count set to infinity. Each right-hand side has step count equal to sum of all its members. Left-hand side has count number equal to right-hand side count+1. If this number is less then the previous step count of nonterminal, step count of nonterminal is updated. This process repeats until none of the step counts changes. Proper grammar should have all count steps less than infinity.

Here are few steps of creating continuation grammar. Numbers in brackets represent step counts [2]

$\langle PROGRAM\rangle[\infty] \rightarrow begin[0]\langle BODY\rangle[\infty]end[0]$
$\langle BODY\rangle[\infty] \rightarrow \varepsilon[0]$
$\quad|\langle STATEMENT\rangle[\infty];[0]\langle BODY\rangle[\infty]$
$\langle STATEMENT\rangle[\infty] \rightarrow id[0] = [0]\langle EXPRESSION\rangle[\infty]$
$\langle TERM\rangle[\infty] \rightarrow int[0]$
$\quad|id[0]$
$\langle EXPRESSION\rangle[\infty] \rightarrow \langle TERM\rangle[\infty]\langle EXP\rangle[\infty]$

$\langle EXP\rangle[\infty] \rightarrow -[0]\langle EXPRESSION\rangle[\infty]$
$\quad|\varepsilon[0]$

$\langle PROGRAM\rangle[\infty] \rightarrow begin[0]\langle BODY\rangle[\infty]end[0]$
$\langle BODY\rangle[1] \rightarrow \varepsilon[0]$
$\quad|\langle STATEMENT\rangle[\infty];[0]\langle BODY\rangle[\infty]$
$\langle STATEMENT\rangle[\infty] \rightarrow id[0] = [0]\langle EXPRESSION\rangle[\infty]$
$\langle TERM\rangle[1] \rightarrow int[0]$
$\quad|id[0]$
$\langle EXPRESSION\rangle[\infty] \rightarrow \langle TERM\rangle[\infty]\langle EXP\rangle[\infty]$
$\langle EXP\rangle[1] \rightarrow -[0]\langle EXPRESSION\rangle[\infty]$
$\quad|\varepsilon[0]$

For each nonterminal (left-hand side) the right-hand side with lowest step count is picked. If there is no single lowest step count number among the right-hand sides, the author of the parser has to pick one rule (in this paper $\langle TERM\rangle[1] \rightarrow id[0]$ is not used). These rules form a continuation grammar. Continuation grammar doesn't have to form a proper grammar, it is used only for error recovery.

1. $\langle PROGRAM\rangle \rightarrow begin\langle BODY\rangle end$
2. $\langle BODY\rangle \rightarrow \varepsilon$
3. $\langle STATEMENT\rangle \rightarrow id = \langle EXPRESSION\rangle$
4. $\langle TERM\rangle \rightarrow int$
5. $\langle EXPRESSION\rangle \rightarrow \langle TERM\rangle\langle EXP\rangle$
6. $\langle EXP\rangle \rightarrow \varepsilon$

This grammar is called continuation grammar and will be used in continuation in an LL parser. When error occurs, parser has to call error recovery function that continues parsing in error mode. As long as parser is not recovered, parser in error mode use continuation grammar instead of normal grammar.

3.3 Continuation in an LL parser

Continuation grammar is already set, so the continuation is easy to compute - when an error occurs, rules from continuation grammar are used. Now it is necessary to determinate acceptable set. Every time the nonterminal is derived, all terminals, which are produced, end in the acceptable set. Moreover, every time a nonterminal is replaced by its right-hand side from continuation grammar, First set of this nonterminal is added to acceptable set [2].

Now, all the important things are known, so an algorithm can be written. The method works as follows [4]:

1. **replace nonterminal**: First nonterminal on the stack is found. Using only continuation grammar, the nonterminal is replaced by its right-hand side. Each terminal from the right-hand side of the rule is added to the acceptable set. Also, the First set of derived nonterminal is added to the the acceptable set.
2. **skip unacceptable tokens**: Zero or more tokens form the input are skipped, until a symbol from acceptable set is found. Since ending symbol (in this paper it is terminal end) is acceptable, this step is not infinite.
3. **resynchronize the parser**: Parser tries to continue. If it is possible to make a move, parsing can normally continue. If parser can't continue, modified parser has to continue as follows:
    - **nonterminal on the top of the stack**: If nonterminal is on top of the stack and there is no grammar rule to use, algorithm repeats from 1.
    - **terminal on the top of the stack**: If there is a terminal on top of the stack and it cannot be matched with the input symbol, expected symbol is inserted. Since every token has its value, implicit value is given to each inserted symbol (e.g. 0 to int, err_id to id). Step 3 is repeated until the parser is resynchronized.

We propose a modification of the algorithm. In step 3, symbol on top of the stack is not popped immediately. Parser only tries to match the terminal with the input symbol or only tries to find a grammar rule for the nonterminal. If it is possible to make a move, error recovery was successful. Move itself is made by parser in normal mode. If the move is not possible, error recovery is not finished and continues depending on what is on top of the stack - step 1 for nonterminal, step 3 for terminal. The advantage of the proposed approach is that when move is possible, it is clear that the parser has reached correct state and therefore has recovered from the error. The second advantage of this approach is, that when terminal is popped or nonterminal replaced by its right-hand site, error recovery is successfully ended. However, what if another error on the input occurs? Immediately after error recovery the parser cannot continue and error recovery has to stop. When the parser in error mode only checks if the move can be made, no error immediately after recovery can happen.

Proposed modification only uses continuation grammar for error recovery and unlike the original algorithm, normal grammar is only used for test and not for the move itself. The complexity is slightly higher but the error recovery mechanism is encapsulated and more clear.

Suppose that there is a short program with error as shown in figure 1

```
begin
     a == b;
end
```

Figure 1. Short program with error

Sequence of tokens is: *T_begin T_id T_equal T_equal T_id T_semicolon T_end.*

Error is on the second line: extra *equal*. The recovery with my modification works as follows:

Parser starts parsing the input.

stack	input	rule
$\langle PROGRAM\rangle$	T_begin	1
$end\langle BODY\rangle begin$	T_begin	POP
$end\langle BODY\rangle$	T_id	3
$end\langle BODY\rangle;\langle STATEMENT\rangle$	T_id	6
$end\langle BODY\rangle;\langle EXPRESSION\rangle = id$	T_id	POP
$end\langle BODY\rangle;\langle EXPRESSION\rangle =$	T_equal	POP
$end\langle BODY\rangle;\langle EXPRESSION\rangle$	T_equal	ERROR

There is no grammar rule that can be used. Parser cannot continue, error recovery has to start.

Replace nonterminal: First terminal, $\langle EXPRESSION\rangle$, is found. It is replaces by its right-hand side, using continuation grammar:
$\langle EXPRESSION\rangle \rightarrow \langle TERM\rangle\langle EXP\rangle$,
$\langle TERM\rangle \rightarrow int$,
$\langle EXP\rangle \rightarrow \varepsilon$
Acceptable set = { *(, int, id* }.

stack	input	rule
$end\langle BODY\rangle;int$	T_begin	1

Skip unacceptable tokens: In the input is *T_equal* that is not in acceptable set, token is skipped.
In the input is *T_id* that is in acceptable set. Token is not skipped.

stack	input	rule
$end\langle BODY\rangle;int$	T_int	1

Terminal on the top of the stack: Terminal *T_int* is on the top of the stack. It can be matched with the input.
Error recovery is finished, parsing can continue.

3.4 Implementation of error recovery

We have successfully implemented parser with this error recovery method as a console application. It expects two files on input: one file with tokens (terminals) and second, where used rules are written. Information about the analysis is written to standard output. Optional parameter is -n. With this parametr error recovery is turned on. This parameter helps users to compare parser with error recovery and without error recovery.

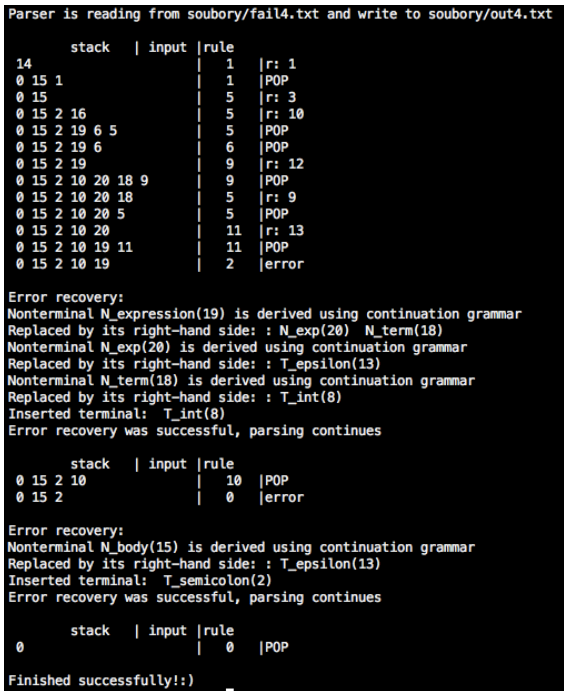
Suppose that there is a short program with error as shown in figure 2

```
begin
     a = (b+\;
     )
end
```

Figure 2. Short program with error

Sequence of tokens is: *T_begin T_id T_equal T_bracket_left T_id T_plus T_semicolon T_bracket_right T_end.*

Output from my application using Acceptable-sets derived from continuation method to deal with these errors can be seen on screenshot 3.



```
Parser is reading from soubory/fail4.txt and write to soubory/out4.txt

       stack   | input |rule
 14                    |   1   |r: 1
 0 15 1                |   1   |POP
 0 15                  |   5   |r: 3
 0 15 2 16             |   5   |r: 10
 0 15 2 19 6 5         |   5   |POP
 0 15 2 19 6           |   6   |POP
 0 15 2 19             |   9   |r: 12
 0 15 2 10 20 18 9     |   9   |POP
 0 15 2 10 20 18       |   5   |r: 9
 0 15 2 10 20 5        |   5   |POP
 0 15 2 10 20          |   11  |r: 13
 0 15 2 10 19 11       |   11  |POP
 0 15 2 10 19          |   2   |error

Error recovery:
Nonterminal N_expression(19) is derived using continuation grammar
Replaced by its right-hand side: : N_exp(20)  N_term(18)
Nonterminal N_exp(20) is derived using continuation grammar
Replaced by its right-hand side: : T_epsilon(13)
Nonterminal N_term(18) is derived using continuation grammar
Replaced by its right-hand side: : T_int(8)
Inserted terminal:  T_int(8)
Error recovery was successful, parsing continues

       stack   | input |rule
 0 15 2 10             |   10  |POP
 0 15 2                |   0   |error

Error recovery:
Nonterminal N_body(15) is derived using continuation grammar
Replaced by its right-hand side: : T_epsilon(13)
Inserted terminal:  T_semicolon(2)
Error recovery was successful, parsing continues

       stack   | input |rule
 0                     |   0   |POP

Finished successfully!:)
```

Figure 3. Screenshot of application run from terminal

4. Conclusions

Parser is one of the main parts of compiler. When error in the input occurs, parser without error recovery has to stop. When parser stops, the whole compiler has to stop. This is unfortunate, because only the first error is detected. Errors can be lexical, discovered by lexical analyzer, semantic, discovered by semantic analyzer and syntactic, discovered by syntactic analyzer - parser. In this paper, only syntactic errors are discussed.

There are many error recovery methods. They are divided into classes, depending on level on which they handle the error.

Acceptable-sets derived from continuation method described in this paper belongs to the largest class: local error handling methods. Acceptable-sets derived from continuation is a method, which is not commonly described. However, it is easy to explain so it can be used in lessons to demonstrate how error recovery in Top-Down Parsers works. When an error in input occurs, program cannot be compiled. Advantage of this method is, that error recovery is as fast as possible and that it reports errors effectively.

It would be very interesting to compare this method with other methods more deeply. Also this method could be implemented in LR parsers. Since this method is implemented only as a console application, it would be great to implement graphical application for better demonstration in lectures.

Acknowledgements

I would like to thank my supervisor Prof. RNDr. Alexander Meduna, CSc. for all his help with with this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley, 2nd ed., 2007. ISBN: 0321486811.

[2] D. Grune and C. J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer; 2nd ed., 2008. ISBN: 1441919015.

[3] Alexander Meduna. *Formal Languages and Computation*. Taylor & Francis Informa plc, 2014. ISBN: 978-1-4665-1345-7.

[4] D. Grune, K. van Reeuwijk, E. H. Bal, C. J. H. Jacobs, and K. Langendoen. *Modern Compiler Design*. Springer; 2nd ed., 2012. ISBN: 1461446988.