This book is licensed under a Creative Commons Attribution 3.0 License

7. Syntax analysis

Learning objectives:

- syntax is the frame that carries the semantics of a language
- syntax analysis
- syntax tree
- top-down parser
- syntax analysis of parenthesis-free expressions by counting
- syntax analysis by recursive descent
- recursive coroutines

The role of syntax analysis

The syntax of a language is the skeleton that carries the semantics. Therefore, we will try to get as much work as possible done as a side effect of syntax analysis; for example, compiling a program (i.e. translating it from one language into another) is a mainly semantic task. However, a good language and compiler are designed in such a way that syntax analysis determines where to start with the translation process. Many processes in computer science are syntax-driven in this sense. Hence syntax analysis is important. In this section we derive algorithms for syntax analysis directly from syntax diagrams. These algorithms reflect the recursive nature of the underlying grammars. A program for syntax analysis is called a *parser*.

The composition of a sentence can be represented by a *syntax tree* or *parse tree*. The root of the tree is the start symbol; the leaves represent the sentence to be recognized. The tree describes how a syntactically correct sentence can be derived from the start symbol by applying the productions of the underlying grammar (Exhibit 7.1).

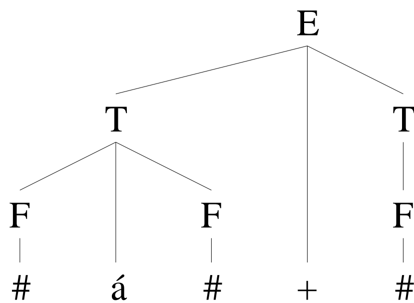


Exhibit 7.1: The unique parse tree for # · # + #

Top-down parsers begin with the start symbol as the goal of the analysis. In our example, "search for an E". The production for E tells us that we obtain an E if we find a sequence of T's separated by + or –. Hence we look for T's. The structure tree of an expression grows in this way as a sequence of goals from top (the root) to bottom (the leaves). While satisfying the goals (nonterminal symbols) the parser reads suitable symbols (terminal symbols) from left to right. In many practical cases a parser needs no backtrack. No backtracking is required if the current

input symbol and the nonterminal to be expanded determine uniquely the production to be applied. A recursive-descent parser uses a set of recursive procedures to recognize its input with no backtracking.

Bottom-up methods build the structure tree from the leaves to the root. The text is reduced until the start symbol is obtained.

Syntax analysis of parenthesis-free expressions by counting

Syntax analysis can be very simple. Arithmetic expressions in Polish notation are analyzed by counting. For sake of simplicity we assume that each operand in an arithmetic expression is denoted by the single character #. In order to decide whether a given string $c_1\, c_2 \ldots c_n$ is a correct expression in postfix notation, we form an integer sequence t_0, $t_1, \ldots, t_n$ according to the following rule:

$t_0 = 0$.
$t_{i+1} = t_i + 1$, if $i > 0$ and c_{i+1} is an operand.
$t_{i+1} = t_i - 1$, if $i > 0$ and c_{i+1} is an operator.

Example of a correct expression:

```
#  #  #  #  −  −  +  #  ·

c1 c2  c3 c4 c5 c6 c7 c8 c9
t0 t1  t2 t3 t4 t5 t6 t7 t8 t9

     0  1  2  3  4  3  2  1  2  1
```

Example of an incorrect expression (one operator is missing):

```
     #    #    #     +     ·    #    #   /
     c1   c2   c3    c4    c5   c6   c7  c8

t0   t1    t2   t3   t4    t5    t6   t7   t8
0    1     2    3    2     1     2    3    2
```

Theorem: The string $c_1\, c_2 \ldots c_n$ over the alphabet A = { # , + , − , · , / } is a syntactically correct expression in postfix notation if and only if the associated integer sequence $t_0, t_1, \ldots, t_n$ satisfies the following conditions:

$t_i > 0$ for $1 \le i < n$, $t_n = 1$.

Proof ⇒: Let $c_1\, c_2 \ldots c_n$ be a correct arithmetic expression in postfix notation. We prove by induction on the length n of the string that the corresponding integer sequence satisfies the conditions.

Base of induction: For n = 1 the only correct postfix expression is c_1 = #, and the sequence $t_0 = 0$, $t_1 = 1$ has the desired properties.

Induction hypothesis: The theorem is correct for all expressions of length ≤ m.

Induction step: Consider a correct postfix expression S of length m + 1 > 1 over the given alphabet A. Let $s = (s_i)_{0 \le i \le m+1}$ be the integer sequence associated with S. Then S is of the form S = T U Op, where 'Op' is an operator and T and U are correct postfix expressions of length j ≤ m and length k ≤ m, j + k = m. Let $t = (t_i)_{0 \le I \le j}$ and $u = (u_i)_{0 \le i \le k}$ be the integer sequences associated with T and U. We apply the induction hypothesis to T and U. The sequence s is composed from t and u as follows:

This book is licensed under a Creative Commons Attribution 3.0 License

$s = s_0, s_1, s_2, \dots, s_j, s_j + 1, s_j + 2, \dots, s_m, s_{m+1}$

$t_0, t_1, t_2, \dots, t_j, u_1 + 1, u_2 + 1, \dots, u_k + 1, 1$

$0, \dots, 1, \dots, 2, 1$

Since t ends with 1, we add 1 to each element in u, and the subsequence therefore ends with $u_k + 1 = 2$. Finally, the operator 'O_p' decreases this element by 1, and s therefore ends with $s_{m+1} = 1$. Since $t_i > 0$ for $1 \le i < j$ and $u_i > 0$ for $1 \le i < k$, we obtain that $s_i > 0$ for $1 \le i < k + 1$. Hence s has the desired properties, and we have proved one direction of the theorem.

Proof ⇐: We prove by induction on the length n that a string $c_1 c_2 \dots c_n$ over A is a correct arithmetic expression in postfix notation if the associated integer sequence satisfies the conditions stated in the theorem.

Base of induction: For n = 1 the only sequence is $t_0 = 0, t_1 = 1$. It follows from the definition of the sequence that $c_1 = \#$, which is a correct arithmetic expression in postfix notation.

Induction hypothesis: The theorem is correct for all expressions of length ≤ m.

Induction step: Let $s = (s_i)_{0 \le i \le m+1}$ be the integer sequence associated with a string $S = c_1 c_2 \dots c_{m+1}$ of length m + 1 > 1 over the given alphabet A which satisfies the conditions stated in the theorem. Let j < m + 1 be the largest index with $s_j = 1$. Since $s_1 = 1$ such an index j exists. Consider the substrings $T = c_1 c_2 \dots c_j$ and $U = c_j c_{j+1} \dots c_m$. The integer sequences $(s_i)_{0 \le i \le j}$ and $(s_i - 1)_{j \le i \le m}$ associated with T and U both satisfy the conditions stated in the theorem. Hence we can apply the induction hypothesis and obtain that both T and U are correct postfix expressions. From the definition of the integer sequence we obtain that c_{m+1} is an operand 'O_p'. Since T and U are correct postfix expressions, $S = T\,U\,O_p$ is also a correct postfix expression, and the theorem is proved.

A similar proof shows that the syntactic structure of a postfix expression is unique. The integer sequence associated with a postfix expression is of practical importance: The sequence describes the depth of the stack during evaluation of the expression, and the largest number in the sequence is therefore the maximum number of storage cells needed.

Analysis by recursive descent

We return to the syntax of the simple arithmetic expressions of chapter 6 in the section “Example: syntax of simple expressions” (Exhibit 7.2). Using the expression # · (# – #) as an example, we show how these syntax diagrams are used to analyze any expressions by means of a technique called *recursive-descent parsing*. The progress of the analysis depends on the current state and the next symbol to be read: a lookahead of exactly one symbol suffices to avoid backtracking. In Exhibit 7.3 we move one step to the right after each symbol has been recognized, and we move vertically to step up or down in the recursion.

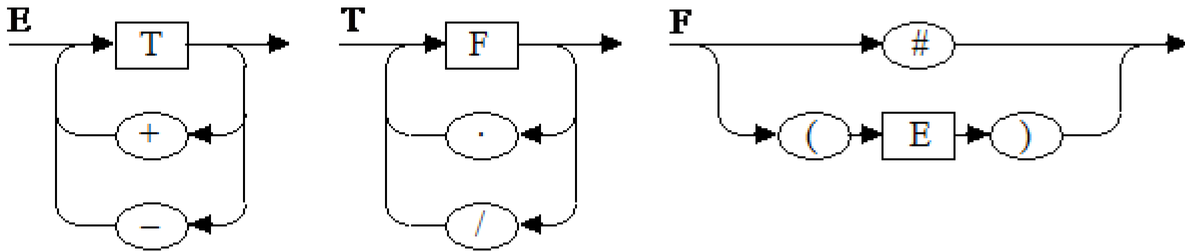


Exhibit 7.2: Standard syntax for simple arithmetic expressions (graphic does not match)

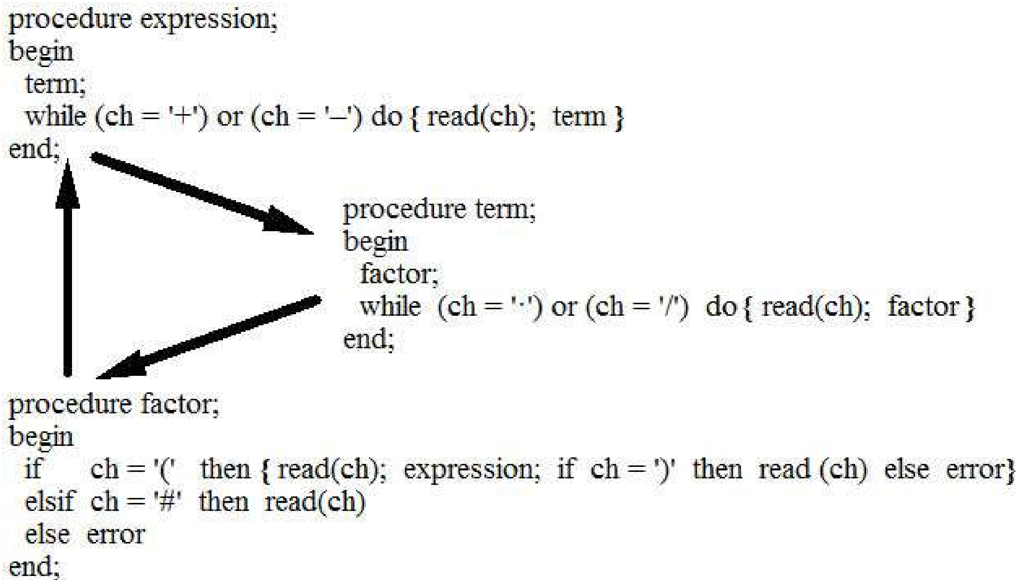
Exhibit 7.3: Trace of syntax analysis algorithm parsing the expression # · (# – #).

Turning syntax diagrams into a parser

In a programming language that allows recursion the three syntax diagrams for simple arithmetic expressions can be translated directly into procedures. A nonterminal symbol corresponds to a procedure call, a loop in the diagram generates a while loop, and a selection is translated into an if statement. When a procedure wants to delegate a goal it calls another, in cyclic order: E calls T calls F calls E, and so on. Procedures implementing such a recursive control structure are often called *recursive coroutines*.

This book is licensed under a Creative Commons Attribution 3.0 License

The procedures that follow must be embedded into a program that provides the variable 'ch' and the procedures 'read' and 'error'. We assume that the procedure 'error' prints an error message and terminates the program. In a more sophisticated implementation, 'error' would return a message to the calling procedure (e.g. 'factor'). Then this error message is returned up the ladder of all recursive procedure calls active at the moment.



Before the first call of the procedure 'expression', a character has to be read into 'ch'. Furthermore, we assume that a correct expression is terminated by a period:

```
…
read(ch);  expression;  if  ch ≠ '.'  then  error;
…
```

Exercises

1. Design recursive algorithms to translate the simple arithmetic expressions of chapter 6 in the section “Example: syntax of a simple expressions” into corresponding prefix and postfix expressions as defined in chapter 6 in the section “Parenthesis-free notation for arithmetic expressions”. Same for the inverse translations.
2. Using syntax diagrams and EBNF define a language of 'correctly nested parentheses expressions'. You have a bit of freedom (how much?) in defining exactly what is correctly nested and what is not, but obviously your definition must include expressions such as (), ((())), (()(())), and must exclude strings such as (,)(, ())().
3. Design two parsing algorithms for your class of correctly nested parentheses expressions: one that works by counting, the other through recursive descent.