Grammars and parsing with Java[1]

Peter Sestoft, Department of Mathematics and Physics
Royal Veterinary and Agricultural University, Denmark
E-mail: `sestoft@dina.kvl.dk`

Version 0.08, 1999-01-18

[1]The first (ML) version of these notes were written in 1992 while at the Technical University of Denmark.

Contents

1 Grammars and Parsing

Often the input to a program is given as a text, but internally in the program it is better represented more abstractly: by a Java object, for instance. The program must read the input text, check that it is well-formed, and convert it to the internal form. This is particularly challenging when the input is in 'free format', with no restrictions on the lay-out.

For example, think of a program for doing symbolic differentiation of mathematical expressions. It needs to read an expression involving arithmetic operators, variables, parentheses, etc. It must check that the parentheses match, it should allow any number of blanks around operators, and so on, and must build a suitable internal representation of the expression. Doing this without a systematic approach is very hard.

Example 1 This text file describes the probable states of a slightly defective gas gauge in a car, given the state of the car's battery and its gas tank:

```
probability(GasGauge | BatteryPower, Gas)
{
        (0, 0): 100.0,  0.0;
        (0, 1): 100.0,  0.0;
        (1, 0): 100.0,  0.0;
        (1, 1):   0.1, 99.9;
}
```

These lecture notes explain how to create programs that can read an input text file such as the above, check that its format is correct, and build an internal representation (an array or a list) of the data in the input file. Here we shall not be concerned with the meaning[2] of these data. □

Thus we provide simple tools to perform these tasks:

- systematic *description* of the structure of input data, and
- systematic *construction* of programs for reading and checking the input, and for converting it to internal form.

The input descriptions are called *grammars*, and the programs for reading input are called *parsers*. We explain grammars and the construction of parsers in Java. The methods shown here are essentially independent of Java, and can be used with suitable modifications in any language that has recursive procedures (Ada, C, ML, Modula, Pascal, etc.)

The order of presentation is as follows. First we introduce grammars, then we explain parsing, formulate some requirements on grammars, and show how to construct a parser skeleton from a grammar which satisfies the requirements. These parsers usually read sequences of symbols instead of raw texts. So-called *scanners* are introduced to convert texts to symbol sequences. Then we show how to extend the parsers to build an internal representation of the input while reading and checking it.

Throughout we illustrate the techniques using a *very* simple language of arithmetic expressions. At the end of the notes, we apply the techniques to parse and evaluate more realistic arithmetic expressions, such as `3.1*(7.6-9.6/-3.2)+(2.0)`.

When reading these notes, keep in mind that although it may look 'theoretical' at places, the goal is to provide a *practically* useful tool.

[2]The lines (0, 0) and (0, 1) say that if the battery is completely uncharged (0) and the tank is empty (0) or non-empty (1), then the meter will indicate Empty with probability 100%. The line (1, 0) says that if the battery is charged (1) and the tank is empty (0), then the gas gauge will indicate Empty with probability 100% also. Finally, the line (1, 1) says that even when the battery is charged (1) and the tank is non-empty (1), the gas gauge will (erroneously) indicate Empty with probability 0.1% and Nonempty with probability 99.9%.

2 Grammars

2.1 Grammar notation

A *grammar* G is a set of rules for combining symbols to a well-formed text. The symbols that can appear in a text are called *terminal symbols.* The combinations of terminal symbols are described using *grammar rules* and *nonterminal symbols.* Nonterminal symbols cannot appear in the final texts; their only role is to help generating texts: strings of terminal symbols.

A *grammar rule* has the form `A = f`$_1$ `| ... | f`$_n$ where the `A` on the left hand side is the nonterminal symbol defined by the rule, and the `f`$_i$ on the right hand side show the legal ways of deriving a text from the nonterminal `A`.

Each *alternative* `f` is a *sequence* `e`$_1$ `...` `e`$_m$ of symbols. We write Λ for the empty sequence (that is, when $m = 0$).

A *symbol* is either a *nonterminal symbol* `A` defined by some grammar rule, or a *terminal symbol* `"c"` which stands for `c`.

The *starting symbol* `S` is one of the nonterminal symbols. The well-formed texts are precisely those derivable from the starting symbols.

The grammar notation is summarized in Figure 1.

A *grammar* $G = (T, N, R, \mathtt{S})$ has a set T of terminals, a set N of nonterminals, a set R of rules, and a starting symbol $\mathtt{S} \in N$.
A *rule* has form `A = f`$_1$ `| ... | f`$_n$, where $\mathtt{A} \in N$ is a nonterminal, each alternative `f`$_i$ is a sequence, and $n \geq 1$.
A *sequence* has form `e`$_1$ `...` `e`$_m$, where each `e`$_j$ is a symbol in $T \cup N$, and $m \geq 0$. When $m = 0$, the sequence is empty and is written Λ.

Figure 1: Grammar notation

Example 2 Simple arithmetic expressions of arbitrary length built from the subtraction operator '`-`' and the numerals `0` and `1` can be described by the following grammar:

```
E     = T "-" E | T .
T     = "0" | "1" .
```

The grammar has terminal symbols $T = \{$`"-"`, `"0"`, `"1"`$\}$, nonterminal symbols $N = \{$`E`, `T`$\}$, two rules in R with two alternatives each, and starting symbol `E`. By convention, the first nonterminal is the starting symbol. □

2.2 Derivation

The grammar rule `T = "0" | "1"` above says that we may derive either the string `"0"` or the string `"1"` from the nonterminal `T`, by replacing or substituting either `"0"` or `"1"` for `T`. These *derivations* are written `T` $\Longrightarrow$ `"0"` and `T` $\Longrightarrow$ `"1"`.

Similarly, from nonterminal `E` we can derive `T`, for instance. From `T` we could derive `"0"`, for example, which shows that from `E` we can derive `"0"`, written `E` $\Longrightarrow$ `"0"`.

Choosing the other alternative for `E` we might get the derivation

$$
\begin{aligned}
\mathtt{E} &\Longrightarrow \mathtt{T\ "\text{-}"\ E} \\
&\Longrightarrow \mathtt{"0"\ "\text{-}"\ E} \\
&\Longrightarrow \mathtt{"0"\ "\text{-}"\ T} \\
&\Longrightarrow \mathtt{"0"\ "\text{-}"\ "1"}
\end{aligned}
$$

In each step of a derivation we replace a nonterminal with one of the alternatives on the right hand side of its rule. A derivation can be shown as a tree; see Figure 2.

Every internal node in the tree in labelled by a nonterminal, such as `E`. The sequence of children of an internal node, such as `T`, `"-"`, `E`, represents an alternative from the corresponding grammar rule.

A leaf of the tree is labelled by a terminal symbol, such as `"-"`. Taking the leaves in sequence from left to right gives the string derived from the symbol at the root of the tree: `"0" "-" "1"`.

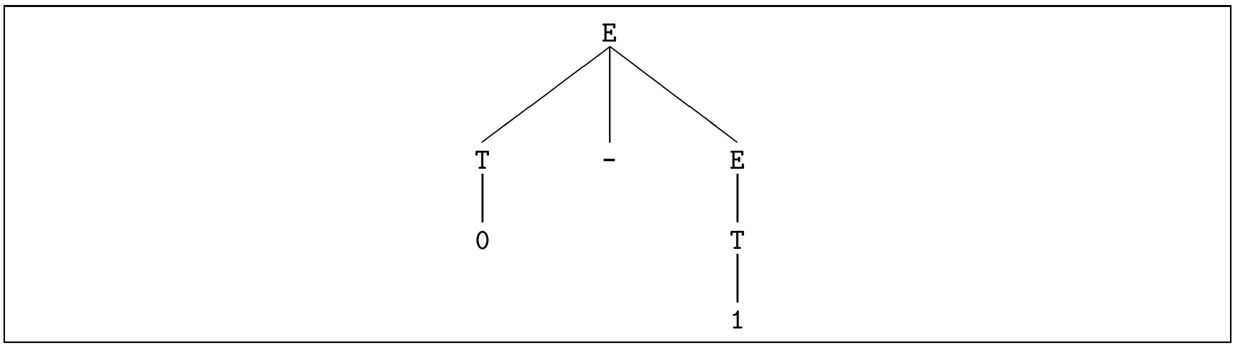


Figure 2: A derivation tree

One can think of a grammar G as a generator of strings of terminal symbols. Let T^* be the set of all strings of symbols from T, including the empty string Λ. When `A` is a nonterminal, the set of strings derivable from `A` is called $L(\mathtt{A})$:

$$L(\mathtt{A}) \;= \{\, w \in T^* \mid \mathtt{A} \Longrightarrow w \,\}$$

When grammar G has starting symbol S, the *language* generated by G is $L(G) = L(S)$. Grammars are useful because they are finite and compact descriptions of usually infinite languages.

In the example above we have $L(\mathtt{E}) = \{$`0, 1, 0-0, 0-1, 1-0, 1-1, 0-0-0,` $\ldots\}$, namely, the set of well-formed texts according to the grammar. As shown here, the quotes around strings of terminals are often left out.

Example 3 In mathematics, a rather liberal notation is used for writing down polynomials in x, such as $x^3 - 2x^2$. The following grammar describes such polynomials:

```
Poly      = Term
          | Plusminus Term
          | Poly Plusminus Term .
Term      = Natnum "x" Exponent
          | Natnum
          | "x" Exponent .
Exponent  = "^" Natnum
          | Λ .
Plusminus = "+" | "-" .
```

Assume that `Natnum` stands for any natural number `0`, `1`, `2`, ...

Check that the following strings are derivable: `"0"`, `"-0"`, `"2x + 5"`, `"x^3 - 2x^2"`, and that the following strings are not derivable: `"2xx"`, `"+-1"`, `"5 7"`, `"x^3 + - 2x^2"`. □

3 Parsing theory

We have seen that a grammar can be used to derive symbol strings having a certain structure. When a program reads an input file, the problem is the opposite: given an input text and a grammar, we want to see whether the text *could* have been generated by the grammar. Moreover, *when* this is the case, we want to know *how* it was generated, that is, which grammar rules and which alternatives were used in the derivation. This process is called *parsing* or *syntax analysis.*

For the class of grammars defined in Figure 1 it is always possible to reconstruct a correct derivation. In the method below we shall further restrict the grammars so that there is a simple and efficient way to perform the reconstruction.

This section explains a simple parsing principle. Section 4 explains how to construct Java parser programs working according to this principle.

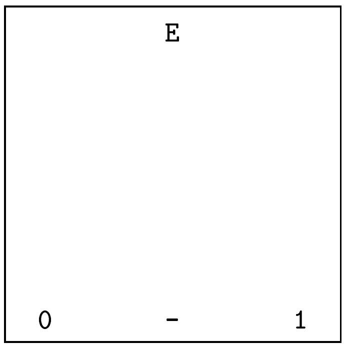
3.1 Parsing: reconstruction of a derivation tree

An attempt to reconstruct the derivation of a given string is called *parsing.* In these notes, we perform the reconstruction by working from the starting symbol down towards the given string. This method is called *top-down parsing.*

Consider again the grammar in Example 2:

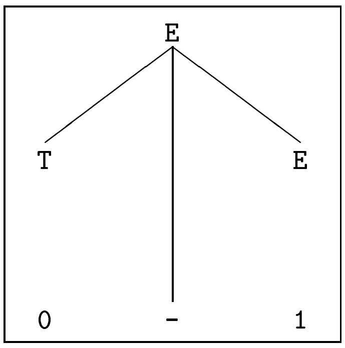
```
E     = T "-" E | T .
T     = "0" | "1" .
```

Let us reconstruct a derivation of the string `"0" "-" "1"` from the starting symbol `E`. We will do it by reconstructing the derivation tree, and therefore draw a box, write the starting symbol `E` at the top, and write the given input string `"0" "-" "1"` at the bottom of the box:



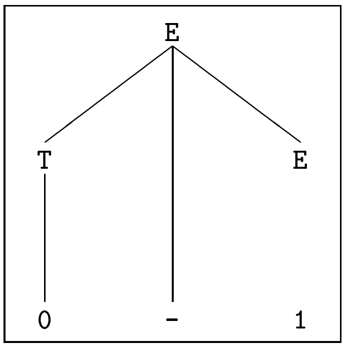
(a)

Our task is to find a derivation tree which connects `E` with the string at the bottom. We start from the top, and must derive something from `E`. According to the grammar, there are two possibilities, `E` $\Longrightarrow$ `T "-" E` and `E` $\Longrightarrow$ `T`. Only the first alternative is useful because the string, which involves a minus sign, could never be derived from `T`. So we extend the tree with the branches `T`, `"-"`, and `E`, as shown in box (b):



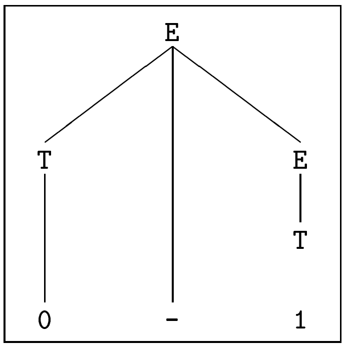
(b)

The next task is to derive the string `"0"` from `T`; luckily the grammar allows `T` $\Longrightarrow$ `"0"`, so we can extend the tree with the branch from `T` to `"0"` as shown in box (c):



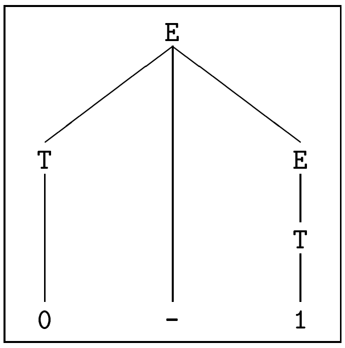
(c)

Next we must see how the remaining input symbol `"1"` can be derived from `E`. The `E` $\Longrightarrow$ `T` alternative is reasonable, so we extend the tree with a branch from `E` to `T`, as shown in box (d):



(d)

Finally, we must derive `"1"` from `T`, but again there is a rule `T` $\Longrightarrow$ `"1"`, so we extend the tree with a branch from `T` to `"1"`, as shown in box (e):



(e)

The parsing is complete: given the input string `"0" "-" "1"` we have constructed a derivation tree for it. When a derivation tree is the result of parsing, it is usually called a *parse tree*.

The derivation tree tells us two things. First, the input string *is* derivable from the starting symbol `E`. Secondly, we know at least one *way* it can be derived.

In the reconstruction, we worked from the top (`E`) and downwards; thus *top-down* parsing. Also note that in each step we extended the tree at the *leftmost* nonterminal.

3.2 A more machine-oriented view of parsing

We now consider another way to explain top-down parsing, more suited for programming than the trees shown above. We solve the same problem once more: can the string `"0" "-" "1"` be derived from `E` using the grammar in Example 2?

Previously we wrote down the string, wrote the nonterminal `E` above it, and reconstructed a derivation tree connecting the two. Now we write the string to the left, and the nonterminal `E` to the right:

```
"0" "-" "1"              E
```

This corresponds to the situation in box (a). In general there is a string of remaining input symbols on the left and a sequence of remaining grammar symbols (nonterminals or terminals) on the right. This situation can be read as an equation `"0" "-" "1"` = `E` between the two sides. Parsing solves the equation in a number of steps. Each step rewrites the leftmost nonterminal on the right hand side, until the input string has been derived. Whenever the same symbol is at the head of both sides, we can cancel it. This is much like cancellation in algebra, where $x + y = x + z$ can be reduced to $y = z$ by cancelling x. The parsing is successful when both sides are empty, that is, Λ.

Returning to our task, we must rewrite `E`. There are two possibilities, `E` $\Longrightarrow$ `T "-" E` and `E` $\Longrightarrow$ `T`. It is easy to see for the human reader that `"0" "-" "1"` can be derived only from the first alternative, because of the `"-"` symbol. We now rewrite `E` to `T "-" E` and have the configuration

```
"0" "-" "1"              T   "-" E
```

This corresponds to the situation in box (b). Since `T` $\Longrightarrow$ `"0"`, we can get

```
"0" "-" "1"              "0" "-" E
```

corresponding to the situation in box (c). Now we can cancel `"0"` and then `"-"` in both columns, so we need only see how the remaining input symbol `"1"` can be derived from `E`. Choosing the `E` $\Longrightarrow$ `T` alternative and then `T` $\Longrightarrow$ `"1"`, we get in turn:

```
            "1"                 E
            "1"                 T
            "1"                 "1"
```

The two latter lines correspond to the situations in box (d) and (e). Now we can cancel `"1"` on both sides, leaving the empty string Λ on both sides, so the parsing process was successful. The complete sequence of parsing steps was:

```
    "0" "-" "1"         E
    "0" "-" "1"         T   "-" E
    "0" "-" "1"         "0" "-" E
            "1"                 E
            "1"                 T
            "1"                 "1"
             Λ                   Λ
```

We want to mechanize the parsing process by writing a program to perform it, but there is one problem. To decide which alternative of `E` to use (in the first parsing step), we had to look ahead in the input string to find the symbol `"-"`. This lookahead is complicated to do in a program.

If our parsing program could choose the alternative by looking only at the *first* symbol of the remaining input, then the program would be simpler and more efficient.

3.3 Left factorization

The problem is with rules such as `E = T "-" E | T`, where both alternatives start with the same symbol, `T`. We would like to *factorize* the right hand side into '`T ("-" E | Λ)`', pulling the `T` outside a parenthesis, so to speak, and thus postponing the choice between the alternatives until after `T` has been parsed.

However, our grammar notation does not allow such parenthesized grammar fragments. To solve this problem we introduce a new nonterminal `Eopt` defined by `Eopt = "-" E | Λ`, and redefine `E` as `E = T Eopt`. Thus `Eopt` represents the parenthesized grammar fragment above.

Moreover, in Section 6 below it will prove useful to replace `E` in the `Eopt` rule with its only alternative `T Eopt`.

Example 4 Left factorization of the Example 2 grammar therefore gives

```
    E    = T Eopt .
    Eopt = "-" T Eopt | Λ .
    T    = "0" | "1" .
```

The set of strings derivable from `E` is the same as in Example 2, but the derivations will be different. □

Now the derivation of `"0" "-" "1"` (in fact, any derivation) must begin with `E` $\Longrightarrow$ `T Eopt`, and we need to see how `"0" "-" "1"` can be derived from `T Eopt`. Since `T` $\Longrightarrow$ `"0"`, we can cancel the `"0"` and only need to see how the remaining input `"-" "1"` can be derived from `Eopt`. There are two alternatives, `Eopt` $\Longrightarrow$ Λ and `Eopt` $\Longrightarrow$ `"-" T Eopt`.

Since Λ can derive only the empty string, whereas the other alternative can derive strings starting with `"-"`, we choose the latter. We now must see how `"-" "1"` can be derived from `"-" T Eopt`. The `"-"` is cancelled, and we must see how `"1"` can be derived from `T Eopt`. Now `T` $\Longrightarrow$ `"1"`, we cancel the `"1"`, and we are left with the empty input. Clearly the empty input can be derived from `Eopt` only by its first alternative, Λ.

The parsing steps for `"0" "-" "1"` with the left factorized grammar of Example 4 are:

```
"0" "-" "1"            E
"0" "-" "1"            T   Eopt
"0" "-" "1"            "0" Eopt
    "-" "1"                "-" T   Eopt
        "1"                    T   Eopt
        "1"                    "1" Eopt
         Λ                         Eopt
         Λ                          Λ
```

Notice that now we can always choose between the alternatives by looking only at the first symbol of the remaining input. The corresponding derivation tree is shown in Figure 3.

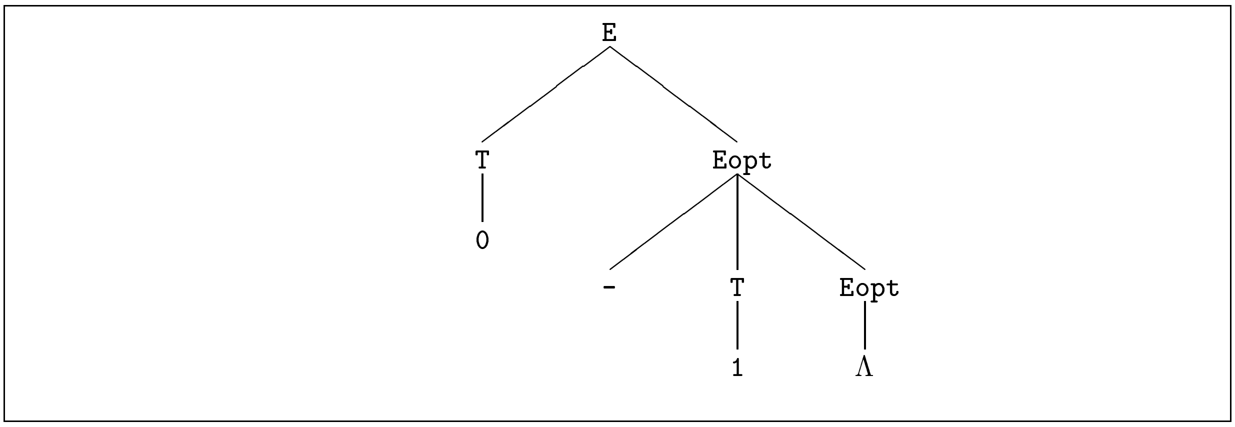


Figure 3: Derivation tree for the left factorized grammar

3.4 Left recursive nonterminals

There is another type of grammar rules we want to avoid. Consider the grammar

```
E    =  E "-" T | T .
T    =  "0" | "1" .
```

Some reflection (or experimentation) shows that it generates the same strings as the grammar from Example 2. However, `E` is *left recursive*: there is a derivation `E` $\Longrightarrow$ `E ...` from `E` to a symbol string that begins with `E`. It is even *self left recursive*: there is an alternative for `E` that begins with `E` itself. This means that the grammar is no good for top-down parsing, since we cannot choose between the alternatives for `E` by looking only at the first input symbol (in fact, not even by looking at any bounded number of symbols at the beginning of the string).

Left factorization is not possible for the above grammar, since the alternatives begin with different nonterminals. The only solution is to change the grammar to one that is not left recursive. Fortunately, this is always possible. In the present case, the original Example 2 grammar is a good solution.

In general, consider a grammar in which nonterminal `A` is self left recursive:

$$\mathtt{A} \quad = \mathtt{A}\ \mathtt{g}_1 \ |\ \ldots\ |\ \mathtt{A}\ \mathtt{g}_m\ |\ \mathtt{f}_1\ |\ \ldots\ |\ \mathtt{f}_n\ .$$

The $\mathtt{g}_i$ and $\mathtt{f}_j$ stand for sequences of grammar symbols (possibly Λ). We require that $m, n \geq 1$, and that no $\mathtt{f}_j$ can derive a string beginning with `A`, so the only left recursion is through the first m alternatives.

Observe that every string derived from `A` must begin with an $\mathtt{f}_j$, and continue with zero or more $\mathtt{g}_i$'s. Therefore we can construct the following equivalent grammar where `A` is not self left recursive:

```
A    = f₁ Aopt | ... | fₙ Aopt .
Aopt = g₁ Aopt | ... | gₘ Aopt | Λ .
```

The role of the new nonterminal `Aopt` is to derive sequences of zero or more $\mathtt{g}_i$'s.

The new grammar produced by this transformation usually is not left recursive, and it generates the same strings as the original one (namely, an $\mathtt{f}_j$ followed by zero or more $\mathtt{g}_i$'s). The transformation sometimes produces a new grammar which is again left recursive. In that case, one must apply (more) cleverness to find a non left recursive grammar.

3.5 First-sets, follow-sets and selection sets

Consider a rule $\mathtt{A} = \mathtt{f}_1 \mid \mathtt{f}_2$, and assume we have an input string '`t ...`' whose first input symbol is `t`. We want to decide whether this string could be derived from `A`. Moreover, we want to choose between the alternatives $\mathtt{f}_1$ and $\mathtt{f}_2$ by looking only at the first input symbol.

There are two ways it might make sense to choose $\mathtt{f}_1$. First, if we can derive a string *starting* with `t` from $\mathtt{f}_1$, then choosing $\mathtt{f}_1$ might be sensible. Secondly, if we can derive the empty string Λ from $\mathtt{f}_1$, and we can derive a string starting with `t` from something *following* whatever is derived from `A`, then choosing $\mathtt{f}_1$ might be sensible.

To make the choice between $\mathtt{f}_1$ and $\mathtt{f}_2$ simple, we shall *require* that for a given input symbol `t`, it makes sense to choose $\mathtt{f}_1$, or $\mathtt{f}_2$, or none of them, but it must never make sense to choose both. Accordingly, the parser chooses $\mathtt{f}_1$, or $\mathtt{f}_2$, or rejects the input as wrong. We now make this idea more precise.

The set of terminal symbols that can begin a string derivable from `f` is called its *first-set* and is written $\mathit{First}(\mathtt{f})$. The set of symbols that can follow a nonterminal `A` is called its *follow-set* and is written $\mathit{Follow}(\mathtt{A})$.

The *selection set* for an alternative $\mathtt{f}_i$ of a nonterminal $\mathtt{A} = \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n$ is $\mathit{First}(\mathtt{f}_i)$ if $\mathtt{f}_i$ cannot derive the empty string Λ, and $\mathit{First}(\mathtt{f}_i) \cup \mathit{Follow}(\mathtt{A})$ if $\mathtt{f}_i$ can derive Λ:

$$\mathit{Select}(\mathtt{f_i}) = \begin{cases} \mathit{First}(\mathtt{f}_i) \cup \mathit{Follow}(\mathtt{A}) & \text{if } \mathtt{f}_i \Longrightarrow \Lambda \\ \mathit{First}(\mathtt{f}_i) & \text{otherwise} \end{cases}$$

Intuitively, the selection set $\mathit{Select}(\mathtt{f}_i)$ is the set of input symbols for which it is sensible to choose $\mathtt{f}_i$. Why? It makes sense to choose $\mathtt{f}_i$ only if the first input symbol can be derived from $\mathtt{f}_i$, or if the input symbol can follow `A`, and `A` can derive Λ via $\mathtt{f}_i$.

How can we compute $\mathit{First}(\mathtt{f})$? If `f` is Λ, we have $\mathit{First}(\Lambda) = \{\}$ because the empty string does not start with any symbol.

If `f` is a terminal symbol `"c"`, we have $\mathit{First}(\texttt{"c"}) = \{\mathtt{c}\}$ because the only string derivable is `"c"`, which begins with `c`.

If `f` is a nonterminal `A` whose rule is $\mathtt{A} = \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n$, then the set of strings derivable is the union of those derivable from the alternatives $\mathtt{f}_i$. Therefore $\mathit{First}(\mathtt{A})$ is the union of the first-sets of the alternatives.

If `f` is a sequence $\mathtt{e}_1\ \mathtt{e}_2\ \ldots\ \mathtt{e}_m$, the set of strings derivable is the concatenation of strings derivable from the elements. Thus $\mathit{First}(\mathtt{f})$ includes $\mathit{First}(\mathtt{e}_1)$. Moreover, if $\mathtt{e}_1$ can derive Λ, then every string derivable from $\mathtt{e}_2\ \ldots\ \mathtt{e}_m$ is derivable also from $\mathtt{e}_1\ \mathtt{e}_2\ \ldots\ \mathtt{e}_m$. Therefore when $\mathtt{e}_1$ can derive Λ, $\mathit{First}(\mathtt{f})$ includes $\mathit{First}(\mathtt{e}_2\ \ldots\ \mathtt{e}_m)$ also.

The computation of $\mathit{First}(\mathtt{f})$ is summarized in Figure 4.

The first-set $First(\texttt{A})$ of a nonterminal `A` is the least (smallest) set of terminal symbols satisfying these equations:

$$\begin{array}{lcll}
First(\Lambda) & = & \{\} & \\
First(\texttt{"c"}) & = & \{\texttt{c}\} & \text{for terminal } \texttt{"c"} \\
First(\texttt{A}) & = & First(\texttt{f}_1) \cup \ldots \cup First(\texttt{f}_n) & \text{for nonterminal } \texttt{A} \\
 & & \text{where } \texttt{A} \text{ is defined by } \texttt{A = f}_1 \mid \ldots \mid \texttt{f}_n & \\
First(\texttt{e}_1\ \texttt{e}_2\ \ldots\ \texttt{e}_m) & = & \begin{cases} First(\texttt{e}_1) \cup First(\texttt{e}_2\ \ldots\ \texttt{e}_m) & \text{if } \texttt{e}_1 \Longrightarrow \Lambda \\ First(\texttt{e}_1) & \text{otherwise} \end{cases} &
\end{array}$$

Figure 4: Computation of first-sets

How can we compute the follow-set $Follow(\texttt{A})$ of a nonterminal `A`? Assume that `A` appears in the rule `B = ... | ... A f | ...` for nonterminal `B`, where `f` is a string of grammar symbols (possibly Λ). Then $Follow(\texttt{A})$ must include everything that `f` can begin with, that is, $First(\texttt{f})$. Moreover, if `f` can derive Λ, then $Follow(\texttt{A})$ must include also everything that can follow `B`. This is expressed by Figure 5.

The follow-set $Follow(\texttt{A})$ of nonterminal `A` is the least (smallest) set of terminal symbols satisfying for every rule `B = ...| ... A f | ...`, that

$$Follow(\texttt{A}) \supseteq \begin{cases} First(\texttt{f}) \cup Follow(\texttt{B}) & \text{if } \texttt{f} \Longrightarrow \Lambda \\ First(\texttt{f}) & \text{otherwise} \end{cases}$$

Figure 5: Computation of follow-sets

Note in particular that the situation `B = ...| ... A | ...` corresponds to that above, with `f` being Λ. In this case, the requirement is just $Follow(\texttt{A}) \supseteq Follow(\texttt{B})$ because $\texttt{f} \Longrightarrow \Lambda$ and $First(\Lambda) = \{\}$.

With these definitions, the requirement on grammars for parser construction is that the selection sets of distinct alternatives $\texttt{f}_i$ and $\texttt{f}_j$ are disjoint: $Select(\texttt{f}_i) \cap Select(\texttt{f}_j) = \{\}$. Then a given input symbol `c` can belong to the selection set of at most one alternative, so the input symbol determines which alternative to choose.

However, in practice we shall use the more easily checkable sufficient requirements given in Figure 6.

Every grammar rule must have one of the forms

Form 0:	$\texttt{A = f}_1$	
Form 1:	$\texttt{A = f}_1 \mid \ldots \mid \texttt{f}_n$	$n \geq 2$
Form 2:	$\texttt{A = f}_1 \mid \ldots \mid \texttt{f}_n \mid \Lambda$	$n \geq 1$

For rules of form 1 or 2 we require:

- For distinct $\texttt{f}_i$ and $\texttt{f}_j$ we must have $First(\texttt{f}_i) \cap First(\texttt{f}_j) = \{\}$.
- No $\texttt{f}_i$ can derive Λ.
- In rules of form 2, we must have $First(\texttt{f}_i) \cap Follow(\texttt{A}) = \{\}$ for all $\texttt{f}_i$.

Figure 6: Sufficient requirements on grammar for parsing

The requirements in Figure 6 imply that the grammar does not contain a left recursive nonterminal (unless the nonterminal is unreachable from the starting symbol, and therefore irrelevant).

Looking again at the left factorization example, we see that it does not satisfy the first requirement in Figure 6.

Example 5 Clearly $\mathit{First}(\texttt{"0"}) = \{\texttt{0}\}$ and $\mathit{First}(\texttt{"1"}) = \{\texttt{1}\}$, so in the grammar from Example 2

```
E     = T "-" E | T .
T     = "0" | "1" .
```

we have

$$\begin{array}{lcl} \mathit{First}(\texttt{T}) & = & \mathit{First}(\texttt{"0"}) \cup \mathit{First}(\texttt{"1"}) = \{\texttt{0, 1}\} \\ \mathit{First}(\texttt{T "-" E}) & = & \mathit{First}(\texttt{T}) = \{\texttt{0, 1}\} \end{array}$$

The rule `E = T "-" E | T` is of form 1 and does not satisfy the requirement on first-sets in Figure 6, since the first-sets of the alternatives are not disjoint; they are identical. This problem occurs whenever two alternatives begin with the same symbol. □

Example 6 Let us compute $\mathit{Follow}(\texttt{T})$ for the grammar shown above. Consulting Figure 5, we see that $\mathit{Follow}(\texttt{T})$ is the smallest set of terminal symbols which satisfies the two inequalities

$$\begin{array}{l} \mathit{Follow}(\texttt{T}) \supseteq \mathit{First}(\texttt{"-" E}) = \mathit{First}(\texttt{"-"}) = \{-\} \\ \mathit{Follow}(\texttt{T}) \supseteq \mathit{Follow}(\texttt{E}) \end{array}$$

The first inequality is caused by the alternative `E = T "-" E | ...`, and the second one by the alternative `E = ... | T`. In the latter case, the `f` following `T` is the empty string Λ.

But what is $\mathit{Follow}(\texttt{E})$? It is the empty set $\{\}$, since $\mathit{Follow}(\texttt{E})$ is defined to be the least set which satisfies

$$\mathit{Follow}(\texttt{E}) \supseteq \mathit{Follow}(\texttt{E})$$

because there is a rule `E = T "-" E | ...`. Any set satisfies this inequality. In particular the empty set does, and this is clearly the least such set.

Using this fact, we see that $\mathit{Follow}(\texttt{T}) = \{\texttt{-}\}$. □

Example 7 In the left factorized Example 4 grammar

```
E    = T Eopt .
Eopt = "-" T Eopt | Λ .
T    = "0" | "1" .
```

the `Eopt` rule has form 2, and we have for the alternatives of `Eopt`:

$$\begin{array}{lcl} \mathit{First}(\texttt{"-" T Eopt}) & = & \mathit{First}(\texttt{"-"}) = \{\texttt{-}\} \\ \mathit{First}(\Lambda) & = & \{\} \end{array}$$

Reasoning as for $\mathit{Follow}(\texttt{E})$ in the previous example, we also find that $\mathit{Follow}(\texttt{Eopt}) = \{\}$.

The first-sets $\{\}$ and $\{\texttt{-}\}$ of the two alternatives are disjoint, and $\mathit{First}(\texttt{"-" T Eopt}) \cap \mathit{Follow}(\texttt{Eopt}) = \{\}$, so the `E` rule satisfies the grammar requirements. The selection sets for the two alternatives are $\{\texttt{"-"}\}$ and $\{\}$. This shows how to choose between the alternatives of `E`: if the first input symbol is `"-"`, then choose the first alternative (`"-" T Eopt`), and if the input is empty, then choose the second alternative (Λ). □

Now we know about first-sets, consider again the left recursive rule `E = E "-" T | T` from Section 3.4. It has form 1, and for the first alternative we have

$$\begin{aligned} First(\texttt{E "-" T}) &= First(\texttt{E}) \\ &= First(\texttt{E "-" T}) \cup First(\texttt{T}) \\ &\supseteq First(\texttt{T}) \end{aligned}$$

Since $First(\texttt{T}) = \{\texttt{0, 1}\}$ is not empty, the first-sets of the alternatives `E "-" T` and `T` are not disjoint, and therefore the requirements of Figure 6 are not satisfied.

Example 8 The following grammar describes more realistic arithmetic expressions:

```
E  =  E "+" T | E "-" T | T .
T  =  T "*" F | T "/" F | F .
F  =  Real | "(" E ")" .
```

Here `E` stands for expression, `T` for term, and `F` for factor. So an expression is the sum or difference of an expression and a term, or just a term. A term is the product or quotient of a term and a factor, or just a factor. A factor is a constant number, or an expression surrounded by parentheses.

The rules for `E` and `T` must be transformed to remove left recursion as explained in Section 3.4:

```
E    = T Eopt .
Eopt = "+" T Eopt | "-" T Eopt | Λ .
T    = F Topt .
Topt = "*" F Topt | "/" F Topt | Λ .
F    = Real | "(" E ")" .
```

Now we must check the grammar requirements. First we compute the follow-sets.

To determine $Follow(\texttt{E})$ we list the requirements imposed by Figure 5, by considering all right hand side occurrences of `E`. There is only one, in the `F` rule:

$$\begin{aligned} Follow(\texttt{E}) &\supseteq First(\texttt{")"}) \\ &= \{\ \texttt{")"}\ \} \end{aligned}$$

Now $Follow(\texttt{E})$ is the smallest set satisfying this requirement, so

$$Follow(\texttt{E}) = \{\ \texttt{")"}\ \}$$

To determine $Follow(\texttt{Eopt})$ we similarly find the requirements

$$\begin{aligned} Follow(\texttt{Eopt}) &\supseteq Follow(\texttt{E}) \\ Follow(\texttt{Eopt}) &\supseteq Follow(\texttt{Eopt}) \end{aligned}$$

Again, $Follow(\texttt{Eopt})$ is the least set satisfying these requirements, so we conclude that

$$Follow(\texttt{Eopt}) = \{\ \texttt{")"}\ \}$$

To determine $Follow(\texttt{T})$ we note the sole requirement

$$\begin{aligned} Follow(\texttt{T}) &\supseteq First(\texttt{Eopt}) \cup Follow(\texttt{Eopt}) \\ &= \{\ \texttt{"+"}, \texttt{"-"}\ \} \cup Follow(\texttt{Eopt}) \end{aligned}$$

for which the smallest solution is

$$Follow(\texttt{T}) = \{\ \texttt{"+"}, \texttt{"-"}, \texttt{")"}\ \}$$

To determine $Follow(\texttt{Topt})$ we note the requirements

$$\begin{aligned} Follow(\texttt{Topt}) &\supseteq Follow(\texttt{T}) \\ Follow(\texttt{Topt}) &\supseteq Follow(\texttt{Topt}) \end{aligned}$$

for which the smallest solution is

$$\begin{aligned} Follow(\texttt{Topt}) &= Follow(\texttt{T}) \\ &= \{ \texttt{"+"}, \texttt{"-"}, \texttt{")"} \} \end{aligned}$$

Now let us check the grammar requirements from Figure 6:

- The rules for `E` and `T` are of type 0 and therefore OK.
- The rule for `F` is of type 1 and OK because the first-sets { `Real` } and { `"("` } are disjoint.
- The rule for `Eopt` is of type 2 and OK because the first-sets { `"+"` } and { `"-"` } and the follow-set $Follow(\texttt{Eopt}) = \{ \texttt{")"} \}$ are disjoint.
- The rule for `Topt` is of type 2 and OK because the first-sets { `"*"` } and { `"/"` } and the follow-set $Follow(\texttt{Topt}) = \{ \texttt{"+"}, \texttt{"-"}, \texttt{")"} \}$ are disjoint.

Thus the transformed grammar satisfies the requirements. □

3.6 Summary of parsing theory

We have shown informally how top-down parsing works. We defined the concepts of first-set and follow-set. Using these concepts, we formulated a sufficient requirement on grammars for parser construction. For a grammar to satisfy this requirement, it must have no two alternatives starting with the same symbol, and no left recursive rules.

4 Parser construction in Java

We now show a systematic way to write a *parser skeleton* (in Java) for input described by a grammar satisfying the requirements in Figure 6. The parser skeleton checks that the input is well-formed, but does not build an internal representation of it; this will be done in Section 6.

4.1 Java representation of the input

The raw input (from a text file) is a stream of characters. For parsing, we need to turn this into a stream of tokens, where a *token* is the internal representation of a terminal symbol.

In Java we can represent a token stream by an object `ts` of some class Tokenstream. This class must have a field `ts.tok` which holds the current token, and it must have a method `ts.next()` to read the next token in the stream. In addition, it must have a method `parseerror` to signal an error in the input. The parser must access the input through the token stream `ts` only.

We shall use integers (`int`) to represent tokens; this allows us to use tokens in Java's `switch` statements. A simple token corresponding to a character, such as `'-'`, may be represented just by its character code (always non-negative). A simple token corresponding to a keyword in the source language, such as `while`, may be represented by a negative integer, e.g. -101.

However, some tokens occur in families. For instance, the terminal symbol `Real` may stand for all floating-point numbers. We cannot have a distinct integer for each floating-point number, so terminal symbols belonging to this family will be represented by a number, such as -97, together with a field `nval` of type `double` in the Tokenstream object. We return to the subject of token streams in Section 5 below.

4.2 Constructing parsing methods in Java

A parser for a grammar G satisfying the requirements of Figure 6 can be constructed systematically from the grammar rules. The parser will consist of a set of mutually recursive *parsing methods*, one for each nonterminal in the grammar.

The parsing method corresponding to nonterminal `A` is called `A` also. It tries to find a string derivable from `A` at the beginning of the current token stream. If it succeeds, then it just returns, possibly after having read more tokens from the token stream. If it fails, then it throws an exception.

Grammar The parser for grammar $G = (T, N, R, \mathtt{S})$ has form

```
void A₁() { ... }
void A₂() { ... }
...
void Aₖ() { ... }

void parse() {
    S();
    if (ts.tok != Tokenstream.EOF)
      throw ts.parseerror("Expected end of file");
    return;
}
```

where $\{\mathtt{A}_1, \ldots, \mathtt{A}_k\} = N$ is the set of nonterminals and `S` is the starting symbol. The main method is `parse`; it tries to parse the input according to `S`. If parsing succeeds and no input remains, it just returns; otherwise it raises an exception.

Rule of form 0 The parsing method for a rule of form `A = f`$_1$ is

```
void A() {
   parse code for f$_1$
   return;
}
```

Rule of form 1 The parsing method for a rule of form `A = f`$_1$ `| ... | f`$_n$ is

```
void A() {
   switch (ts.tok) {
   case t$_{11}$: ... case t$_{1a_1}$:
       parse code for alternative f$_1$
       return;
   ...
   case t$_{n1}$: ... case t$_{na_n}$:
       parse code for alternative f$_n$
       return;
   default:
       throw ts.parseerror("Expected t$_{11}$ or ... or t$_{na_n}$");
   }
}
```

where $\{\texttt{t}_{i1},\ldots,\texttt{t}_{ia_i}\} = First(\texttt{f}_i)$ is the first-set of alternative $\texttt{f}_i$, for $i = 1,\ldots,n$.

Rule of form 2 The parsing method for a rule of form `A = f`$_1$ `| ... | f`$_n$`|` Λ is

```
void A() {
   switch (ts.tok) {
   case t$_{11}$: ... case t$_{1a_1}$:
       parse code for alternative f$_1$
       return;
   ...
   case t$_{n1}$: ... case t$_{na_n}$:
       parse code for alternative f$_n$
       return;
   default:
       return;
   }
}
```

where $\{\texttt{t}_{i1},\ldots,\texttt{t}_{ia_i}\} = First(\texttt{f}_i)$ as above.

Sequence The parse code for an alternative `f` which is a sequence `e`$_1$ `e`$_2$ `...` `e`$_m$ is

```
 $\mathcal{P}$(e$_1$)
 $\mathcal{P}$(e$_2$)
...
 $\mathcal{P}$(e$_m$)
```

where the parse code $\mathcal{P}$`(e`$_i$`)` for each symbol `e`$_i$ is defined below. Note that when the sequence is empty (that is, $m = 0$), the parse code is empty, too.

Nonterminal The parse code $\mathcal{P}$`(A)` for a nonterminal `A` is a call `A();` to its parsing method.

Terminal The parse code $\mathcal{P}$("c") for a terminal "c" depends on its position $\mathtt{e}_j$ in the sequence $\mathtt{e}_1$ ... $\mathtt{e}_m$.

If the terminal is *not* the first symbol $\mathtt{e}_1$, then we must check that c is the current symbol in the token stream ts, and, if so, read the next token:

```
if (ts.tok != c)
  throw ts.parseerror("Expected c");
ts.next();
```

If the terminal *is* the first symbol $\mathtt{e}_1$, then this check has already been made by the switch code for alternatives, so we just need to read the next token:

```
ts.next();
```

Note that in parser construction for grammar rules of form 1 and 2, the grammar requirements ensure that the first-sets are disjoint, so the case-alternatives are all distinct. Also, for rules of form 2, the grammar requirements ensure that every first-set is disjoint from the follow-set of A, so an $\mathtt{f}_i$ alternative is never wrongly chosen instead of the default alternative (for Λ), which occurs last.

Example 9 Applying this construction method to the left factorized grammar from Example 4 gives the parser below. The token stream is provided by an object ts of class Tokenstream.

```
class Example9 {

  private Tokenstream ts;

  public Example9(Tokenstream ts)
  { this.ts = ts; }

  void E()
  { T(); Eopt(); return; }

  void Eopt() {
    switch (ts.tok) {
    case '-':
      ts.next(); T(); Eopt(); return;
    default:
      return;
    }
  }

  void T() {
    switch (ts.tok) {
    case '0':
      ts.next(); return;
    case '1':
      ts.next(); return;
    default:
      throw ts.parseerror("Expected 0 or 1");
    }
  }
```

```
    void parse() {
      E();
      if (ts.tok != Tokenstream.EOF)
        throw ts.parseerror("Expected end of file");
      return;
    }
  }
```

To demonstrate the construction method we have followed it mindlessly in this example. Parts of the program may be improved, but it is advisable to postpone such improvements until you have acquired more practice with parser construction. □

The parser may be used as follows to read file `"example1.zo"` and check that it is well-formed:

```
  class TestExample9 {
    public static void main(String[] args) {
      Tokenstream ts = new ZeroOnescan("example1.zo");
      Example9 p = new Example9(ts);
      p.parse();
    }
  }
```

The subclass ZeroOnescan of Tokenstream is defined in Example 10 below.

4.3 Parsing methods follow the derivation tree

It is interesting to consider how the token stream `<'0', '-', '1', EOF>`, which corresponds to the input `"0" "-" "1"`, is parsed by the parsing methods above. It turns out that the sequence of calls closely follows the derivation tree:

```
E calls T with <'0','-','1',EOF>
    T finds a '0'; now <'-','1',EOF> remains
E calls Eopt with <'-','1',EOF>
    Eopt finds a '-'; now <'1',EOF> remains
    Eopt calls T with <'1',EOF>
        T finds a '1' and returns to Eopt; now <EOF> remains
    Eopt calls a second instance of Eopt with <EOF>
        Eopt finds EOF and returns to the first Eopt; now <EOF> remains
    Eopt returns to E
```

If we draw this sequence of calls as a tree, we get precisely the derivation tree in Figure 3. The parsing methods walk through the derivation tree from left to right. This is no coincidence, but a result of the systematic construction shown above: the derivation tree corresponds to a particular string and a particular grammar, and the parser was constructed systematically from this grammar.

4.4 Summary of parser construction

We have shown a systematic way to construct a parser skeleton from a grammar satisfying the requirements in Figure 6. The parser skeleton just checks that the input, which is a list of terminal symbols, follows the grammar. In Section 6 we show how to make the parser return more information, such as an internal representation of the input.

The parser in Example 9 can be found in file `Example9.java`.

5 Scanners

Now we shall see how to represent a token stream. As explained above, input files are character streams, but it is inconvenient to work with the bare characters. Therefore parsing of text files is usually divided into two phases:

In the *first* phase, the character stream is converted to a stream of *tokens*, and lay-out information (such as blanks, newlines, and tabulation characters) in the input text is removed. This is called *scanning* or *lexical analysis* and is explained in this section.

In the *second* phase, the stream of tokens is parsed as described in the previous sections.

The division into two phases gives a convenient way to allow any number of blanks *between* numerals and names without allowing blanks *inside* numerals and names. By a *blank* we mean a space character, a tabulation character, or a newline. The scanner decides what is a numeral, what is a name, and so on, and throws away all extra blanks. Then the parser never sees a blank: only numerals, names, and so on.

Although a scanner could be constructed systematically from a grammar for terminal symbols, we will not do that here.

We said in Section 4.1 that a token stream object must have a field `ts.tok` which tells us what the current token is, and a method `ts.next()` to read the next token in the stream. In addition it must have a field `nval` for representing the value of a numeric token, a field `sval` for representing the value of a word token, and a method for raising an exception in case of a parse error. We represent these requirements by the following abstract classes Tokenstream and Parseerror:

```
abstract class Tokenstream {

  final static int DOUBLE =  -97;
  final static int NAME   =  -98;
  final static int STRING =  -99;
  final static int EOF    = -100;

  public int tok;
  public double nval;
  public String sval;

  abstract public void next();

  public IllegalArgumentException parseerror(String msg)
  { return new Parseerror(msg + " but found " + this); }
}

class Parseerror extends IllegalArgumentException {
  public Parseerror(String s)
  { super(s); }
}
```

The token constants `DOUBLE`, `NAME`, `STRING`, and `EOF` represent floating-point numerals, names, string constants, and the end-of-file marker. See Sections 5.1, 5.2, and 5.4 below.

To implement concrete subclasses of this class, we use the StreamTokenizer class from the Java library package java.io to split an input character stream into token stream. We might just use the StreamTokenizer class itself, but this turns out to be insufficient when we want to scan languages that have keywords or reserved names; see Section 5.3.

Example 10 The token stream object below could be used with the parser in Example 9. It ignores all blanks, and considers all characters other than '`-`', '`0`' and '`1`' illegal.

```
import java.io.*;

class ZeroOnescan extends Tokenstream {
  private StreamTokenizer strtok;

  public ZeroOnescan(String filename) {
    try {
      strtok = new StreamTokenizer(new FileReader(filename));
      setup();
      next();
    } catch (IOException e)
      { throw new Tokenerror("Error opening file " + filename); }
  }

  private void setup() {
    strtok.resetSyntax();
    strtok.whitespaceChars(' ', ' ');
    strtok.whitespaceChars('\t', '\t');
    strtok.whitespaceChars('\n', '\n');
    strtok.whitespaceChars('\r', '\r');
  }

  public void next() {
    if (tok != EOF)
      try {
         tok = strtok.nextToken();
         switch (tok) {
         case StreamTokenizer.TT_EOF:
           tok = EOF; break;
         case '0': case '1': case '-':
           break;
         default:
           throw new Tokenerror("Illegal token " + this);
         }
      } catch (IOException e) { throw new Tokenerror(e.getMessage()); }
  }

  public String toString() {
    switch (tok) {
    case StreamTokenizer.TT_EOF: case EOF:
      return "<EOF>";
    default:
      return "" + (char)tok;
    }
  }
}

class Tokenerror extends IllegalArgumentException {
  public Tokenerror(String s)
  { super(s); }
}
```

□

The token stream is used as follows with the parser from Example 9 to scan and parse the contents of a file `example1.zo`:

```
class TestExample9 {
  public static void main(String[] args) {
    Tokenstream ts = new ZeroOnescan("example1.zo");
    Example9 p = new Example9(ts);
    p.parse();
  }
}
```

5.1 Scanning floating-point numerals

A numeral is a string of characters that represents a number, such as `3.1414` or `-4.0`. Floating-point numeral can be described by this grammar:

```
Real   = "-" Digits "." Digits | Digits "." Digits .
Digits = Digit | Digit Digits .
Digit  = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

We can make StreamTokenizer recognize numbers by adding this line to method `setup`:

```
strtok.parseNumbers();
```

Then we just need to create a `DOUBLE` token when meeting a `TT_NUMBER` in the `next` method:

```
case StreamTokenizer.TT_NUMBER:
  nval = strtok.nval; tok = DOUBLE; break;
```

5.2 Scanning names

Suppose we need to scan and parse an input language (such as a programming language) which contains *names* of variables, procedures, or similar. Names are also called *identifiers*. A name in this sense is typically a nonempty sequence of letters and digits, beginning with a letter, and containing no blanks. Names can be described by this grammar:

```
Name     = Letter Letdigs .
Letdigs  = Letter Letdigs | Digit Letdigs | Λ .
Letter   = "A" | ... | "Z" | "a" | ... | "z" .
Digit    = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

Since our token stream is based on StreamTokenizer, it is easy to make it recognize such names. We just need to tell the StreamTokenizer, in method `setup`, what characters can make up a name:

```
strtok.wordChars('a', 'z');
strtok.wordChars('A', 'Z');
strtok.wordChars('0', '9');
```

Then we define, in method `next`, that `TT_WORD` should be turned into a `NAME` token:

```
case StreamTokenizer.TT_WORD:
  sval = strtok.sval; tok = NAME; break;
```

5.3 Distinguishing names from keywords

Most (programming) languages have so-called *keywords* or *reserved names* which are sequences of letters that cannot be used as names. For instance, Java and C++ have the keywords '`class`', '`while`', '`do`', etc., which cannot be used as variable names.

Each keyword should be represented by a distinct token, not as yet another name. For instance, if '`class`', '`while`', and '`do`' are keywords, then the corresponding tokens may be `CLASS`, `WHILE`, and `DO`, defined as suitable integer constants:

```
final static int CLASS  = -101;
final static int WHILE  = -102;
final static int DO     = -103;
...
```

The token stream may distinguish names from keywords as follows. Whenever a token that looks like a name has been found, its string value is compared to the list of keywords. The token is classified as a keyword if it is in the list, otherwise it is classified as a name. For example, the following extension of the above scanner's the `next` method will distinguish keywords from names:

```
case StreamTokenizer.TT_WORD:
  sval = strtok.sval;
  if (sval.equals("class"))      tok = CLASS;
  else if (sval.equals("while")) tok = WHILE;
  else if (sval.equals("do"))    tok = DO;
  else ...
  else tok = NAME;
  break;
```

If the set of keywords is large, one may use more efficient means to find the token corresponding to a given string. For instance, the Hashtable class from the java.util package may be useful.

5.4 Scanning string constants

Most (programming) languages have a notion of a string constant, which consists of a sequence of characters enclosed in double quotes, such as `"Hello, world"`. We can make the token stream recognize string constants by adding the following line to the `setup` method:

```
strtok.quoteChar('"');
```

The `next` method must recognize this as a `STRING` token, and store the string's contents in the `sval` field:

```
case '"':
  sval = strtok.sval; tok = STRING; break;
```

5.5 Summary of scanning: token streams

Scanning is the first phase in the parsing of a text. It turns a stream of characters into a stream of tokens, which is then read by a parser in the second phase.

6 Parsers with attributes

So far a parser just checks that an input string can be generated by the grammar: only the *form* or *syntax* of the input is handled. Of course we usually want to know more about the input, so we extend the parsers to return a representation of the input.

For this, every parsing method must return an additional result. Some parsing methods take an additional parameter too. The additional parameters and results are called *attributes*.

6.1 Constructing attributed parsers

We still use parser skeletons constructed as in Section 4.2, but we add code to handle the attributes. So far a parsing method `A` has had type

```
void A() { ... }
```

That is, it has taken no arguments are returned no result. But from now on its type will be

```
outvaluetype A(invaluetype inval)
```

where `invaluetype` and `outvaluetype` are the types of the attributes. We call `A` an *attributed parser*. The `inval` argument is called an *inherited attribute* and the return value is called a *synthesized attribute*. One may decorate parse trees with attribute values. An inherited attribute is sent *down* the tree as an argument to a parsing method, and a synthesized attribute is sent *up* the tree as a result from a parsing method.

Some parsing methods do not take any inherited attributes, but most attributed parsing methods return a synthesized attribute. An attributed parsing method `A` either returns the synthesized attribute, or throws an exception.

One cannot say in general how to turn a parser skeleton into an attributed parser. What extensions and changes are required depends on the kind of information we need about the parsed input. Below we consider a typical example: simple expressions.

The main method `parse` of a parser now either returns a result or raises an exception. For instance, if the type `outvaluetype` is really `double`, then `parse` is written like this:

```
double parse() {
  double ev = E();
  if (ts.tok != Tokenstream.EOF)
    throw ts.parseerror("Expected end of file");
  return ev;
}
```

where `E` is the starting symbol of the grammar. In the examples and exercises below, method `parse` will always have this form, and is therefore not shown.

Arithmetic expressions are usually evaluated from left to right. One also says that the arithmetic operators, such as '`-`', *associate to the left*, that is, group to the left.

Example 11 Recall the parser skeleton for simple arithmetic expressions in Example 9. We extend it so that every parsing method returns a synthesized attribute which is the value of the expression parsed by that method. To evaluate from left to right, we also extend method `Eopt` with an inherited attribute `inval` which at any point is the value of the expression parsed so far. When a `T` is parsed in the `'-'` branch of method `Eopt`, its value `tv` is subtracted from `inval`, and the result is passed to `Eopt` in the recursive call.

```
class Example11 {

  private Tokenstream ts;

  public Example11(Tokenstream ts)
  { this.ts = ts; }

  double E() {
    double tv = T();
    double ev = Eopt(tv);
    return ev;
  }

  double Eopt(double inval) {
    switch (ts.tok) {
    case '-':
      ts.next();
      double tv = T();
      double ev = Eopt(inval - tv);
      return ev;
    default:
      return inval;
    }
  }

  double T() {
    switch (ts.tok) {
    case '0':
      ts.next(); return 0;
    case '1':
      ts.next(); return 1;
    default:
      throw ts.parseerror("Expected 0 or 1");
    }
  }

  double parse() { ... }
}
```

Method `E` first calls `T`. Method `T` parses a `"0"` or a `"1"`, and returns the integer 0 or 1, which gets bound to `tv`. This value is passed to `Eopt` as an inherited attribute `inval`. In method `Eopt` there are two possibilities: *either* it parses `"-" T Eopt`: calls `T` again, subtracts the new `T`-value `tv` from `inval`, and passes the difference to `Eopt` in the recursive call, *or* it parses Λ, in which case it just returns `inval`.

At any point, `inval` is the value of the expression parsed so far. When the input is empty, `inval` is the value of the entire expression. □

Figure 7 shows the attribute values when parsing the input string `"0" "-" "1" "-" "1"` and evaluating from left to right, as done by the parser above. The inherited attribute `inval` is shown to the left of the lines, and the synthesized attributes are shown to the right.

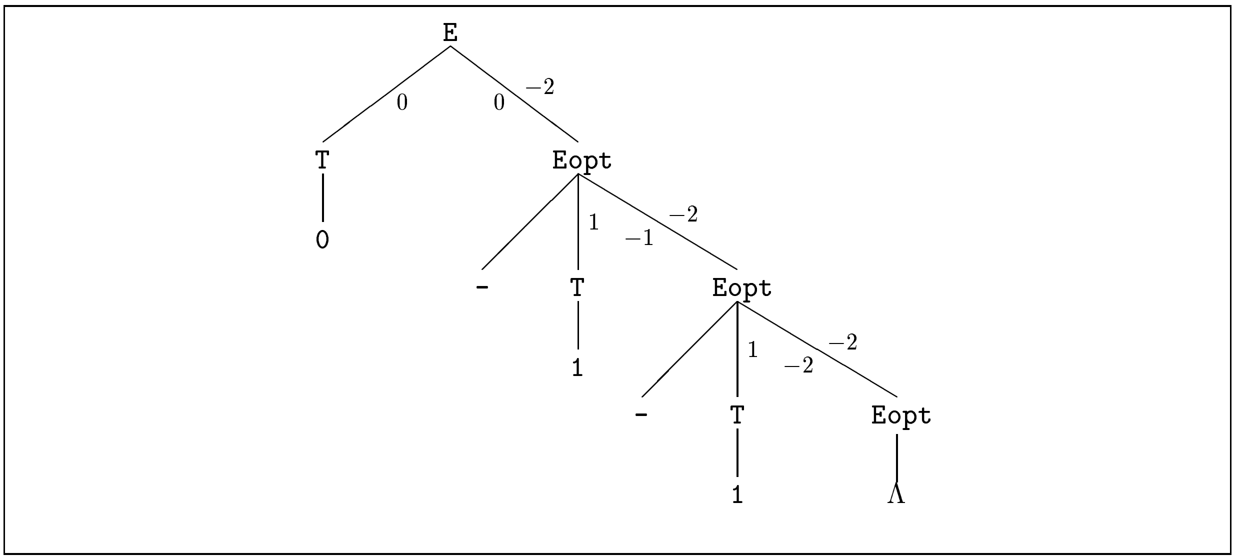


Figure 7: Parse tree with attributes for left-to-right evaluation

A typical use of the attributed parser is

```
class TestExample11 {
  public static void main(String[] args) {
    Tokenstream ts = new ZeroOnescan("example1.zo");
    Example11 p = new Example11(ts);
    System.out.println(p.parse());
  }
}
```

Some parsing methods could be simplified. For instance, the method `E` could be simplified to:

```
double E()
{ return Eopt(T()); }
```

Such simplifications are best done after the parser has been written. Their effect on execution time is limited, so they are mostly of cosmetic value. Also, simplifications require a good understanding of expression evaluation order in Java; otherwise one may introduce subtle errors.

Above we defined left-to-right evaluation of arithmetic expressions, which is usual in programming languages. What if we had a bizarre desire to evaluate from right to left (as in the programming language APL)? This can be done with a small change to the attributed parser above.

Example 12 The attributed parser in Example 11 can be changed to evaluate the expression from right to left as follows:

```
class Example12 {

  private Tokenstream ts;

  public Example12(Tokenstream ts)
  { this.ts = ts; }

  double E() {
    double tv = T();
    double ev = Eopt(tv);
    return ev;
  }

  double Eopt(double inval) {
    switch (ts.tok) {
    case '-':
      ts.next();
      double tv = T();
      double ev = Eopt(tv);
      return inval - ev;
    default:
      return inval;
    }
  }

  double T() {
    switch (ts.tok) {
    case '0':
      ts.next(); return 0;
    case '1':
      ts.next(); return 1;
    default:
      throw ts.parseerror("Expected 0 or 1");
    }
  }

  double parse() { ... }
}
```

The only change is in the `'-'` branch of the `Eopt` method. The subtraction is now done *after* the recursive call to `Eopt`.

At any point, `inval` is the value (0 or 1) of the last `T` parsed. The value `ev` of the remaining expression is subtracted from `inval` *after* the recursive call to `Eopt`. No subtractions are done until the entire expression has been parsed; and then they are done from right to left. □

Figure 8 shows the attribute values when parsing the input string `"0" "-" "1" "-" "1"`, and evaluating from right to left, as done by the parser above. The inherited attribute `inval` is shown to the left of the lines, and the synthesized attributes are shown to the right.

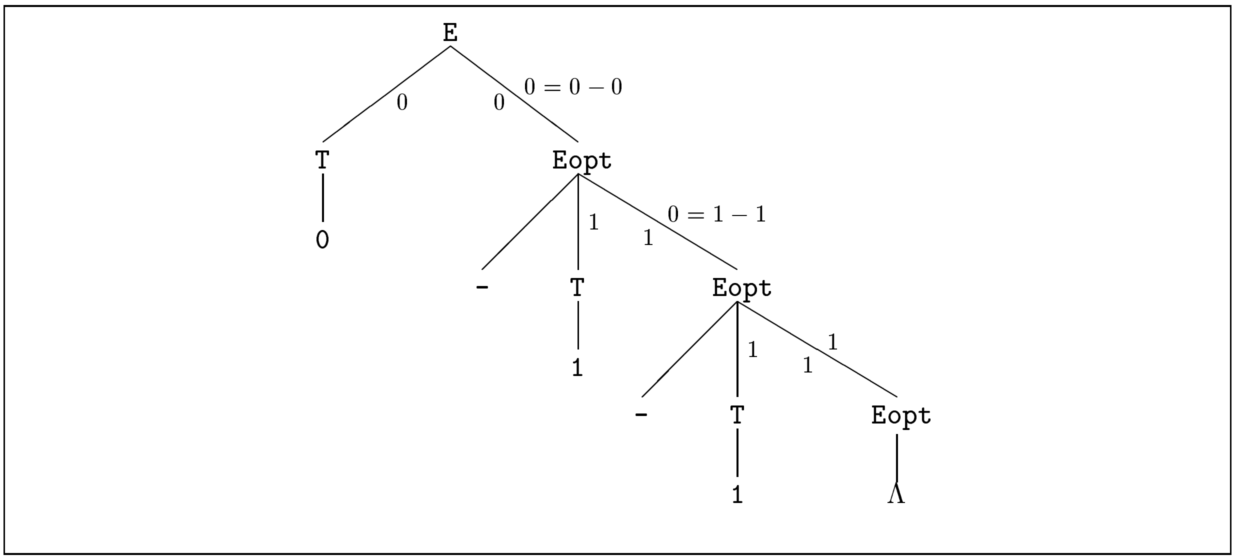


Figure 8: Parse tree with attributes for right-to-left evaluation

6.2 Building representations of input

An important application of attributed parsers is to build representations of the input that has been read by the parser. Such representations are often called abstract syntax trees. An *abstract syntax tree* is a representation of a text which shows the structure of the text and leaves out irrelevant information, such as the number of blanks between symbols.

A flexible and general way to represent abstract syntax trees in Java is to use classes and subclasses. For instance, to represent simple expressions as defined in Example 2, one may define an abstract class Expr of expressions, and concrete classes Zero, One, and Minus, corresponding to each of the three kinds of expressions:

```
abstract class Expr
{ abstract public String toString(); }

class Zero extends Expr {
  public String toString()
  { return "0"; }
}

class One extends Expr {
  public String toString()
  { return "1"; }
}

class Minus extends Expr {
  Expr e1, e2;

  public Minus(Expr e1, Expr e2)
  { this.e1 = e1; this.e2 = e2; }

  public String toString()
  { return e1 + "-" + e2; }
}
```

These class declarations say: there is an abstract concept Expr of expression. There are concrete expression concepts Zero and One, corresponding to `0` and `1`. There is a concrete expression concept Minus, which consists of two subexpressions `e1` and `e2`. Any object belonging to class Expr, or one of its subclasses, knows how to convert itself to a String.

Thus the expression `0-1` can be represented as `new Minus(new Zero(), new One())`. The expression `0-1-1` can be represented either as `new Minus(new Minus(new Zero(), new One()), new One())` or as `new Minus(new Zero(), new Minus(new One(), new One()))`. The first representation corresponds to a left-to-right reading, and the second one corresponds to a right-to-left reading.

Let us make an attributed parser which builds the representation corresponding to a left-to-right reading of simple arithmetic expressions. Such a parser will be very similar to the parser for left-to-right evaluation in Example 11.

Example 13 This parser builds abstract syntax trees for simple arithmetic expressions.

```
class Example13 {

  private Tokenstream ts;

  public Example13(Tokenstream ts)
  { this.ts = ts; }

  Expr E() {
    Expr tv = T();
    Expr ev = Eopt(tv);
    return ev;
  }

  Expr Eopt(Expr inval) {
    switch (ts.tok) {
    case '-':
      ts.next();
      Expr tv = T();
      Expr ev = Eopt(tv);
      return new Minus(inval, ev);
    default:
      return inval;
    }
  }

  Expr T() {
    switch (ts.tok) {
    case '0':
      ts.next(); return new Zero();
    case '1':
      ts.next(); return new One();
    default:
      throw ts.parseerror("Expected 0 or 1");
    }
  }

  Expr parse() { ... }
}
```

Instead of returning an integer (0 or 1), method `T` now returns the representation of an expression: an object of class `Zero` or `One`. Instead of subtracting one number from another, returning a number, method `Eopt` now builds and returns a representation of an expression (in the `'-'` branch).

Since the new representation is built before `Eopt` is called recursively to parse the rest of the expression, the representation is built from left to right as in Example 11. At any point, `inval` is the representation of the expression parsed so far. □

The new parsing methods return a representation of the parsed expression rather than its value. Hence they have return type `Expr`, not `double`. A typical application of the attributed parser in Example 13 would look like this:

```
class TestExample13 {
  public static void main(String[] args) {
    Tokenstream ts = new ZeroOnescan("example1.zo");
    Example13 p = new Example13(ts);
    System.out.println(p.parse());
  }
}
```

where ZeroOnescan is the scanner class defined in Example 10. Calling method `p.parse()` will scan and parse the file `"example1.zo"` and build a representation of its contents, as an object of class Expr. Printing this object will invoke its `toString()` method to convert the object to a string.

6.3 Summary of parsers with attributes

To make parsing methods return information about the input, we add new components to their results and (possibly) to their arguments. Different ways of handling the new results and arguments give different effects, such as left-to-right or right-to-left evaluation. Looking at parse trees is helpful for understanding attribute evaluation.

An abstract syntax tree is a representation of a text without unnecessary detail. Parsers can be extended with attributes to construct the abstract syntax tree for a text while parsing it.

7 A larger example: arithmetic expressions

We now consider arithmetic expressions such as `4.0+5.0*7.0` and `(20.0-5.0)/3.0`, which are found in almost all programming languages, and show how to scan, parse, and evaluate them.

7.1 A grammar for arithmetic expressions

Here is a first attempt at a grammar for arithmetic expressions:

```
E    = E "+" E
     | E "-" E
     | E "*" E
     | E "/" E
     | Real
     | "(" E ")" .
```

This grammar does not satisfy the grammar requirements, but could easily be transformed to do so. However, the grammar does not express the structure of arithmetic expressions very well. In arithmetics, the multiplication and division operators bind more strongly than addition and subtraction. Thus `4.0+5.0*7.0` should be thought of as `4.0+(5.0*7.0)`, giving 39, and not as `(4.0+5.0)*7.0`, giving 63. We say that multiplication and division have higher *precedence* than addition and subtraction.

A subexpression which is a numeral or a parenthesized expression is called a *factor*. A subexpression involving only multiplications and divisions of factors is called a *term*. An expression is a sequence of additions or subtractions of terms.

Then the precedence can be expressed as follows: Factors must be evaluated first, and then terms must be evaluated before additions and subtractions.

To ensure that terms are parsed as units, we introduce a separate nonterminal `T` for them, and similarly for factors `F`. This gives the following grammar for arithmetic expressions:

```
E    = E "+" T | E "-" T | T .
T    = T "*" F | T "/" F | F .
F    = Real | "(" E ")" .
```

The rule for `E` generates strings of form `T "+" T "+"` ⋯ `"-" T` with one or more `T`'s separated by additions and subtractions. Similarly, the rule for `T` generates `F "*" F "*"` ⋯ `"/" F` with one or more `F`'s. Note that `Real` stands for a class of terminal symbols: the floating-point numerals.

To avoid the left recursive rules, we transform the `E` and `T` rules as described in Section 3.4. We obtain the following grammar:

```
E    = T Eopt .
Eopt = "+" T Eopt | "-" T Eopt | Λ .
T    = F Topt .
Topt = "*" F Topt | "/" F Topt | Λ .
F    = Real | "(" E ")" .
```

This grammar satisfies the requirements in Figure 6, as argued in Example 8.

7.2 The parser constructed from the grammar

The terminal symbols of the grammar are the operators '`+`', '`-`', '`*`', '`/`', the parentheses '`(`' and '`)`', and the floating-point numerals.

Application of the construction method from Section 4.2 to the above grammar gives the parser skeleton shown below. (File `Aritskel.java` contains a copy of the parser skeleton).

```
class Aritskelparser {
  private Tokenstream ts;

  public Aritskelparser(Tokenstream ts)
  { this.ts = ts; }

  void E()
  { T(); Eopt(); return; }

  void Eopt() {
    switch (ts.tok) {
    case '+':
      ts.next(); T(); Eopt(); return;
    case '-':
      ts.next(); T(); Eopt(); return;
    default:
      return;
    }
  }

  void T()
  { F(); Topt(); return; }

  void Topt() {
    switch (ts.tok) {
    case '*':
      ts.next(); F(); Topt(); return;
    case '/':
      ts.next(); F(); Topt(); return;
    default:
      return;
    }
  }

  void F() {
    switch (ts.tok) {
    case Aritscan.DOUBLE:
      ts.next(); return;
    case '(':
      ts.next();
      E();
      if (ts.tok != ')')
        throw ts.parseerror("Expected ')'");
      ts.next();
      return;
    default:
      throw ts.parseerror("Expected number or '('");
  } }

  void parse() { ... }
}
```

7.3 A scanner for arithmetic expressions

An appropriate scanner is shown below. File `Aritscan.java` contains a copy of this scanner.

```
import java.io.*;

class Aritscan extends Tokenstream {
  private StreamTokenizer strtok;

  public Aritscan(String filename) {
    try {
      strtok = new StreamTokenizer(new FileReader(filename));
      setup(); next();
    } catch (IOException e)
      { throw new Tokenerror("Error opening file " + filename); }
  }

  private void setup() {
    strtok.resetSyntax();
    strtok.parseNumbers();
    strtok.whitespaceChars(' ', ' ');
    strtok.whitespaceChars('\t', '\t');
    strtok.whitespaceChars('\n', '\n');
    strtok.whitespaceChars('\r', '\r');
  }

  public void next() {
    if (tok != EOF)
      try {
        tok = strtok.nextToken();
        switch (tok) {
        case StreamTokenizer.TT_EOF:
          tok = EOF; break;
        case StreamTokenizer.TT_NUMBER:
          nval = strtok.nval; tok = DOUBLE; break;
        case '(': case ')': case '+': case '-': case '*': case '/':
          break;
        default:
          throw new Tokenerror("Illegal token " + this);
        }
      } catch (IOException e) { throw new Tokenerror(e.getMessage()); }
  }

  public String toString() {
    switch (tok) {
    case StreamTokenizer.TT_EOF: case EOF:
      return "<EOF>";
    case StreamTokenizer.TT_NUMBER: case DOUBLE:
      return "" + strtok.nval;
    case StreamTokenizer.TT_WORD:
      return "" + strtok.sval;
    default:
      return "" + (char)tok;
} } }
```

7.4 Evaluating arithmetic expressions

Now we extend the parser skeleton from Section 7.2 to evaluate the arithmetic expressions while parsing them. As observed previously, arithmetic expressions should be evaluated from left to right, so the resulting attributed parser below is similar to that in Example 11, except that it has been simplified by hand. (File `Ariteval.java` contains a copy of this parser).

```
class Aritevalparser {
  private Tokenstream ts;

  public Aritevalparser(Tokenstream ts)
  { this.ts = ts; }

  double E() { return Eopt(T()); }

  double Eopt(double inval) {
    switch (ts.tok) {
    case '+':
      ts.next(); return Eopt(inval + T());
    case '-':
      ts.next(); return Eopt(inval - T());
    default:
      return inval;
  } }

  double T() { return Topt(F()); }

  double Topt(double inval) {
    switch (ts.tok) {
    case '*':
      ts.next(); return Topt(inval * F());
    case '/':
      ts.next(); return Topt(inval / F());
    default:
      return inval;
  } }

  double F() {
    switch (ts.tok) {
    case Aritscan.DOUBLE:
      ts.next(); return ts.nval;
    case '(':
      ts.next();
      double ev = E();
      if (ts.tok != ')')
        throw ts.parseerror("Expected ')'");
      ts.next(); return ev;
    default:
      throw ts.parseerror("Expected number or '('");
  } }

  double parse() { ... }
}
```

8 Some background

8.1 History and notation

Formal grammars were developed within linguistics by Noam Chomsky around 1956, and were first used in computer science by John Backus and Peter Naur in 1960 to describe the Algol programming language. Their notation was subsequently called *Backus-Naur Form* or *BNF*. In the original BNF notation, our grammar from Example 4 would read:

```
<E>    ::= <T> <Eopt>
<Eopt> ::=   | - <T> <Eopt>
<T>    ::= 0 | 1
```

This notation uses a different convention than ours: nonterminals are surrounded by angular brackets, and terminals are not quoted. Also, here the empty string Λ is denoted by nothing (empty space). In compiler books one may find still another notation:

```
E    → T Eopt
Eopt → ϵ
Eopt → - T Eopt
T    → 0
T    → 1
```

In this notation there is only one alternative per rule, so defining a nonterminal may require several rules. Also, ϵ is used instead of our Λ.

As can be seen, the actual notation used for grammars varies, and combinations of these notations exist also. However, the underlying idea of derivation is always the same.

8.2 Extended Backus-Naur Form

Our grammar notation is a simplification of the so-called *Extended Backus-Naur Form* or *EBNF*. The full EBNF notation contains more complicated forms of alternatives `f`.

In EBNF, an *alternative* `f` is a *sequence* $\mathtt{e}_1$ `...` $\mathtt{e}_m$ of elements, not just symbols. An *element* `e` may be a symbol as before, or

- an *option* of form `[ f ]`, which can derive zero or one occurrence of sequence `f`, or
- a *repetition* of form `{ f }`, which can derive zero, one, or more occurrences of `f`, or
- a *grouping* of form `( f )`, which can derive an occurrence of `f`.

A grammar in EBNF notation using the new kinds of elements can be converted to a grammar in our notation. The conversion is done by introducing extra nonterminals and rules:

- an option `[ f ]` is replaced by a new nonterminal `Optf` with rule `Optf = f |` Λ`.`
- a repetition `{ f }` is replaced by a new nonterminal `Repf` with rule `Repf = f Repf |` Λ`.`
- a grouping `( f )` is replaced by a new nonterminal `Grpf` with rule `Grpf = f.`

This shows that our simple grammar notation can express everything that EBNF can, possibly at the expense of introducing more nonterminals.

8.3 Classes of languages

The parsing method described in Section 4 is called *recursive descent* parsing and is an example of a *top-down* parsing method. It works for a class of grammars called $LL(1)$: those that can be parsed by reading the input symbols from the *Left*, making derivations always from the *Leftmost* nonterminal, and using a lookahead of *1* input symbol. This class includes all grammars that satisfy the requirements in Figure 6.

Another well-known class of grammars, more powerful than $LL(1)$, is the $LR(1)$ class which can be parsed *bottom-up*, reading the input symbols from the *Left*, making derivations always from the *Rightmost* nonterminal, and using a lookahead of *1* input symbol. Construction of bottom-up parsers is complicated, and is seldom done by hand. A useful subclass of $LR(1)$ is the class $LALR(1)$ (for 'lookahead LR'), which can be parsed more efficiently, by smaller parsers. Automatic $LALR(1)$ parser generators exist for most programming languages. For Java, use '`JavaCC`' or '`Java CUP`'. For Standard ML, use '`mosmlyac`' or '`ML-Yacc`'. For C, use the classic '`Yacc`' or '`Bison`'. The $LR(1)$ grammars are sufficiently powerful for most computing problems, but as exemplified by Exercise 4 there are grammars for which there is no equivalent LR grammar (and consequently no $LALR(1)$-grammar or LL-grammar).

The class of grammars defined in Figure 1 is properly called the *context-free grammars*. This is just one class in the hierarchy identified by Chomsky: (0) the *unrestricted*, (1) the *context-sensitive*, (2) the *context-free*, and (3) the *regular* grammars. The unrestricted grammars are more powerful than the context-sensitive ones, which are more powerful than the context-free ones, which are more powerful than the regular grammars.

The unrestricted grammars cannot be parsed in general; they are of theoretical interest but of little practical use in computing. All context-sensitive grammars can be parsed, but may take an excessive amount of time and space, and so are of little practical use. The context-free grammars are those defined in Figure 1; they are highly useful in computing, in particular the subclasses $LL(1)$, $LALR(1)$, and $LR(1)$ mentioned above. The regular grammars can be parsed very efficiently using a constant amount of memory, but they are rather weak; they cannot define parenthesized arithmetic expressions, for instance.

The following table summarizes the grammar classes:

Chomsky hierarchy	Example rules	Comments	
0: Unrestricted	`"a" B "b" → "c"`	Rewrite system	
1: Context-sensitive	`"a" B "b" → "a" "c" "b"`	Non-abbreviating rewrite system	
2: Context-free	`B → "a" B "b"`	As defined in Figure 1. Some interesting subclasses:	
		$LR(1)$	bottom-up parsing
		$LALR(1)$	bottom-up, 'Yacc'
		$LL(1)$	top-down, these notes
3: Regular	`B → "a" | "a" B`	parsing by finite automata	

8.4 Further reading

A description (in Danish) of practical recursive descent parsing using Turbo Pascal is given by Kristensen [3].

There is a rich literature on scanning and parsing in connection with compiler construction. The standard reference is Aho, Sethi, and Ullman [1]. More information on recursive descent parsing is found in Lewis, Rosenkrantz, and Stearns [4], and in Wirth [5, Chapter 5].

9 Exercises

Exercise 1 Write down a grammar for arrays of (unsigned) integers. For instance, the empty array of integers is {}. Other examples of lists of integers are `[117]`, `[2,3,5,7,11,13]`. Show the derivations of `[]` and `[7, 9, 13]`. □

Exercise 2 Consider the grammar

```
E = T "+" E | T "-" E | T .
T = "0" | "1" .
```

Left factorize it and find selection sets for the alternatives of the resulting grammar. □

Exercise 3 Consider the grammar below, which is self left recursive:

```
S = S S | "0" | "1" .
```

Apply the technique for removing left recursion (Section 3.4). Find first-, follow-, and selection sets for the resulting grammar. Does it satisfy the grammar requirements?

What strings are derivable from this grammar? Find a grammar which generates the same strings and satisfies the requirements (this is quite easy). □

Exercise 4 The grammar

```
P = "a" P "a" | "b" P "b" | Λ .
```

generates palindromes (strings which are equal to their reverse). Find first-, follow-, and selection sets for this grammar. Which requirement in Figure 6 is not satisfied? (In fact, there is no way to transform this grammar into one that satisfies the requirements). □

Exercise 5 Consider the grammar in Exercise 2. Left factorize it. Construct a parser skeleton for the left factorized grammar, using the tokens '`+`', '`-`', '`0`', and '`1`'. □

Exercise 6 The grammar

```
T = "0" | "1" | "(" T ")" .
```

describes simple expressions such as `1`, `(1)`, `((0))`, etc. with well-balanced parentheses. Choose a suitable set of tokens to represent the terminal symbols, and construct a Java parser for the grammar. Test it on the expressions above, and on some ill-formed inputs. □

Exercise 7 Write a grammar and construct a parser for parenthesized expressions such as `0`, `0+(1)`, `1-(1+1)`, `(0-1)-1`, etc. □

Exercise 8 Consider the grammar for polynomials from Example 3. (1) Remove the left recursion in the rule for `Poly`. (2) Left factorize the rule for `Term`. (3) Choose a suitable set of tokens to represent the terminal symbols. Note that `Natnum` in the grammar stands for a family of terminal symbols `1`, `2`, ...; the terminal symbol `123` could be represented by the token `DOUBLE` with the value 123 held in field `nval` of the Tokenstream. (4) Construct a parser skeleton for the transformed grammar and test it. □

Exercise 9 Show that the requirements in Figure 6 imply that for every grammar rule, and distinct alternatives $\mathtt{f}_i$ and $\mathtt{f}_j$, it holds that $\mathit{Select}(\mathtt{f}_i) \cap \mathit{Select}(\mathtt{f}_j) = \{\}$. □

Exercise 10 The input language for the scanner in Example 10 is described by the grammar:

```
input = "-" input | "0" input | "1" input | blank input | Λ .
blank = " " | "\t" | "\n" .
```

Make sure the grammar satisfies the requirements, then use the construction method of Section 4 to systematically make a scanner for it. Your scanner must check the form of the input, but need not return a list of terminals. □

Exercise 11 Extend the parser constructed in Exercise 5 to evaluate the parsed expression and return its value. You may decide yourself whether evaluation should be from left to right or right to left. □

Exercise 12 Extend the parser constructed in Exercise 5 to build an abstract syntax tree for the parsed expression, using the following classes:

```
abstract class Expr { }

class Zero extends Expr {}

class One extends Expr {}

class Minus extends Expr {
  Expr e1, e2;

  public Minus(Expr e1, Expr e2)
  { this.e1 = e1; this.e2 = e2; }
}

class Plus extends Expr {
  Expr e1, e2;

  public Plus(Expr e1, Expr e2)
  { this.e1 = e1; this.e2 = e2; }
}
```

What are the types of the attributed parsing methods? □

Exercise 13 Extend the scanner from Section 7.3 to recognize Java floating-point numbers with exponents such as '`6.6256E34`' or '`3E8`'. □

Exercise 14 What changes are necessary to make the parser in Example 13 build representations from right to left? □

Exercise 15 Check that the grammar at the end of Section 7.1 satisfies the grammar requirements. □

Exercise 16 Extend the grammar, scanner, and parser from Section 7 to handle arithmetic expressions with exponentiation, such that `3.0*4.0^2.0` evaluates to 48, that is, 3 times the square of 4. Note that the exponentiation operator usually associates to the right and has higher precedence than multiplication and division, so `2.0^2.0^3.0` is `2.0^(2.0^3.0)` and evaluates to 256, not to 64.

What changes are necessary if the exponentiation operator were '`**`' instead of '`^`'? □

Exercise 17 The following classes may be used to represent the arithmetic expressions from Section 7:

```
abstract class Expr { }

class Binop extends Expr {
  Expr e1, e2;
  char op;

  public Binop(Expr e1, char op, Expr e2)
  { this.e1 = e1; this.op = op; this.e2 = e2; }
}

class Real extends Expr {
  double r;

  public Real(double r)
  { this.r = r; }
}
```

Write an attributed parser that builds abstract syntax trees of this form. □

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] H. Elbrønd Jensen and T. Høholdt. *Grundlæggende Matematik for Dataloger.* Matematisk Institut, Danmarks Tekniske Højskole, Lyngby, Danmark, 1993.

[3] J.T. Kristensen. *Konstruktion af indlæseprogrammer.* Teknisk Forlag, 1990.

[4] P.M. Lewis II, D.J. Rosenkrantz, and R.E. Stearns. *Compiler Design Theory.* The Systems Programming Series. Addison-Wesley, 1976.

[5] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.