What every computer scientist should know about syntax and syntax tools

Lothar Schmitz

lothar@informatik.unibw-muenchen.de

August 2004

1. Example and Overview

Do you know exactly what happens if in one of your programs the line

```
get(Temp);
```

is executed? (We assume that `Temp` has been declared as a Float variable and that the user of your program types in `-47.11E1`, completing input by hitting the "return" key.)

Obviously, the following tasks have to be performed:

1. Check whether the input character sequence `-47.11E1` actually represents an admissible Float number value.
2. In case it is, the number's constituents (sign, mantissa, exponent) have to be identified.
3. Finally, the binary internal representations of the number's constituents have to be computed.

For the first task, we need an unambiguous definition of what character sequences we are going to accept as representations of floating point number constants. Technically speaking, we have to **define the language** of Float constants. The second task requires us to determine the contant's **syntactical structure** by parsing it into its constituents: sign, mantissa (including the position of the decimal point), and exponent. The third task is to **translate** the character string into the internal number format used by a computers: typically some fixed length bit strings. This conversion will be based on the result of the second task: First each of the constituents is converted. The partial results are then combined to yield the complete internal representation.

In a similar way natural language translation is taught in language classes: First (analogous to steps 1 and 2 above) the syntactical structure of the given text is determined. This we call **analysis**. Along the lines of this syntactic structure the constituents of the text are translated and then built into complete target language sentences (see step 3 above). This is called **synthesis**. In the translation of natural languages we can distinguish two different levels: translation of single words and translation of complete sentences. Analogously we distinguish between **lexikal** and **syntactic analysis** when translating sentences from artificial languages (e.g., programming languages). Artificial computer languages differ from natural languages in that they have neither conjugation nor declination.

In the following sections we shall use the language of arithmetic expressions as our **running example**. In some cases, we shall use the even simpler language of "balanced" strings — strings that can be obtained from arithmetic expressions by stripping off everything except for parentheses. E.g., from the arithmetic expression `(17-(a-1)) * (x/y)` we obtain the balanced string `((())()` .

For describing syntax we rely on well established notational frameworks: Lexical structures are described by *regular expressions*, syntactic structures by *context-free grammars*. Using the arithmetic expressions example we refresh relevant notions.

For lexical analysis we derive from regular expressions the corresponding acceptors, i.e., *finite automata*. The transformation method we employ was first described by McNaughton and Yamada. Though seldom used it is very intuitive. Also, it introduces the "dot marker" technique also found in LR parsing. When applying *various different* finite automata to one character input stream in order to recognize *tokens* of different classes (e.g., variable names, number constants, operator symbols, parentheses) priority issues have to be resolved. Lexical analysis results in a token sequence which is passed on to syntax analysis.

The purpose of syntax analysis is to transform the given token sequence into a *syntax tree* which is built according to the rules of the given context-free grammar. Ambiguous grammars possibly assign more than one syntax tree to a given token sequence and thus complicate the translation process. Therefore, only unambiguous grammars are used in practice. We take a look at common parsing algorithms: LL(1) top down and different bottom up strategies (canonical LR, SLR(1), and LALR(1)).

Attribute evaluation assigns one or more *attributes* to the nodes of the syntax tree — nodes correspond to grammar symbols. Attributes are "user defined" information introduced only for use in the intended translation. An *attribute grammar* essentially consists of a set of *attribute evaluation rules* that describes how an attribute is evaluated either directly or from the values of "neighbouring" attributes. An *attribute evaluation strategy* evaluates all attributes in one or more syntax tree traversals. Using one attribute grammar we translate arithmetic expressions into equivalent machine instruction sequences. Another attribute grammar serves to generate Kantorovic tree representations (text graphics) from arithmetic expressions.

The techniques presented here are well suited for a wide spectrum of tasks. Virtually all languages whose syntax can be defined unambiguously by context-free grammars may be translated or transformed otherwise by our tools. This applies to objects as different as chemical formulae, data base contents, games of chess, XML documents, Postscript files, LaTeX texts etc.

Finally, we name important sources and syntax tools. Available tools comprise compiler compilers, programming languages with built-in regular expression processing, syntax based editors, and compiler frameworks for XML applications.

2. Describing Syntax

We consider arithmetic expressions made up from number constants, variables, parentheses, and operator symbols as in

```
a*a - 2*a*b + b*b
```

and

```
(alpha + 17) * (beta - alpha /17)
```

On the lexical level we have to define exactly the syntax of number constants, variables, and other token types. For this purpose we use regular expressions (so-called `Pattern` definitions):

```
Patterns:
    addOpRA $+|$-
    multOpRA $*|$/
    openingParenthesisRA $(
    closingParenthesisRA $)
    letterRA {$a-$z}|{$A-$Z}
    digitRA {$0-$9}
    nameRA letterRA(letterRA|digitRA)[0-*]
    numberRA digitRA[1-*]
```

Each line introduces a named regular expression: First the **token name** is given, then its defining expression.

The simplest of regular expressions consist of one (ASCII) symbol only, e.g., `openingParenthesisRA` and `closingParenthesisRA`. The **escape** symbol `$` indicates that the ASCII symbol immediately following it (here the opening or closing parenthesis) is the content of the definition. Escape symbols and other **metasymbols** are part of the notational framework and do *not* belong to the language being defined.

The metasymbol | separates **alternatives**: Thus `addOpRA` stands for either a "plus" or a "minus" symbol.

A **range** like the one in `digitRA` describes a set of successive ASCII symbols. An equivalent (but rather tedious!) definition would have been:

```
    digitRA $0|$1|$2|$3|$4|$5|$6|$7|$8|$9
```

Repetition clauses (like the one `numberRA` is ending with) define the number of times the regular subexpression preceding it may be "traversed". The number on the left is interpreted as the lower bound of repetitions, the other number as the upper bound. The special upper bound $*$ stands for "unrestricted". Thus a "number" as defined by `numberRA` has at least one digit; there is no upper bound on its number of digits. (Note: Repetition clauses consist of metasymbols only!) The token definition `nameRA` for name tokens is based on the **auxiliary token** letterRA which defines single letters. Therefore, a name starts with a letter which is followed by any number (0 or more) of letters and/or digits.

The **token declaration** below establishes which kinds of tokens may appear in the token sequence to be passed to the parser and by what name (e.g., names may appear as `variable` tokens, opening parentheses as `oP` tokens):

```
Tokens:
    addOp addOpRA
    multOp multOpRA
    oP openingParenthesisRA
    cP closingParenthesisRA
    variable nameRA
    number numberRA
```

For the above example, lexical analysis will produce (next to the token name the original character input sequence is shown in funny brackets):

```
variable≪a≫
multOp≪*≫
variable≪a≫
addOp≪-≫
number≪2≫
multOp≪*≫
variable≪a≫
multOp≪*≫
variable≪b≫
addOp≪+≫
variable≪b≫
multOp≪*≫
variable≪b≫
```

and

```
oP≪(≫
variable≪alpha≫
addOp≪+≫
number≪17≫
cP≪)≫
multOp≪*≫
oP≪(≫
variable≪beta≫
addOp≪-≫
variable≪alpha≫
multOp≪/≫
number≪17≫
```

An obvious approach to defining the syntax of arithmetic expressions is to use the rule of grammar $G_{\text{arithsimple}}$ shown below:

$$E \;\rightarrow\; E\ addOp\ E \;\mid\; E\ multOp\ E \;\mid\; oP\ E\ cP \;\mid\; variable \;\mid\; number$$

This is an abbreviation for the more detailed notation

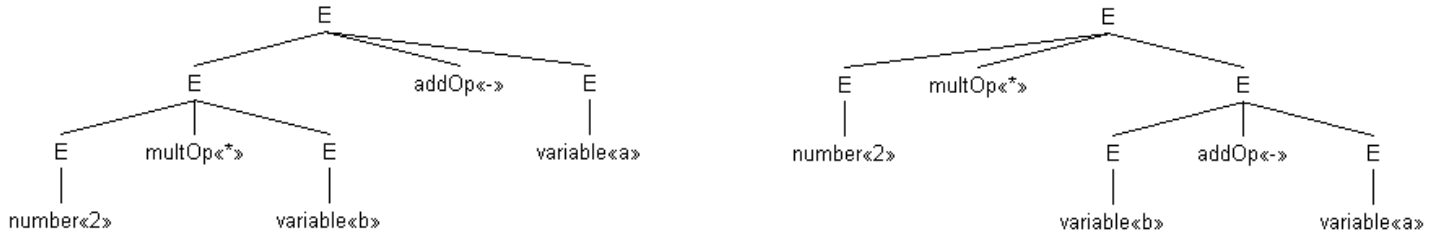
$$\begin{aligned} E &\rightarrow E\ addOp\ E \\ E &\rightarrow E\ multOp\ E \\ E &\rightarrow oP\ E\ cP \\ E &\rightarrow variable \\ E &\rightarrow number \end{aligned}$$

where each of the five production rules is on a line of its own.

Unfortunately, grammar $G_{\text{arithsimple}}$ is ambiguous. A simple examples shows that operator precedences are not always reflected correctly by the syntax. For

```
2 * b + a
```

we obtain two different syntax trees:



Obviously, in the syntax tree on the right hand side precedences are not reflected correctly!

(The pictures are screenshots of trees as generated by our tools. This guarantees a high degree of authenticity but may result in rather poor picture quality!)

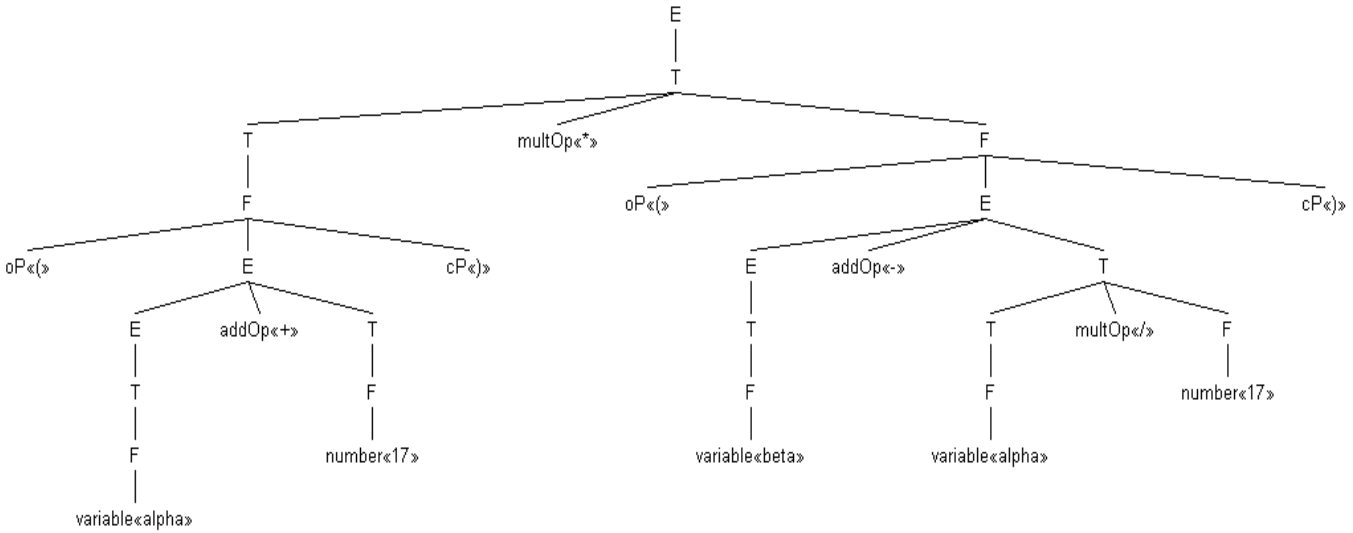
This is why one uses the well-known, if slightly more complex grammar G_{arith}:

$$
\begin{array}{rcl}
E & \rightarrow & E\ addOp\ T \\
 & | & T \\
T & \rightarrow & T\ multOp\ F \\
 & | & F \\
F & \rightarrow & oP\ E\ cP \\
 & | & variable \\
 & | & number
\end{array}
$$

Arithmetic expressions like our second example

```
(alpha + 17) * (beta - alpha /17)
```

have one syntax tree only — reflecting operator precedencies correctly:



Because of the loose coupling between lexical and syntactic analysis it is very simple to later on add "broken" number constants as used in the formula

```
2 * (3.14159 * r * r)
```

In order to do so we replace the definition

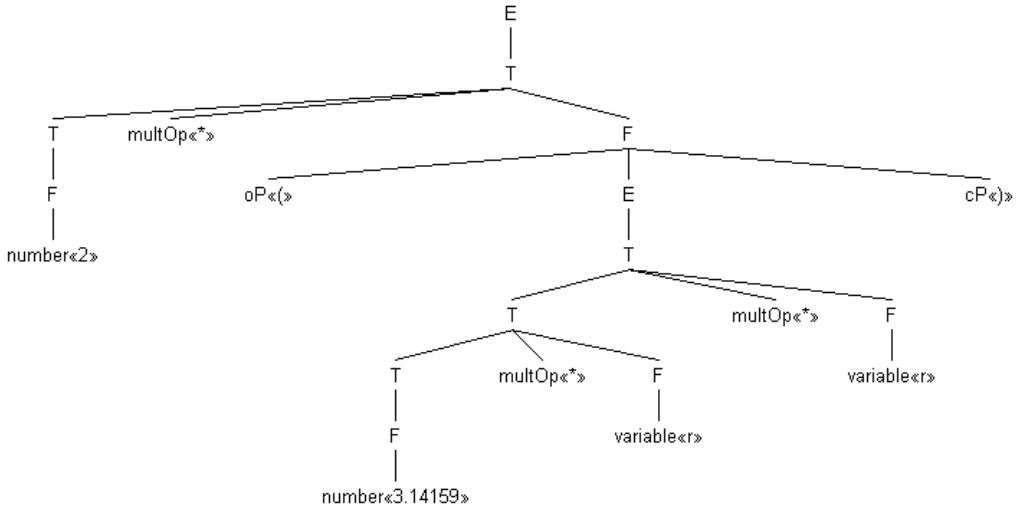
```
numberRA digitRA[1-*]
```

by a new one which allows for broken constanst as well

```
numberRA digitRA[1-*]($.digitRA[1-*])[0-1]
```

Lexical analysis applied to the above formula yields the token sequence and the syntax tree shown below

```
number≪2≫
multOp≪*≫
oP≪(≫
number≪3.14159≫
multOp≪*≫
variable≪r≫
multOp≪*≫
variable≪r≫
cP≪)≫
```



3. Lexical Analysis

By substituting in the definition of nameRA the names of the auxiliary expressions letterRA and digitRA by their contents we obtain the expanded regular expression below.

```
nameRA ({$a-$z}|{$A-$Z})(({$a-$z}|{$A-$Z})|{$0-$9})[0-*]
```

We demonstrate our technique for generating finite automata from regular expressions on this example. The basic idea is simple: Insert **marker points** into regular expressions to indicate progress of execution.

In the **start state** the marker point is at the beginning, i.e., to the left of the regular expression:

```
.({$a-$z}|{$A-$Z})(({$a-$z}|{$A-$Z})|{$0-$9})[0-*]
```

Without reading any input symbol the marker point may be propagated across the metasymbol (to the beginning of both (!) alternatives of the first letterRA (in its expanded form). Accordingly, we add two more marker points to the above start state to obtain:

```
.(.{$a-$z}|.{$A-$Z})(({$a-$z}|{$A-$Z})|{$0-$9})[0-*]
```

Notice that we do not "push marker points into ranges". Obviously, a marked subexpression like `.{$a-$c}` is meant to be an abbreviation for the more explicit form `.$a|.$b|.$c` (By the way: Technically this corresponds to Myhill's subset construction as applied to nondeterministic finite automata with ϵ transitions.)

Let X be any (lower or upper case) letter. Reading X in the above start state results in the new marker point position

```
({$a-$z}|{$A-$Z}).(({$a-$z}|{$A-$Z})|{$0-$9})[0-*]
```

which indicates that the first letterRA has been matched successfully with the input and that now the rest of the regular expression has to be matched with the following input symbols.

As in the start state marker points are propagated to all positions that can be reached without actually reading an input symbol. Thus we obtain the complete **X-successor** of the start state:

```
({$a-$z}|{$A-$Z}).((.{$a-$z}|.{$A-$Z})|.{$0-$9})[0-*].
```

Notice that because of the lower bound of the repetition clause a marker point appears at the end (right hand side) of the whole regular expression: This means that a nameRA may consist of one letter only. States with a marker point at the end are called **final states**.

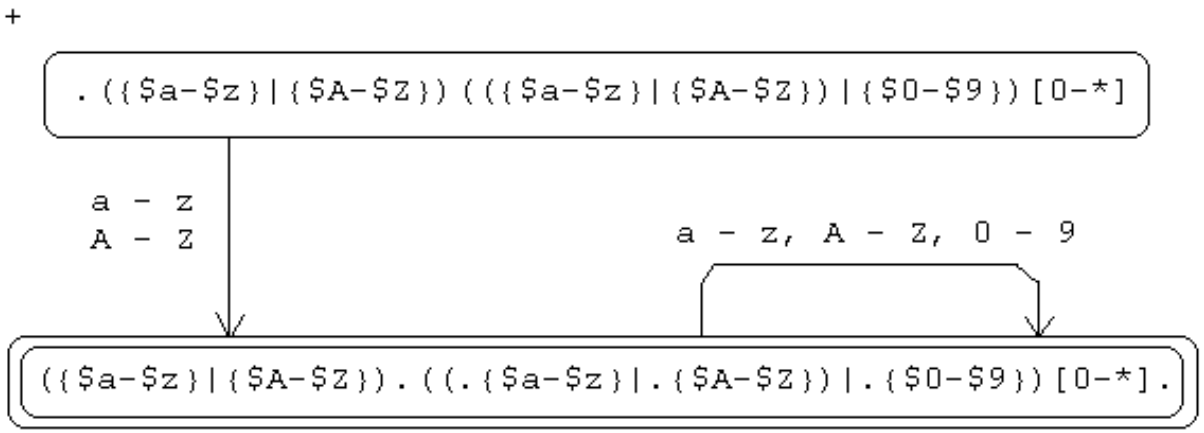
In state

```
({$a-$z}|{$A-$Z}).((.{$a-$z}|.{$A-$Z})|.{$0-$9})[0-*].
```

marker points immediately precede ranges of (lower case or upper case) letters and of digits. Because of the repetition clause and the propagation of marker points after reading any one of these symbols the same state is reached again.

Actually, the finite automaton corresponding to `nameRA` only has the two states shown above. While there is always exactly one start state, there may be more than one final state. In theory, the start state might also be one of the final states (allowing the empty string to be accepted). In the context of lexical analysis, however, this does not make sense.

In the **graphical representations** of finite automata below, the start state is always marked with a plus sign. Final states are marked with double frames, as usual.

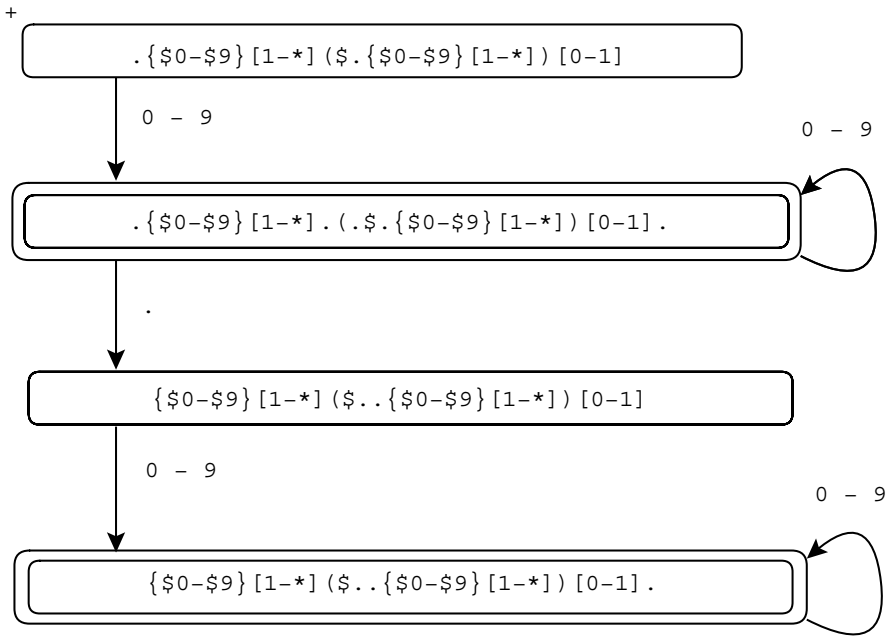


The marked regular expressions within states are shown to explain how the automaton was constructed; for processing strings, they are irrelevant. The behaviour of a finite automaton depends exclusively on the transitions between states and on the states being start or final states.

Now, let us apply the above automaton to the string `Z3D` (which can be interpreted to denote the non-existent Diesel variant of a well-known German sports car):

- Initially, the automaton is in its start state.
- The input symbol `Z` can be read because in the start state there is a transition arrow labeled with `Z`. This makes the destination of the transition arrow the new current state.
- In this state there are — among others — transitions labeled with `3` and with `D`. The transition arrows are loops leading to the same state again. So the next two input symbols can be processed analogously.
- After processing the complete input string `Z3D` the automaton is in a final state. Therefore, the input is **accepted** by the automaton, i.e., it belongs to the automaton's **language**.

For the broken number constants defined in the preceding section we obtain the following automaton (*Note:* The point after $ is a decimal point, in contrast to all the other points which are marker points.):



During lexical analysis the finite automata constructed from the different token definitions (i.e., regular expressions) are operating "concurrently" on the given input character sequence. Since different automata may have transitions for the same input symbol (e.g., digits in both identifiers and numbers, and letters in both key words and identifiers) it is not obvious which automaton gets to read which input symbol.

In our tools we have implemented the following set of **precedence rules** which we (and others) find convenient (however, other tools may arbitrarily use other conventions):

- Initially all automata are in their start states. They all attempt to read the next (first) input symbol. The same holds true whenever a token has been recognized (see below).
- Automata that in their current state do not have a transition for the next input symbol, are eliminated (for the moment) from processing. The other automata read the next input symbol and update their current state accordingly.
- After all automata have been eliminated the automaton that was last in a final state is declared "successful" (i.e., its token is recognized). Among different successful automata the one with the highest priority is chosen. (We use the convention that within the list of token definitions priorities increase.)
- That part of the input which starts at the symbol where all automata were last started up to (and including) the symbol which last brought the successful automaton into a final state is taken to be the recognized token string — it belongs to the token class of the successful automaton. All automata are started again on the input symbol immediately following the recognized token string.
- If the recognized token string is empty, lexical analysis stops with an error message.

This can be summarized more intuitively: *From the starting point we try to find the longest possible token string*. For programming languages priorities typically are required to distinguish between key words and variable identifiers.

Here, we should mention a feature characteristic of our tools which can also be used to resolve concurrency conflicts between token classes: Token recognition is combined with the evaluation of "start" and "end conditions" and the execution of "action blocks" (all of which may have side effects).

4. Syntax Analysis

For syntax analysis (or "parsing") we again use **marker points** to make visible the boundary between parts processed and parts still to be processed. Here, one marker point is inserted into the right hand side of a production rule to yield a so-called **item**. In order to better distinguish them from production rules, items often are embedded in brackets. E.g., the item $[A \rightarrow \alpha \,.\, \beta]$ belongs to the production rule $A \rightarrow \alpha\beta$.

For technical reasons, we add a new start symbol S' and a new start rule $S' \rightarrow S$ (where S is the original start symbol) to each grammar. Regarding syntax analysis we distinguish the following kinds of items (their names indicate the corresponding parsing **actions**):

- The **start item** $[S' \rightarrow \,.\, S]$ describes that we have just started the parsing process (the marker point is immediately to the left of the original start symbol). This item is the starting point for all parsing strategies.
- The start item is a special expansions item: An **expansions item** is an item $[A \rightarrow \lambda \,.\, B\ \rho]$, where the marker point immediately precedes a nonterminal symbol B. Therefore, while processing the right hand side of an A rule we next have to process nonterminal B. For this purpose, processing of the A rule is interrupted temporarily, and on a "lower" level processing continues with one or more items of the form $[B \rightarrow \,.\, \gamma]$.
- In a **shift item** $[A \rightarrow \lambda \,.\, b\ \rho]$ the marker point is immediately to the left of a terminal symbol b. In case the next input symbol is b, too, the input symbol b is thrown away and processing continues with item $[A \rightarrow \lambda\, b \,.\, \rho]$.
- A **reduction item** $[B \rightarrow \gamma \,.]$ indicates that the right hand side of the production rule has been processed successfully. Now processing on the "next higher" level is continued. On this level, processing of an item $[A \rightarrow \lambda \,.\, B\ \rho]$ was interrupted by expansion with $[B \rightarrow \,.\, \gamma]$ and is now (after "reducing" γ to B) continued with item $[A \rightarrow \lambda\, B \,.\, \rho]$.
- The **accept item** $[S' \rightarrow S \,.]$ is a special reduction item. Any successful bottom up analysis ends with this particular reduction: Then the right hand side of $S' \rightarrow S$, i.e., the original start symbol S has been processed completely.

All the practical parsing algorithms described here read the input string *exactly once* from left to right, without ever undoing any decision: For efficiency reasons, any form of backtracking is excluded.

During parsing the syntax tree is built and at the same time traversed from left to right. There are two main parsing strategies which differ in the order and the kind of actions (expand vs. reduce) that are used to actually build the syntax tree. **Shift actions** are performed by both strategies in the same way.

- In the **top down** strategy **expansions** are performed explicitly while reductions are skipped. Starting at the root the syntax tree (labeled with the start symbol of the grammar) is built up in expansion steps towards the leaves of the tree. Since in computer science trees are drawn with "the root in the sky" this strategy is called "top down".
- The **bottom up** strategy works the other way round: **reductions** are performed explicitly while expansions are skipped. Starting from the token sequence at the leaves the tree is built by inverse application of production rules (i.e., reductions) towards the root.

In a complete left-to-right tree traversal every subtree T is entered via an expansion ($[B \rightarrow \ . \ \gamma]$). Then T is traversed in a series of shifts and recursive traversals of T's subtrees. Finally, T is left via the reduction ($[B \rightarrow \gamma \ .]$) that corresponds to the expansion. *Notice that in every traversal expansion steps always precede their corresponding reduction steps!* Therefore, in a top down analysis decisions are due earlier (when expanding) than in a bottom up analysis (where decisions are due at reductions). Since decisions have to be taken only after seeing more of the input sequence, bottom up strategies are deterministically applicable to more grammars and thus more powerful than top down strategies.

Remember: Top down analysis is more natural, bottom up analysis is more powerful.

Parsing algorithms always use a **stack** of grammar symbols, however in different ways: During top down analysis the stack contains the tasks still to be carried out (i.e., the symbols to be expanded). During bottom up analysis the stack contains the tasks that have been completed already (i.e., the sequence of symbols the initial part of the input sequence has been reduced to so far).

Traces of parsing algorithms are written down as sequences of configurations, where a **configuration** is a triple (that also includes the `<action>` performed last):

```
( <stack> , <rest of input> , <action> )
```

The most important top down algorithm is **LL(1) parsing**:

The stack of an LL(1) parser initially contains what is to the right of the marker point in the start item $[S' \rightarrow . S]$, i.e., the original start symbol S which is to be expanded to the full syntax tree. Hence, the **start configuration** of an LL(1) parser is:

```
( S , <input string> , start )
```

In case the first and second components of a configuration start with the same terminal symbol `b`

```
( b <rest of stack> , b <rest of input> , ... )
```

a **shift** transition leads to the new configuration

```
( <rest of stack> , <rest of input> , shift "b" )
```

(In case the first and second components of a configuration start with *different* terminal symbols LL(1) parsing stops with an **error**.)

Now if the top (i.e., left hand) symbol of the stack is a nonterminal `B` and `b` is the next input symbol as in

```
( B <rest of stack> , b <rest of input> , ... )
```

then an **expansion** with a suitable production rule $B \rightarrow X_1X_2...X_n$ results in the successor configuration

```
( X1 X2 ... Xn <rest of stack> , b <rest of input> , expand "B -> X1 X2 ... Xn" )
```

How do we know which of the alternative rules for B is "suitable"? Recall that in the end the sequence of terminal symbols at leaf nodes (from left to right) must be identical to the given input symbol sequence. So by comparing B's alternatives to the next input symbols we have to decide which alternative to choose. "LL(1)" means that we may take into account only one input symbol when looking for the suitable alternative, i.e., the next input symbol b. Obviously, rule $B \rightarrow X_1X_2...X_n$ can be suitable only if $X_1X_2...X_n$ can produce a string starting with b. Fortunately, the set $first(X_1X_2...X_n)$ of terminal symbols with whom a string derived from $X_1X_2...X_n$ may begin can be computed automatically from the grammar. Thus we only have to check whether $b \in first(X_1X_2...X_n)$.

However, there may be another alternative $B \rightarrow Y_1 Y_2 ... Y_m$ for B with $b \in first(Y_1 Y_2 ... Y_m)$. In that case the next input symbol b does not suffice when deciding which B to use in the expansion. A grammar is called an **LL(1) grammar**, if this problem does not arise, i.e., if for each nonterminal B the $first$ sets of the right hand sides are pairwise disjoint.

Let us take a look at an example: We shall see that the grammar G_{arith} of arithmetic expressions is not LL(1). Therefore, we first consider the simpler *balanced strings* example. Recall that a balanced string is what remains if you strip from an arithmetic expression everything except for the parentheses. Obviously, `()(())` and `((()())())` are balanced strings, while `)(()())` and `((())` are not!

For balanced strings we use the grammar $G_{\text{balanced2}}$:

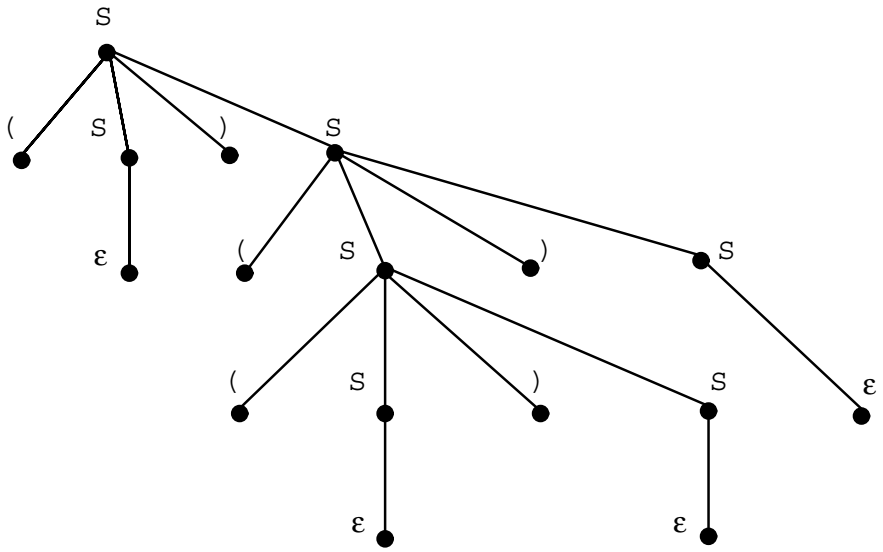
$$S \rightarrow (\; S \;) \; S$$
$$S \rightarrow \epsilon$$

Obviously, every string derived from the right hand side of the first rule starts with an opening parenthesis symbol. So if the next input symbol is an opening parenthesis we use the first rule for expansion. Otherwise — the next input symbol is a closing parenthesis or the **end-of-input symbol** # — we use the second rule.

This results in the following LL(1) parsing sequence for `()(())`:

```
( S       , ()(())# ,  start )
( (S)S    , ()(())# ,  expand "S -> (S)S" )
( S)S     , )(())#  ,  shift "(" )
( )S      , )(())#  ,  expand "S -> " )
( S       , (())#   ,  shift ")" )
( (S)S    , (())#   ,  expand "S -> (S)S" )
( S)S     , ())#    ,  shift "(" )
( (S)S)S  , ())#    ,  expand "S -> (S)S" )
( S)S)S   , ))#     ,  shift "(" )
( )S)S    , ))#     ,  expand "S -> " )
( S)S     , )#      ,  shift ")" )
( )S      , )#      ,  expand "S -> " )
( S       , #       ,  shift ")" )
(         , #       ,  expand "S -> " )
```

By applying all the expansions (in the above order) we obtain the syntax tree:



Here, the sequence of terminal symbols at leaf nodes (from left to right) is `()(())` (Notice that each ϵ is stripped off — it denotes the empty string, i.e., "nothing"!).

Keen observers may already have noted that the second rule $S \rightarrow \epsilon$ always produces the empty string — which neither starts with a closing parenthesis nor with the end-of-input symbol #!

This problem is inherent to all production rules $A \rightarrow \beta$ that can (directly or indirectly) produce the empty string: In that case the next symbol is not produced by the rule itself but from the right hand context of the left hand side A of the rule. Fortunately, this set ($follow(A)$) like the $first$ sets introduced above can be computed automatically .

Therefore, the set actually used for distinguishing a rule $A \rightarrow \beta$ from other rules with the same left hand side is $first(\beta\ follow(A))$. Because of their use in LL(1) parsing such sets are called **LL(1) decision sets**.

The LL(1) decision sets for G_{arith} are (in the tool-generated output below, set elements are separated by blanks only):

```
E -> E addOp T    { number oP variable }
E -> T            { number oP variable }

T -> T multOp F   { number oP variable }
T -> F            { number oP variable }

F -> number       { number }
F -> oP E cP      { oP }
F -> variable     { variable }
```

The different alternatives for F have disjoint LL(1) decision sets. Unfortunately, this does not hold for neither the alternatives of E nor for the alternatives of T. Hence, the grammar is not LL(1).

One possible workaround would be to transform G_{arith} into an LL(1) grammar which describes the same language. This succeeds: For grammar G_{arithll} the LL(1) decision sets are shown after each rule:

```
E -> T A      { id ( }
F -> id       { id }
F -> ( E )    { ( }
T -> F M      { id ( }
A ->          { # ) }
A -> + T A    { + }
M -> * F M    { * }
M ->          { # + ) }
```

But this solution has a drawback, too: Both the grammar and the syntax trees based on it are not very intuitive. Thus the grammar is rather unsuited as a basis for translation.

A better alternative is to use the original grammar in combination with a more powerful parsing scheme. We are now going to discuss such algorithms: bottom up strategies from the LR family of parsing algorithms.

As pointed out before, LL (top down) and LR (bottom up) parsing strategies differ in which kinds of steps they carry out explicitly and which kinds they perform "silently". To show how both strategies are related to each other, in the following transition sequence for input `(()())` we show *all* steps explicitly. Also, in the stack not only the current (innermost) items are shown, but also the surrounding (upper) levels:

```
( [S'->.S]                                  , (()())# ,  start )
( [S'->[S->.(S)S]]                          , (()())# ,  expand "S->(S)S" )
( [S'->[S->(.S)S]]                          , )()())#  ,  shift "(" )
( [S'->[S->([S->.])S]]                      , )()())#  ,  expand "S-> " )
( [S'->[S->(S.)S]]                          , )()())#  ,  reduce "S->  )
( [S'->[S->(S).S]]                          , ()())#   ,  shift ")" )
( [S'->[S->(S)[S->.(S)S]]]                  , ()())#   ,  expand "S->(S)S" )
( [S'->[S->(S)[S->(.S)S]]]                  , ())#     ,  shift "(" )
( [S'->[S->(S)[S->([S->.(S)S])S]]]          , ())#     ,  expand "S->(S)S" )
( [S'->[S->(S)[S->([S->(.S)S])S]]]          , ))#      ,  shift "(" )
( [S'->[S->(S)[S->([S->([S->.])S])S]]]      , ))#      ,  expand "S-> " )
( [S'->[S->(S)[S->([S->(S.)S])S]]]          , ))#      ,  reduce "S-> " )
( [S'->[S->(S)[S->([S->(S).S])S]]]          , )#       ,  shift ")" )
( [S'->[S->(S)[S->([S->(S)[S->.]])S]]]      , )#       ,  expand "S-> " )
( [S'->[S->(S)[S->([S->(S)S.])S]]]          , )#       ,  reduce "S-> " )
( [S'->[S->(S)[S->(S.)S]]]                  , )#       ,  reduce "S->(S)S" )
( [S'->[S->(S)[S->(S).S]]]                  , #        ,  shift ")" )
( [S'->[S->(S)[S->(S)[S->.]]]]              , #        ,  expand "S-> " )
( [S'->[S->(S)[S->(S)S.]]]                  , #        ,  reduce "S-> " )
( [S'->[S->(S)S.]]                          , #        ,  reduce "S->(S)S" )
( [S'->S.]                                  , #        ,  reduce "S->(S)S" )
(                                           , #        ,  accept )
```

Please check for yourself:

- In every line the current (innermost) item and the next input symbol uniquely determine the content of following line.
- From the above sequence the **LL(1) parsing sequence** is obtained by deleting
  - all reduction lines and
  - from the other lines: the marker point, all stack symbols to the left of it, and all remaining brackets (this leaves in the stack the sequence of symbols symbols that still have to be expanded).

- Analogously, the **LR parsing sequence** is obtained by deleting
  - all expansion lines and
  - from the other lines: the marker point, all stack symbols to the right of it, all left hand sides of rules (including the arrows), and all remaining brackets (this leaves in the stack the sequence of symbols whose subtrees have been traversed completely).

That way we obtain the following LR parsing sequence for input `(()())`:

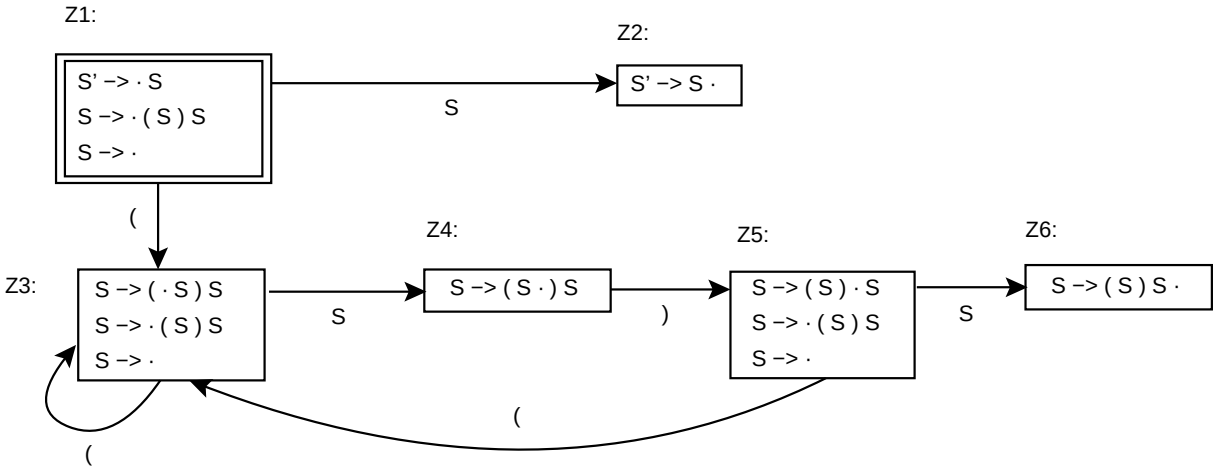
```
(            , (()())# ,  start )
( (          , )()())#  ,  shift "(" )
( (S         , )()())#  ,  reduce "S-> " )
( (S)        , ()())#   ,  shift ")" )
( (S)(       , ())#     ,  shift "(" )
( (S)((      , ))#      ,  shift "(" )
( (S)((S     , ))#      ,  reduce "S-> " )
( (S)((S)    , )#       ,  shift ")" )
( (S)((S)S   , )#       ,  reduce "S-> " )
( (S)(S      , )#       ,  reduce "S->(S)S" )
( (S)(S)     , #        ,  shift ")" )
( (S)(S)S    , #        ,  reduce "S-> " )
( (S)S       , #        ,  reduce "S->(S)S" )
( S          , #        ,  reduce "S->(S)S" )
( S'         , #        ,  accept )
```

When comparing the LL(1) and the LR parsing sequences the following differences are noticeable:

- For LL(1) parsing in each configuration the stack contents, the next input symbol, and the LL(1) decision sets suffice to uniquely determine the following configuration.
- For LR parsing the stack contents and the next input symbol do not suffice to determine the next configuration. We need control information (comparable to LL(1) decision sets) to determine, e.g., the current item. This kind of control information for LR parsing is called an LR automaton.

LR automata are finite automata whose states states contain sets of items. They have transitions for both terminal and nonterminal symbols. An LR configuration's *current* LR state is obtained as follows: start with the start state of the LR automaton and then for each stacked symbol in order perform the corresponding transition in the LR automaton. The LR state thus reached is the current LR state whose (shift and reduce) items define the LR parser's next transition (for efficiency reasons, practical parsers stack LR states instead of grammar symbols as demonstrated below).

The **LR(0) automaton** for our example grammar is:



Construction of the LR(0) automaton begins with the start item `S' -> .S`. After that, two items are added that are obtained from the start item by expansion. Since no more expansions can be applied, these three items are the contents of the (doubly framed) **start state** `Z1` of the LR(0) automaton.

All other states `Znew` are computed from already existing states `Zold` in the following way: Whenever `Zold` contains an item `A -> alpha . X beta`, the **X-successor** `Znew` of `Zold` contains `A -> alpha X . beta` (i.e., the marker point is pushed across symbol `X`). Like for the start state all available expansion steps are performed to complete state `Znew`.

That was it! (By the way: this again can be explained as an application of "Myhill's subset construction" to a nondeterministic automaton with ϵ transitions.)

The LR(0) parsing sequence for `(()())` is:

```
( Z1                          , (()())# ,  start )
( Z1:Z3                       , )()())#  ,  shift "(" )
( Z1:Z3:Z4                    , )()())#  ,  reduce "S-> " )
( Z1:Z3:Z4:Z5                 , ()())#   ,  shift ")" )
( Z1:Z3:Z4:Z5:Z3              , ())#     ,  shift "(" )
( Z1:Z3:Z4:Z5:Z3:Z3           , ))#      ,  shift "(" )
( Z1:Z3:Z4:Z5:Z3:Z3:Z4        , ))#      ,  reduce "S-> " )
( Z1:Z3:Z4:Z5:Z3:Z3:Z4:Z5     , )#       ,  shift ")" )
( Z1:Z3:Z4:Z5:Z3:Z3:Z4:Z5:Z6  , )#       ,  reduce "S-> " )
( Z1:Z3:Z4:Z5:Z3:Z4           , )#       ,  reduce "S->(S)S" )
( Z1:Z3:Z4:Z5:Z3:Z4:Z5        , #        ,  shift ")" )
( Z1:Z3:Z4:Z5:Z3:Z4:Z5:Z6     , #        ,  reduce "S-> " )
( Z1:Z3:Z4:Z5:Z6              , #        ,  reduce "S->(S)S" )
( Z1:Z2                       , #        ,  reduce "S->(S)S" )
( S'                          , #        ,  accept )
```

Notice that each (shift or reduce) action is determined by a corresponding item from the current LR state of the preceding configuration. The current state is topmost (on the right hand side) of the stack. E.g., all `reduce "S->(S)S"` actions are determined by the only item of state `Z6`. It is instructive to understand this sequence in detail by looking at the current states of the LR(0) automaton and by comparing it to the previous LR parsing sequence.

Unfortunately, there are three LR states that each contain contradictory items: `Z1`, `Z3`, and `Z5` all contain the shift item `S -> .(S)S` and the reduce item `S -> .`, which call for different actions. This situation is called a **shift reduce conflict**. Other kinds of conflicts (which do not occur in this example) are **reduce reduce conflicts**, where two items call for different reductions. Notice that two shift actions can never contradict each other: There are *no* shift shift conflicts! A grammar is called an **LR(0) grammar** if its LR(0) automaton does not contain any conflict.

LR(0) parsing does not work deterministically for grammar $G_{\text{balanced2}}$ (like for most practical grammars), because reductions do not depend on the next input symbol (hence, the 0 in LR(0)).

In the above situation (and in similar cases) the conflict can be resolved by a simple observation: Reduction according to $[B \rightarrow \gamma\ .]$ should be performed only if the next input symbol belongs to the set $follow(B)$ of symbols that can follow B in some context.

For grammar $G_{\text{balanced2}}$ we have `follow(S) = { ) , # }`. Thus there is no problem to distinguish a reduction to `S` from a shift with symbol `(`. Since all LR(0) conflicts for $G_{\text{balanced2}}$ can be resolved by that simple method, the grammar is called an **SLR(1) grammar** (SLR(1) stands for "Simple LR with 1 symbol lookahead"). Applying the SLR(1) method the above LR(0) parsing sequence is obtained deterministically.

Grammar G_{arith} has the same characteristics: it is SLR(1), but not LR(0).

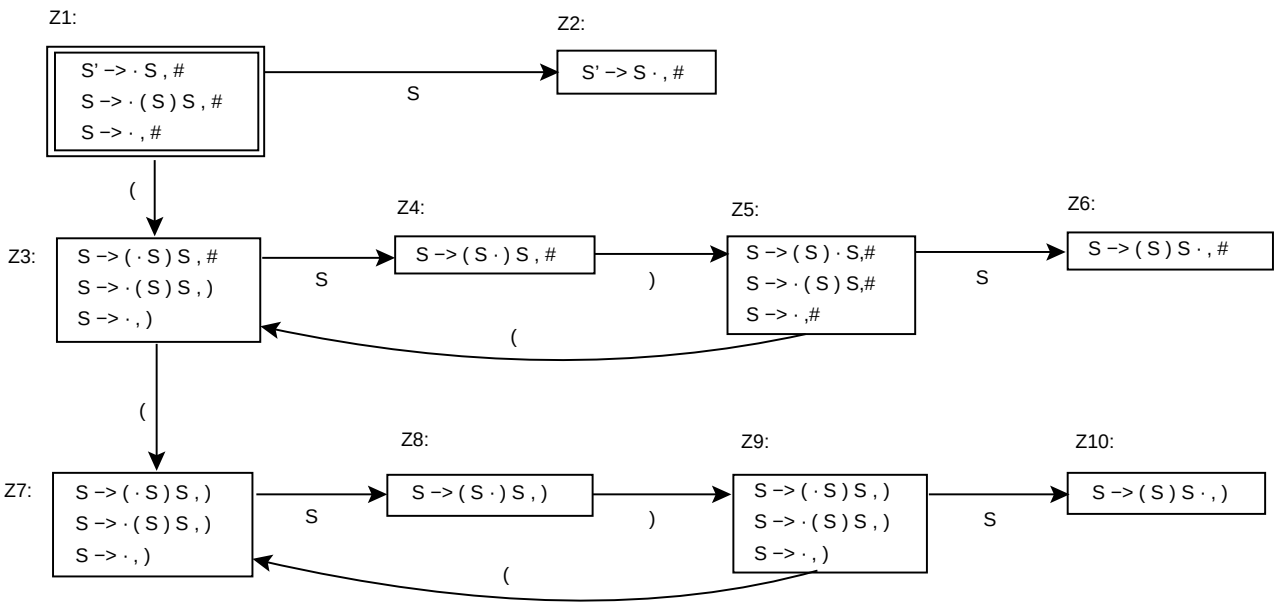
Typically, for large programming language grammars, the SLR(1) method does not suffice: A reduction using $[B \rightarrow \gamma\ .]$ is applicable whenever the next input symbols belongs to the set $follow(B)$ of all symbols that may follow in B *some* context. If some symbols from $follow(B)$ are not admissible in the *current* context of B, this information may still be too coarse. Therefore, more precise methods for right hand context ("lookahead") computation are needed.

The **canonical LR(1) automaton** contains for each item the "best possible one symbol lookahead" which is shown at the end of the item and is computed as follows:

- At the end of the parsing process the input symbol sequence will be exhausted leaving only the (virtual) end-of-input symbol — which is added to the input in order to signal the end of input. Therefore, the only admissible right hand context for S is $\#$. Adding this lookahead to the LR(0) start item one obtains the **LR(1) start item** $[S' \rightarrow\ .\ S\ ,\ \#]$.
- Like in the LR(0) construction the items of the X-**successor** of a state are computed by "pushing the marker point" across symbol X. Lookahead symbols are passed on unaltered. E.g., the S-successor of the start state contains the item $[S' \rightarrow\ S\ .\ ,\ \#]$ computed from the LR(1) start item.
- As in the LR(0) construction each state is completed by adding all available **expansion items**: For each item $[A \rightarrow\ \lambda\ .\ B\ \rho\ ,\ r]$ all expansion items $[B \rightarrow\ .\ \gamma\ ,\ s]$ with $s \in first(\rho\ r)$ are added (because B may be followed by exactly those symbols that strings derived from $\rho\ r$ may start with).

Canonical LR(1) automata differ from LR(0) automata only in the lookahead information added to each item. The start item's lookahead is $\#$. Other lookahead symbols are computed in expansion steps. Lookaheads are then passed on unaltered to be used in reduction steps: The **LR(1) reduction** defined by item $[B \rightarrow\ \gamma\ .\ ,\ c]$ is admissible only if the next next input symbol is equal to the lookahead symbol c.

For our example grammar we obtain the following **canonical LR(1) automaton**:



Here, the cost of increased precision is made out immediately: States Z7 through Z10 differ from states Z3 through Z6 in the lookahead information only. Because of this "combinatorial explosion effect" (which becomes worse for larger grammars) canonical LR(1) automata are not used in practice.

A good compromise combining sufficient precision with high storage efficiency is obtained by the **LALR(1)** method: Whenever two states differ only in their lookahead information (like Z3 and Z7 above), they are merged by taking the union of their sets of items. E.g., for Z3 and Z7 above the merged state contains the item set

```
{ [S -> S.(S),#], [S -> S.(S),)], [S -> .S(S),)], [S -> .,)] }
```

LALR(1) parsing is the method of choice in today's parser generators.

5. Synthesis Using Attributes

The syntax tree built by syntax analysis from the token sequence produced by lexical analysis makes visible the hierarchical structure of language artifacts. This structure and the information atoms provided by lexical tokens form the basis of the second part of the translation process, i.e., **synthesis**.

The **"attributes method"** systematically adds such information to the given syntax tree as will support the required translation task. An **attribute** is a piece of information which is assigned to a node of the syntax tree. Corresponding to the wide range of translation purposes there is a wide variety of attribute types used to support the translation process. In order to be able to process all kinds of syntax trees in a systematic and type-safe way, the following requirements have to be met:

- Every attribute has a **name** and a fixed **data type**.
- Tree nodes labelled with the same (nonterminal) grammar symbol are assigned **the same set of attributes**. E.g., we might assign to each $Term$ node an attribute `sign` of type `bool` and an attribute `amount` of type `double`.
- Different grammar symbols may have **overlapping attribute sets** — they usually do, since during attribute evaluation (see below) attribute values are often passed along a series of different tree nodes.

The process of computing the **attribute values** of a syntax tree is called **attribute evaluation**. Attribute evaluation is defined by a set of **attribute evaluation rules** that describe how attribute values are computed from other attribute values (also using the token strings). In special cases an attribute may have a constant value (i.e., a value which is identical for corresponding tree nodes in similar contexts). For attribute evaluation we require that:

- An attribute evaluation rule is always evaluated in the **context** of one of the grammar rules: Thus the attribute being computed and the attributes used in that computation are assigned to the same tree node or to neighbouring tree nodes. This is called the **locality principle of attribute evaluation**.
- Attribute evaluation rules are defined relative to grammar rules — one attribute evaluation rule for every attribute computed in that context. Since a grammar has a finite number of production rules each containing a finite number of grammar symbols which in turn are assigned a finite number of attributes, there is only a **finite number of attribute evaluation rules**. These suffice to compute all attributes in each of the **infinitely many syntax trees** that can be built from the grammar's production rules.

- Except for the root and the leaves every node of the syntax tree belongs to exactly two contexts (on the right hand side of one production rule and on the left hand side of another). Therefore, information can be propagated by "handshakes" from one context to the next and thus stepwise through the whole syntax tree.
- We distinguish between **inherited** and **synthesized** attributes: If an attribute evaluation rule computes an attribute of the *left hand side* of a production rule, we have a synthesized attribute, otherwise (attribute of a *right hand side* symbol) an inherited attribute. An attribute is either synthesized in all (production rule) contexts or inherited in all contexts.
- A production rule has a **complete** set of attribute evaluation rules, if for each synthesized attribute of the symbol on left hand side and for each inherited attribute of a symbol on the right hand side there is exactly one attribute evaluation rule.
- Initially, all the attribute values of a syntax tree are undefined. By applying attribute evaluation rules they are evaluated in an order which is compatible with the evaluation dependencies between the attributes. The (synthesized) attributes of the root node are considered to be the **result** of attribute evaluation.
- The attribute values and the dependencies between them form a (directed) **attribute graph** which is superimposed on the syntax tree. Complete attribute evaluation is possible only if no attribute value directly or indirectly depends on itself. In other words: The attribute graph must be **cycle free**.
- A context-free grammar with an assigned set of attributes and a corresponding set of attribute evaluation rules together form an **attribute grammar**. An attribute grammar is called
  - **complete**, if each production rule has a complete set of attribute evaluation rules;
  - **cycle free**, if *for each* syntax tree the corresponding attribute graph is cycle free;
  - **well-defined**, if it is complete and cycle free.

These are the basic notions for attribute grammars.

We now take a look at two different attribute grammars based on grammar G_{arith}. At the end of this section, we shall discuss two basic attribute evaluation algorithms and their properties.

The *first attribute grammar* translates a given arithmetic expression into an equivalent sequence of machine instructions which computes the value of the expression.

The *second attribute grammar* derives from a given arithmetic expression a so-called *Kantorovic tree*, i.e., a graphical representation which reduces a syntax tree to its very essence (in the form of a "text graphic").

These attribute grammars derive from the arithmetic expression

```
a*a - 2*a*b + b*b
```

the machine instruction sequence

```
b:
        DD 0
a:
        DD 0
        MOVE W a , -!SP
        MOVE W a , -!SP
        MULT W !SP+, !SP+, -!SP
        MOVE W I 2 , -!SP
        MOVE W a , -!SP
        MULT W !SP+, !SP+, -!SP
        MOVE W b , -!SP
        MULT W !SP+, !SP+, -!SP
        SUB  W !SP+, !SP+, -!SP
        MOVE W b , -!SP
        MOVE W b , -!SP
        MULT W !SP+, !SP+, -!SP
        ADD  W !SP+, !SP+, -!SP
```

(The first four lines reserve storage space for the two variables, a and b. A MOVE instruction pushes the value of some variable or some "immediate" number constant like 2 onto the stack. The arithmetic instructions (ADD, SUB, MULT) each take two operands from the stack and replace them on the stack by the operation's result.)

and

the Kantorovic tree

```
+
|\
| *
| |\
| | b
| |
| b
|
-
|\
| *
| |\
| | b
| |
| *
| |\
| | a
| |
| 2
|
*
|\
| a
|
a
```

(Notice that left hand side operands are further down than their corresponding right hand ones!)

A new start symbol, `S` , and a new start rule, `S->E` , are added for technical reasons: Now, inherited attributes can be assigned to the nonterminal `E` if required, and the start rule is an appropriate place for initializations.

We start developing an attribute grammar by considering which kind of information will be helpful at what places for the translation process to be implemented. We ask: Which attributes are needed? What are their types and kinds, i.e., are they synthesized or inherited? To which nonterminal symbols are they assigned? By default, our tools implement for each terminal symbol (token) a pseudo attribute `string` containing the token string, e.g., the digits of a `number` token which can be used to compute the number value of the token.

In the **translation to machine code** attribute grammar the complete instruction sequence is built from code pieces. We therefore use `code` attributes of type `String` and assign them to all nonterminal symbols of the grammar. Below, `String` constants are written between single apostrophes. The machine code starts with instructions that reserve storage for the variables occuring in the expression — one word of storage for each variable. For collecting the variables that occur in the expression we use an attribute `variables` of type `Set of String` (it represents a set of variable names). Both attributes are computed proceeding "from the leaves towards the root" of the syntax tree, i.e., both are synthesized.

Thus we obtain (≫ marks a synthesized attribute):

```
Nonterminals:
    E ≫code ≫variables
    T ≫code ≫variables
    F ≫code ≫variables
    S ≫code
```

For type `String` we have the following *operations*:

- `||` concatenates two `String`s.
- `newline` returns a `String` containing the "newline character".
- `blanks(<i>)` returns a `String` containing exactly `i` blanks.

For type `Set of String` we have the following *operations* (Smalltalk method names may end in a colon):

- `Set new` returns a new `Set of String`.
- `S1 union: S2` returns the union of sets `S1` and `S2`.
- `S add: name` adds the element `name` to set `S`.

In the attribute evaluation rules below, `codeE2` denotes the `code` attribute of the second instance of the nonterminal `E` in its production rule, `variablesF` denotes the `variables` attribute of the nonterminal's `F` only instance in its production rule, etc.

In the context of the first production rule for `E` we define the following two attribute evaluation rules:

The code of `E1` consists of the code for `E2`, followed by the code for `T` and either an `ADD` or a `SUB` instruction (depending on whether the addOp was a plus or minus). By taking the union of variables from `E2` and `T` one obtains those of `E1`.

Notice that evaluation rules are enclosed in brackets.

```
E -> E addOp T
   [if stringaddOp = '+'
    then codeE1 := (codeE2 || codeT || newline || blanks(8) || 'ADD  W !SP+, !SP+, -!SP')
    else codeE1 := (codeE2 || codeT || newline || blanks(8) || 'SUB  W !SP+, !SP+, -!SP')]
   [variablesE1 := variablesE2 union: variablesT]
```

The attribute evaluation rules for production rule `E -> T` below are trivial.

```
E -> T
   [codeE := codeT]
   [variablesE := variablesT]
```

The attribute evaluation rules for nonterminal `T` are analogous to those of `E` and, therefore, are not shown.

The attribute evaluation rules for `F` are:

```
F -> oP E cP
   [codeF := codeE]
   [variablesF := variablesE]!

F -> variable
   [codeF := blanks(8) || 'MOVE W ' || stringvariable || ' , -!SP']
   [variablesF := (Set new) add: stringvariable]

F -> number
   [codeF := '     MOVE W I ' || stringnumber || ' , -!SP']
   [variablesF := Set new]
```

In the context of the new start rule, for each of the variables collected in `variablesE` two lines of storage reservation code (labelled with the variable name) are generated. Finally, the storage reservation code is put in front of the instruction sequence that computes the value of the expression:

```
S -> E
   [String h := '';
    for v in variablesE do
        h := v || ':' || newline ||
             blanks(8) || 'DD 0' || newline ||
             h
    od;
    codeS := h || codeE]
```

(Since there is only one attribute assigned to `S`, only one attribute evaluation rule is required for this context.)

In the attribute grammar that **generates the Kantorovic tree**, the text representations of trees are composed of the text representations of their subtrees. In the desired representation of a tree the left hand subtree is shown in the bottom (!) lines while the right hand subtree immediately follows the line with the operator symbol. In front of text representing the right hand subtree there is always a vertical bar which belongs to the vertical line leading to the left hand subtree.

In order to properly represent the recursive nesting of subexpressions we use two attributes: `text` attributes of type `String` are used for building up text representations of (sub)trees and `prefix` attributes of type `String` containing the (properly indented) sequences of vertical bars leading to left hand subtrees further down. The `text` attributes are assigned to all nonterminal symbols, the `prefix` attributes are required only for symbols representing subtrees.

The `text` attributes like the attributes in the first attribute grammar are computed "from the leaves towards the root", i.e., are synthesized attributes. The `prefix` attributes are computed "from the root towards the leaves" and, therefore, are inherited attributes.

Thus we obtain (≪ marks an inherited attribute):

```
Nonterminals:
    E ≫text ≪prefix
    T ≫text ≪prefix
    F ≫text ≪prefix
    S ≫text
```

Interesting evaluation rules are found in the context of E's first production rule:

```
E -> E addOp T
   [textE1 :=
      prefixE1 || stringaddOp || newline ||
      prefixE1 || '|\' || newline ||
      textT || newline ||
      prefixE1 || '|' || newline ||
      textE2]
   [prefixE2 := prefixE1]
   [prefixT := prefixE1 || '| ']
```

In the evaluation rule for textE1, each line starts with the prefixE1 of vertical bars preceding E1. (The subexpressions, E2 and T, have their own prefixes that are computed by evaluation rules two and three.) In the first line of the text representation of E1 the prefix is followed by the text of the addOp operator token as available from the pseudo attribute stringaddOp. The second line shows (after the prefix) the fork joining the two subtrees below. (To understand why the following two lines are preceded by the same number of vertical bars, check the rule for prefixT!) In the third line the text representation of the right hand subtree is included — probably spanning a few lines in the result. The fourth line only contains vertical bars. Finally, in the fifth line the representation of the left hand subtree is attached — again spanning a few lines in the result.

In the context of E's second production rule "transfer rules" simply pass on informationen in both directions:

```
E -> T
   [textE := textT]
   [prefixT := prefixE]
```

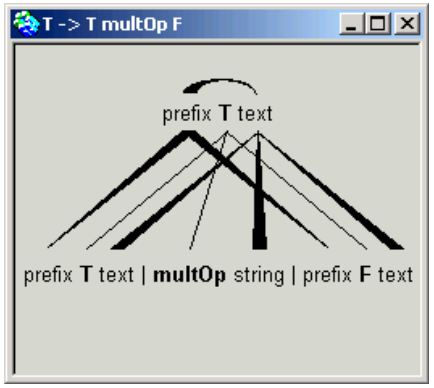
Again, the analogous rules for T are not shown. The rules for F below do not require any explanation (or do they?).

```
F -> oP E cP
   [textF := textE]
   [prefixE := prefixF]

F -> variable
   [textF := prefixF || stringvariable]

F -> number
   [textF := prefixF || stringnumber]
```

In the Kantorovic tree attribute grammar the interaction of inherited `prefixT` attributes and synthesized `text` attributes results in complex attribute dependencies.



The screenshot (generated by our tool SIC) visualizes the attribute dependencies occuring in the context of production rule `T -> T multOp F` : To the left of (bold) grammar symbols its inherited attributes are shown, and to the right its synthesized attributes. Attribute dependencies are represented by bold arcs. Thin edges connect grammar symbols on the left hand side (top) and the right hand side (bottom) of the production rule.

An attribute graph for a syntax tree is obtained by superimposing each application of a production rule in the syntax tree by its associated attributes and attribute dependencies. As mentioned before, every such attribute graph must be cycle free in order to guarantee that all attribute values can be computed.

The attributes of a cycle free attribute graph can be **"sorted topologically"**, i.e., arranged in a linear order where each attribute can only depend on attribute values to the left of it. Obviously, the leftmost attribute cannot depend on any other attribute value. Any such topological order also is a suitable order of evaluation.

Users of compiler compilers need not worry about suitable orders of evaluation: Either the attribute evaluator generator computes such an order in advance (thus producing a **"static evaluator"**) or the attribute evaluator computes a topological order during attribute evaluation (this is called a **"dynamic evaluator"**).

When deciding whether an attribute grammar is cycle free, or when precomputing the "visiting sequences" of a static evaluator, the first step is to determine for any two attributes assigned to the same grammar symbol, whether one may depend on the other in any syntax tree by any chain of attribute dependencies. For computing this information, rather complex approximation algorithms are used. Details can be found in the literature.

By another well-known theoretical result we can do completely without inherited attributes and still do the same transformations. The proof essentially says that instead of pushing information downwards into the tree using inherited attributes you can instead use synthesized attributes to transport information upwards. Since inherited attributes only serve to carry information towards the leaves until it is combined with information in synthesized attributes, one can instead carry the synthesized information up the tree until it meets the same inherited information somewhere closer to the root. In this transformation, however, the "local information processing" characteristic of attribute grammars is lost! **Inherited attributes allow attribute evaluation to be defined in a more lucid style.**

Dynamic evaluators dynamically check for cycles. Basically, there are two kinds of dynamic evaluators:

Data driven evaluator

The data driven evaluator first computes auxiliary information:

- For each attribute `a` its *number of immediate predecessors* (attributes on which `a` depends directly).
- For each attribute `a` its *list of immediate successors* (attributes depending directly on `a`).
- A list `F` of attributes that have no predecessors.

Also, the *topological order* to be computed is initialized with an empty list.

Now, while list `F` is not empty, one of its elements, `x`, is removed and processed as follows: `x` is appended to the end of the topological order. For each successor `y` of `x` we decrease `y`'s number of predecessors. In case `y`'s number of predecessors becomes 0, `y` is added to `F`.

In case `F` is empty, but the topological order does not contain all the attributes in the syntax tree, then there is a cycle in the attribute graph (formed by some of the remaining attributes).

Demand driven evaluator

Initially, all attributes are marked "undefined".

The demand driven evaluator is a recursive procedure which (in a loop) is called for every (synthesized) attribute. At a call on an "undefined" attribute `x`, `x` is marked "in process". Then, all information required for computing `x` is obtained by recursive calls of the demand driven evaluator on all immediate predecessors of `x`. Finally, the value of `x` is computed and its status becomes "completed".

If the demand driven evaluator is called for a "completed" attribute, it just returns its (available) value.

A cycle in the attribute graph is recognized, if an attribute marked "in process" is called (thus directly or indirectly depending on its own value).

If storage efficiency is an issue, such attribute evaluators are attractive that can be **embedded into syntax analysis**, i.e., where attributes are evaluated in an order compatible with parsing. That way, all translation processing can be done in one pass without ever storing the complete syntax tree and all the attributes. The parsing strategy best suited to this purpose is LL(1) parsing because it recognizes syntactic structures before entering them. For an attribute grammar with synthesized attributes only, attribute evaluation can be performed during LL(1) parsing.

6. Sources

Regular Expressions have been made popular by the Unix "Pipe&Filter" programs: small utility programs that read and produce character streams. Because of their common data format they can be flexibly combined into ever new combinations. Many useful programs belong to this tool set. Among them are

- *filter programs* (like "grep", "fgrep", and "egrep") that find instances of regular patterns in texts;
- the *stream editor* "sed" for simple, batch oriented text processing;
- the popular *text editor* "emacs" offers operations that are based on regular pattern matching and can be adapted by users to their particular needs;
- powerful *string processing languages* like "awk" and "perl" whose capabilities are based regular expressions;
- the *scanner generator* "lex" whose tokens are defined by regular expressions, too.

A complete monograph is dedicated to *practical uses of regular expressions* — all the fine details of slightly different versions of regular expressions, how they are to be interpreted and how they are implemented by the different tools; also, many examples how to make good use of them:

J.E.F. Friedl: Mastering Regular Expressions, O'Reilly 1997.

The above tools are described in great detail in this and other O'Reilly books.

Many of the compiling techniques described in this article were created or at least heavily influenced by **D. Knuth** (who also has given us the text formatting tool *TEX*): He started the development of both *LR parsing* algorithms and the *attribute grammars* approach to translation. More details can be found in **classical compiler text books** including the well-known "Dragon Book":

A.V. Aho, R. Sethi, J.D. Ullman: Compilers - Principles, Techniques, and Tools, Addison-Wesley 1986.

A comprehensive textbook on the design and construction of compilers, covering in particular techniques and structures for the efficient compilation of OO, functional, and logic programming languages — aspects not covered by the Dragon Book — is:

R. Wilhelm, D. Maurer: Compiler Design, Addison-Wesley 1995.

Among the many more good text books on compilers Java experts might choose:

A.W. Appel: Modern Compiler Implementation in Java, Cambridge Univ. Press 1997.

The **theoretical basis** is covered comprehensively in:

S. Sippu, E. Soisalon-Soininen: Parsing Theory I/II, Springer 1988/1990.

H. Alblas, B. Melichar (Ed.): Attribute Grammars, Applications and Systems, Springer 1991.

Links to interesting compiler compilers and to a comprehensive link collection, which also includes related tools like Reps' and Teitelman's *Synthesizer Generator* (generates syntax based editors that perform incremental attribute evaluation):

lex & yacc *This "classic" tool comes with Unix.*

All kinds of expressions can be specified in a convenient way: For every operator symbol, only their associativity (left, right, or none) and their priority level have to be defined by users. This is a particularly safe and convenient approach to expression syntax.

Eli *A comprehensive "tool kit" for almost every need.*

This is an international project, which is supported by teams from Australia, Germany, and the USA. There is very good user support, with short reaction times and detailed answers to problem reports.

Catalog of Compiler Construction Tools *A very useful collection of links* that is maintained by Fraunhofer-Institut for computer architecture and software technology.

From the many sources on **XML** we explicitly mention Sun's parameterized JAXP Framework for creating XML parsers (both DOM and SAX). The accompanying tutorial explains its use in great detail on helpful examples. This (and a wealth of other relevant tools and information) is available via the following links:

Sun Parser Framework JAXP

Sun Parser Framework Tutorial

XML Software

Free XML tools and software

X-ING site (HU Berlin)