Grammars and parsing with Standard ML

Peter Sestoft[1]

1994-01-09
Updated 1997-02-04 for Moscow ML 1.41

Updated 2002-10-16 by Ken Friis Larsen[2]

[1] Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Denmark. E-mail: `sestoft@dina.kvl.dk`. These notes were written while at the Technical University of Denmark.

[2]Department of Innovation, IT University of Copenhagen, Denmark. E-mail: `kfl@it.edu`

Contents

1 Grammars and Parsing **4**

2 Grammars **5**
- 2.1 Grammar notation 5
- 2.2 Derivation 5

3 Parsing theory **7**
- 3.1 Parsing: reconstruction of a derivation tree 7
- 3.2 A more machine-oriented view of parsing 9
- 3.3 Left factorization 10
- 3.4 Left recursive nonterminals 11
- 3.5 First-sets, follow-sets and selection sets 12
- 3.6 Summary of parsing theory 15

4 Parser construction in SML **16**
- 4.1 SML representation of input strings 16
- 4.2 Constructing parsing functions in SML 16
- 4.3 Parsing functions follow the derivation tree 18
- 4.4 Summary of parser construction 19

5 Scanners **20**
- 5.1 Character classification functions 21
- 5.2 Scanning names 21
- 5.3 Distinguishing names from keywords 22
- 5.4 Scanning real numerals 22
- 5.5 Reading text files with SML 23
- 5.6 Summary of scanners 24

6 Parsers with attributes **25**
- 6.1 Constructing attributed parsers 25
- 6.2 Building representations of input 27
- 6.3 Summary of parsers with attributes 29

7 A larger example: arithmetic expressions **30**
- 7.1 A grammar for arithmetic expressions 30
- 7.2 The parser constructed from the grammar 31
- 7.3 A scanner for arithmetic expressions 32
- 7.4 Evaluating arithmetic expressions 33

8 Scanner and parser facilities in the Basis Library **34**
- 8.1 Making a `StringCvt.reader` for scanning arithmetic expressions 34
- 8.2 A `StringCvt.reader` for parsing and evaluating arithmetic expressions 35
- 8.3 Using a `StringCvt.reader` 36

9 Some background **37**
9.1 History and notation 37
9.2 Extended Backus-Naur Form 37
9.3 Classes of languages 38
9.4 Further reading 38

10 Exercises **39**

References **41**

Index **42**

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 an Standard ML (SML) datatype, 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[3] of these data. □

This chapter provides 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 SML. The methods shown here are essentially independent of SML, and can be applied also to imperative languages with recursive procedures (Ada, C, 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 chapter, 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 this chapter, keep in mind that although it may look 'theoretical' at places, the goal is to provide a *practically* useful tool.

[3]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`. Usually the starting symbol is listed first. □

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

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

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"`.

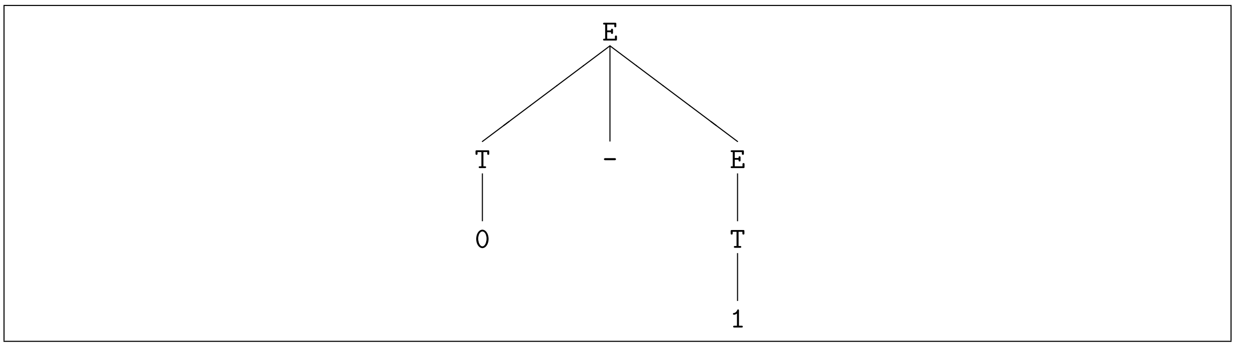


Figure 2: A derivation tree

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 $L(\texttt{A})$:

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

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 $L(\texttt{E}) = \{$`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 SML parser programs working according to this principle.

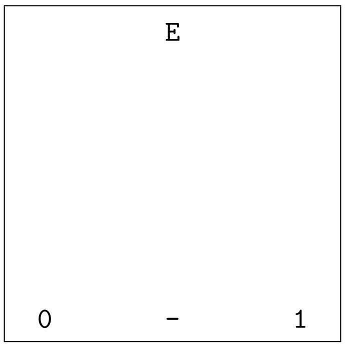
3.1 Parsing: reconstruction of a derivation tree

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:

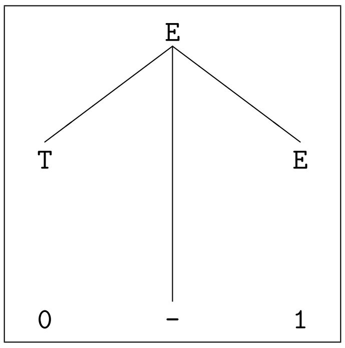
```
E     = T "-" E | T .
T     = "0" | "1" .
```

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:



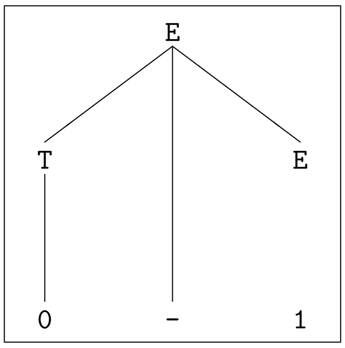
(a)

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):



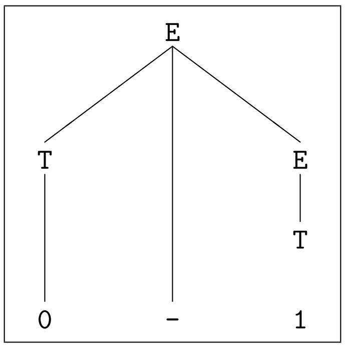
(b)

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):



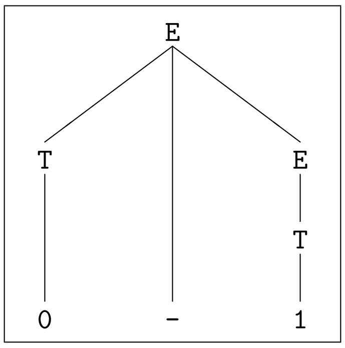
(c)

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.

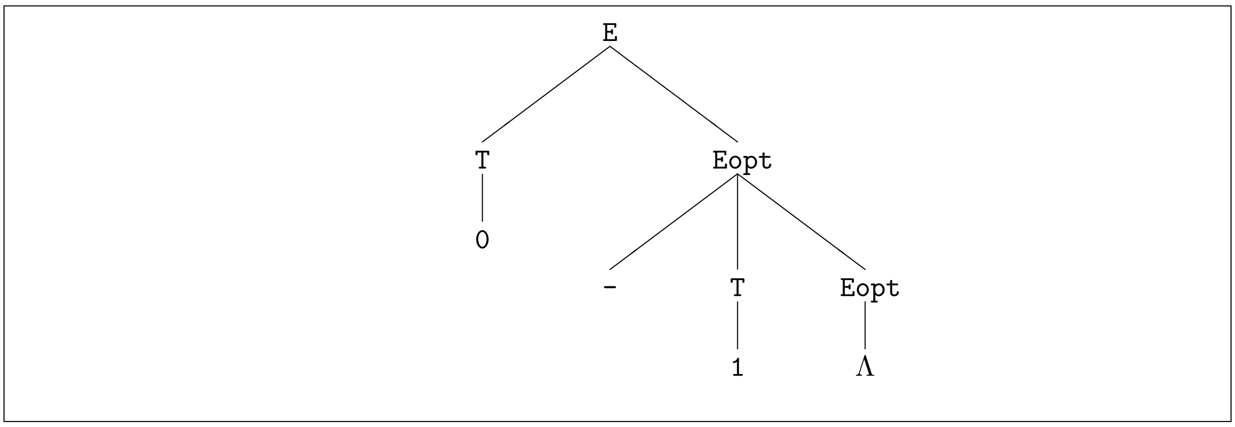


Figure 3: Derivation tree for the left factorized grammar

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:

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

The $\texttt{g}_i$ and $\texttt{f}_j$ stand for sequences of grammar symbols (possibly Λ). We require that $m, n \geq 1$, and that no $\texttt{f}_j$ 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 *First*(f). The set of symbols that can follow a nonterminal A is called its *follow-set* and is written *Follow*(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 *First*(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 *First*("c") = {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 *First*(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 *First*(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, *First*(f) includes $\mathit{First}(\mathtt{e}_2\ \ldots\ \mathtt{e}_m)$ also.

The computation of *First*(f) is summarized in Figure 4.

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

Figure 4: Computation of first-sets

How can we compute the follow-set *Follow*(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*(A) must include everything that f can begin with, and, if f can derive Λ, then also everything that can follow B. This is expressed by Figure 5.

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

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

Figure 5: Computation of follow-sets

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: $\mathit{Select}(\texttt{f}_i) \cap \mathit{Select}(\texttt{f}_j) = \{\}$. 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 $\mathit{First}(\texttt{f}_i) \cap \mathit{First}(\texttt{f}_j) = \{\}$.
- No $\texttt{f}_i$ can derive Λ.
- In rules of form 2, we must have $\mathit{First}(\texttt{f}_i) \cap \mathit{Follow}(\texttt{A}) = \{\}$ 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 *First*("0") = {0} and *First*("1") = {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 *Follow*(T) for the grammar shown above. Consulting Figure 5, we see that *Follow*(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 *Follow*(E)? It is the empty set {}, since *Follow*(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 *Follow*(T) = {-}. □

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 *Follow*(E) in the previous example, we also find that *Follow*(Eopt) = {}.

The first-sets {} and {-} of the two alternatives are disjoint, and *First*("-" T Eopt) ∩ *Follow*(Eopt) = {}, so the `E` rule satisfies the grammar requirements. The selection sets for the two alternatives are {"-"} 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

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

Since *First*(`T`) = {`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.

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 conecpts, 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 SML

We now show a systematic way to write a *parser skeleton* (in SML) 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 SML representation of input strings

In SML we can represent the input string as a *list* of terminal symbols, where the terminal symbols belong to a suitable datatype `terminal`. Thus the input `ts` to a parser has type `terminal list`.

A terminal symbol is represented as a constructor. A simple terminal symbol such as `"-"` may be represented by a parameterless constructor such as `Sub`. However, some terminal symbols occur in families. For instance, the terminal symbol `Real` may stand for all real numbers. Clearly we cannot have an SML constructor for each real number, so terminal symbols belonging to this family will be represented by a parametrized constructor `Real of real`. For example, the nonterminal `666.0` will be represented in SML as `Real(666.0)`.

4.2 Constructing parsing functions in SML

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 functions*, one for each nonterminal in the grammar.

The parsing function corresponding to nonterminal `A` is called `A` also. It takes as input a terminal list `ts`, and tries to find a string derivable from `A` at the beginning of `ts`. If it succeeds, then the rest of `ts` is returned. If it fails, then the exception `Parseerror` is raised, and nothing is returned. Thus a parsing function has type `A: terminal list -> terminal list`.

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

```
datatype terminal = ...
exception Parseerror

fun A₁ ts = ...
and A₂ ts = ...
...
and Aₖ ts = ...

fun parse ts =
    case (S ts) of
        [] => ()
      | _  => raise Parseerror
```

where $\{\mathtt{A}_1, \ldots, \mathtt{A}_k\} = N$ is the set of nonterminals and `S` is the starting symbol. The main function is `parse`; it checks that no input remains after parsing. It returns '`()`' if the parse succeeds, and raises an exception otherwise.

Rule of form 0 The parsing function for a rule of form $\mathtt{A} = \mathtt{f}_1$ is

```
fun A ts = parse code for f₁
```

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

```
fun A ts =
   case ts of
       $t_{11}$ :: tr   => parse code for alternative $f_1$
        ...
     | $t_{1a_1}$ :: tr  => parse code for alternative $f_1$
        ...
     | $t_{n1}$ :: tr   => parse code for alternative $f_n$
        ...
     | $t_{na_n}$ :: tr  => parse code for alternative $f_n$
     | _            => raise Parseerror
```

where $\{t_{i1},\ldots,t_{ia_i}\} = \mathit{First}(f_i)$ is the first-set of alternative f_i, for $i = 1, \ldots, n$.

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

```
fun A ts =
   case ts of
       $t_{11}$ :: tr   => parse code for alternative $f_1$
        ...
     | $t_{1a_1}$ :: tr  => parse code for alternative $f_1$
        ...
     | $t_{n1}$ :: tr   => parse code for alternative $f_n$
        ...
     | $t_{na_n}$ :: tr  => parse code for alternative $f_n$
     | _            => ts
```

where $\{t_{i1},\ldots,t_{ia_i}\} = \mathit{First}(f_i)$ as above.

Sequence The parse code for an alternative `f` which is a sequence e_1 e_2 ... e_m is

```
let val $ts_1$  =  $\mathcal{P}$($e_1$, ts)
    val $ts_2$  =  $\mathcal{P}$($e_2$, $ts_1$)
    ...
    val $ts_m$  =  $\mathcal{P}$($e_m$, $ts_{m-1}$)
in $ts_m$ end
```

where the parse code $\mathcal{P}$(`e`$_i$, `ts`) for each symbol e_i is defined below. Note that when the sequence is empty (that is, $m = 0$), the parse code is just `let in ts end`, or equivalently, `ts`.

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

Terminal The parse code $\mathcal{P}$(`"c", ts`) for a terminal `"c"` depends on its position e_j in the sequence e_1 ... e_m.

If the terminal is *not* the first symbol e_1, then we must check that `c` appears first in `ts`:

```
case ts of
    c :: tr => tr
  | _       => raise Parseerror
```

If the terminal *is* the first symbol e_1, then this check has already been made by the `case` code for alternatives, so the code for the terminal symbol is simply `tr`.

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`-patterns do not overlap. 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 Λ alternative, which occurs last.

Example 8 Applying this construction method to the left factorized grammar from Example 4 gives the parser below. The terminal symbols are represented by the datatype `terminal`.

```
datatype terminal = Sub | Zero | One
exception Parseerror

fun E ts =
    let val ts1 = T ts
        val ts2 = Eopt ts1
    in ts2 end
and Eopt ts =
    case ts of
        Sub :: tr => let val ts1 = tr
                         val ts2 = T ts1
                         val ts3 = Eopt ts2
                     in ts3 end
      | _         => ts
and T ts =
    case ts of
        Zero :: tr => tr
      | One  :: tr => tr
      | _          => raise Parseerror

fun parse ts =
    case (E ts) of
        [] => ()
      | _  => raise Parseerror
```

To demonstrate the construction method we have followed it mindlessly in this example. This gives some clumsy code in the `Sub` branch of the `Eopt` function, where the entire `let` expression can be reduced to `Eopt (T tr)`. Such optimizations can be done after the systematic construction process. □

The parser is used as follows:

```
parse [Zero,Sub,One];
> val it = () : unit

parse [Zero,Sub,Sub];
> uncaught exception Parseerror
```

4.3 Parsing functions follow the derivation tree

It is interesting to consider how the terminal list `[Zero, Sub, One]`, which corresponds to the input `"0" "-" "1"`, is parsed by the parsing functions above. It turns out that the sequence of calls closely follows the derivation tree:

`E` calls `T` with `[Zero,Sub,One]`
    `T` finds a `Zero` and returns `[Sub,One]` to `E`

```
E calls Eopt with [Sub,One]
    Eopt finds a Sub
    Eopt calls T with [One]
        T finds a One and returns [] to Eopt)
    Eopt calls a second instance of Eopt with []
        Eopt finds [] and returns [] to the first Eopt
    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 functions 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 8 can be found in file `gram/ex8.sml`.

5 Scanners

In the parser above the input was represented as a list of terminal symbols. Real input files are character strings (divided into lines), but it is inconvenient to let terminal symbols be just characters. Therefore parsing of text files is usually divided into two phases.

In the first phase, the character string is converted to a string of terminal symbols, and lay-out information (such as blanks) in the input text is removed. This is called *scanning* or *lexical analysis* and is explained below. A *scanner* has type `scan: string -> terminal list`.

In the second phase, the list of terminal symbols 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.

Example 9 The scanner below could be used with the parser in Example 8. It ignores all blanks, and considers all characters other than '`-`', '`0`' and '`1`' as errors.

```
datatype terminal = Sub | Zero | One
exception Scanerror
fun isblank c = (c = #" " orelse c = #"\t" orelse c = #"\n")

fun scan s =
    let fun sc cs =
        case cs of
            []          => []
          | #"-" :: cr  => Sub  :: sc cr
          | #"0" :: cr  => Zero :: sc cr
          | #"1" :: cr  => One  :: sc cr
          | c   :: cr   => if isblank c then sc cr
                           else raise Scanerror
    in sc (explode s) end
```

Note the use of `explode: string -> char list` to split turn a string into a list of characters, and the auxiliary function `isblank: char -> bool`. □

The scanner is used as follows:

```
scan "0-1";
> val it = [Zero,Sub,One] : terminal list

scan "0-  -";
> val it = [Zero,Sub,Sub] : terminal list
```

5.1 Character classification functions

Above we used the SML function `isblank` to decide whether an input character is a blank. A few other functions, also of type `char -> bool`, for classifying characters are shown below:

```
fun isblank c  = (c = #" " orelse c = #"\t" orelse c = #"\n")
fun isdigit c  = (#"0" <= c andalso c <= #"9")
fun islower c  = (#"a" <= c andalso c <= #"z")
fun isupper c  = (#"A" <= c andalso c <= #"Z")
fun isletter c = (islower c orelse isupper c)
fun isletterdigit c = (isletter c orelse isdigit c)
```

Note, that the SML Basis Library contains character classification functions in the structure `Char`. For example, `Char.isSpace` corresponds to the function `isblank` above. In this note we use the explicit declared functions for clarity. See Section 8 for more information about Basis Library facilities.

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" .
```

A scanning function `scname: char list * char list -> char list * string` for names is shown below. It is called as `scname(cr, [c])` from the main scanner when a letter `c` is met.

```
fun scname (cs, value) =
    case cs of
        c :: cr => if isletterdigit c then scname(cr, c :: value)
                   else (cs, implode (rev value))
      | []      => (cs, implode (rev value))
```

It accumulates the characters of the name (in reverse order) in the argument `value` until it meets a character which is neither a letter nor a digit, or until the input ends. It returns a pair of the remaining input `cs` and the name, reversing and imploding the character list to obtain a `string`.

The name scanner is invoked by extending the case expression in the `scan` function as shown below. The terminal symbol representing a name is the constructor `Name` applied to a string, such as `Name "A38"`.

```
datatype terminal = ... | Name of string
fun scan s =
    let fun sc cs =
        case cs of
          ...
          | c :: cr        => if isblank c then sc cr
                              else if isletter c then
                                   let val (cs1, n) = scname(cr, [c])
                                   in Name n :: sc cs1 end
                              else raise Scanerror
    in sc (explode s) end
```

5.3 Distinguishing names from keywords

Most (programming) languages contain so-called *keywords* or *reserved names* which are sequences of letters that cannot be used as names. For instance, Standard SML has the keywords '`let`', '`in`', '`end`' and many others, and C and Pascal have the keywords '`while`', '`do`', and so on.

Keywords are represented by other terminal symbols than names. For instance, if '`let`', '`in`', and '`end`' are keywords, then the corresponding terminal symbols may be `LET`, `IN`, and `END`, instead of `Name "let"` etc.

A scanner distinguishes names from keywords as follows. Whenever something that looks like a name has been found by the scanner, it is compared to the list of keywords. It is classified as a keyword if in the list, otherwise it is classified as a name. For example, the following extension of the scanner above will distinguish keywords from names:

```
datatype terminal = ... | Name of string | LET | IN | END
fun scan s =
    let fun sc cs =
        case cs of
          ...
          | c :: cr        => if isblank c then sc cr
                              else if isletter c then
                                   let val (cs1, n) = scname(cr, [c])
                                   in case n of
                                       "let" => LET    :: sc cs1
                                     | "in"  => IN     :: sc cs1
                                     | "end" => END    :: sc cs1
                                     | _     => Name n :: sc cs1
                                   end
                              else raise Scanerror
    in sc (explode s) end
```

5.4 Scanning real numerals

A numeral is a string of characters that represents a number, such as `"3.1414"` or `"~4.0"`. Let us define a real numeral as follows: a nonempty sequence of digits followed by a point '`.`' followed by a nonempty sequence of digits, the whole possible preceded by a minus sign '`~`', and with no blanks between symbols. Real numerals 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" .
```

The scanning function `screal: char list * real -> char list * real` shown below is called as `screal(cr, realval c)` from the main scanning function when a digit `c`, or a minus sign '~' followed by a digit `c`, has been met by the scanner.

```
fun realval c = real(ord c - ord #"0")
fun screal (cs, value) =
    case cs of
        #"." :: c :: cr => if isdigit c then scfrac(c :: cr, value, 0.1)
                           else raise Scanerror
      | c :: cr         => if isdigit c then screal(cr,10.0*value+realval c)
                           else raise Scanerror
      | []              => raise Scanerror
and scfrac (cs, value, wt) =
    case cs of
        c :: cr => if isdigit c then scfrac(cr, value+wt*realval c, wt/10.0)
                   else (cs, value)
      | []      => (cs, value)
```

The `screal` function accumulates the `value` of the digits before the point until a point is met, or another non-digit is met, or the input ends. In the two latter cases the number is ill-formed and an exception is raised. In the first case, there must be a digit after the point. If there is, then function `scfrac` is called. Function `scfrac` accumulates the `value` of the digits after the point until a non-digit is met. Then it returns a pair of the remaining input `cs` and the value of the entire number.

The scanner function for reals is invoked by modifying the `case` expression in the `scan` function of Example 9 as follows

```
fun scan s =
    let fun sc cs =
        case cs of
          ...
          | #"~" :: c :: cr => if isdigit c then
                                   let val (cs1, r) = screal(cr, realval c)
                                   in Real (~ r) :: sc cs1 end
                               else raise Scanerror
          | c :: cr         => if isblank c then sc cr
                               else if isdigit c then
                                   let val (cs1, r) = screal(cr, realval c)
                                   in Real r :: sc cs1 end
                               else raise Scanerror
    in sc (explode s) end
```

The scanner calls `screal` when it meets a minus sign '~' or a digit; in the former case it negates the result `r`.

5.5 Reading text files with SML

Scanners and parsers can be tested with small input strings typed directly in the SML session, but for larger input texts this is impractical. The SML function `readfile` shown below can read a text from a file. The function is called with the file name as argument and returns the contents of the file as a string.

```
fun readfile (filename: string): string =
    let open TextIO
        val infile = openIn filename
        val str    = inputAll infile
    in closeIn infile; str end
```

For example, assume that file `"source"` contains the following single input line:

```
0-1-1-
```

Then we use `readfile` to scan this line as follows:

```
val ts = scan (readfile "source");
> val ts = [Zero,Sub,One,Sub,One,Sub,Sub] : terminal list
```

5.6 Summary of scanners

Scanning is the first phase in the parsing of a text. It turns the string of characters into a list of terminal symbols, which is then read by a parser in the second phase.

The scanner fragments, the scanning functions, and the `readfile` function can be found in the files `gram/scan.sml`, `gram/scname.sml`, etc.

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 function must return an additional result. Some parsing functions 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 function `A` has had type

```
A: terminal list -> terminal list
```

but from now on its type will be

```
A: terminal list * invalue -> terminal list * outvalue
```

where `invalue` and `outvalue` are the types of the attributes. We call `A` an *attributed parser*. The `invalue` attribute is said to be *inherited* and `outvalue` is *synthesized*. One may decorate parse trees with attribute values. An inherited attribute is sent down the tree as an additional argument to a parsing function, and a synthesized attribute is sent up the tree as an additional result from a parsing function.

Some parsing functions do not take any inherited attributes, but most attributed parsing functions return a synthesized attribute. An attributed parsing function `A` either returns a pair of the remaining input and the synthesized attribute, or raises 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 function `parse` of a parser now takes as input a list of terminals and either returns a result or raises an exception. It is coded as:

```
fun parse ts =
    case (E ts) of
        ([], result) => result
      | _            => raise Parseerror
```

where `E` is the starting symbol of the grammar. In the examples and exercises below, function `parse: terminal list -> outvalue` 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 10 Recall the parser skeleton for arithmetic expressions in Example 8. We extend it so that every parsing function returns a synthesized attribute which is the value of the expression parsed by that function. To evaluate from left to right, we also extend function `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 `Sub` branch of function `Eopt`, its value `tv` is subtracted from `inval`, and the result is passed to `Eopt` in the recursive call.

```
fun E ts =
    let val (ts1, tv) = T ts
        val (ts2, ev) = Eopt (ts1, tv)
    in (ts2, ev) end
and Eopt (ts, inval) =
    case ts of
        Sub :: tr => let val ts1       = tr
                         val (ts2, tv) = T ts1
                         val (ts3, ev) = Eopt (ts2, inval-tv)
                     in (ts3, ev) end
      | _         => (ts, inval)
and T ts =
    case ts of
        Zero :: tr => (tr, 0)
      | One  :: tr => (tr, 1)
      | _          => raise Parseerror
```

Function `E` first calls `T`. Function `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 function `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. □

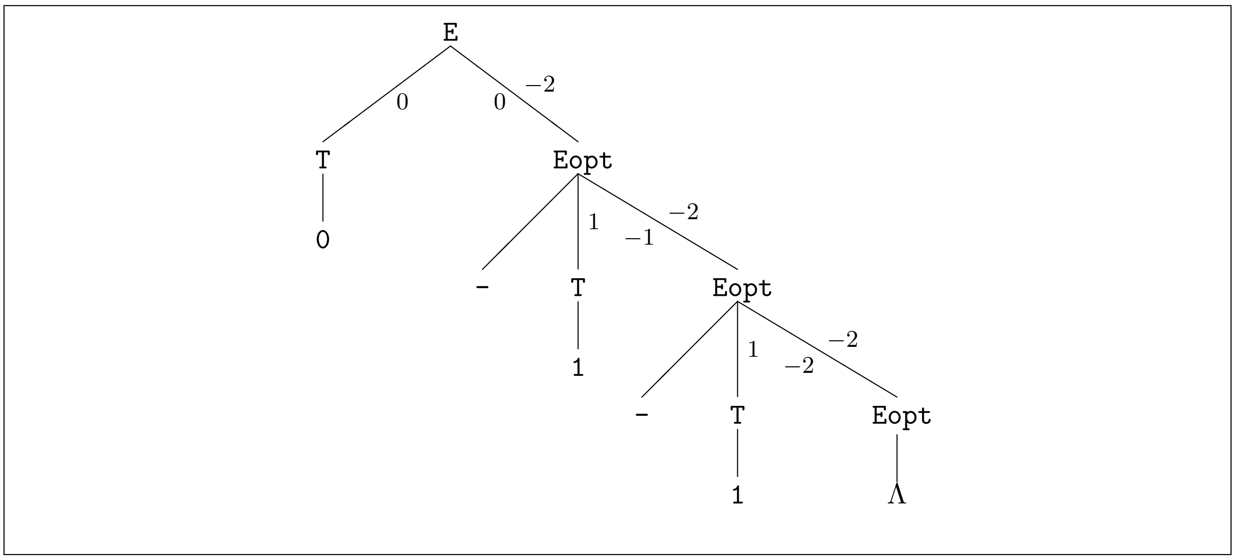


Figure 7: Parse tree with attributes for left-to-right evaluation

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.

The parsing functions have the following types

```
E     : terminal list       -> terminal list * int
Eopt  : terminal list * int -> terminal list * int
T     : terminal list       -> terminal list * int
parse : terminal list       -> int
```

A typical use of the attributed parser is

```
parse (scan "0-1-1");
> val it = ~2 : int
```

The parsing functions could be simplified a little. In particular the body of the `E` function could be simplified to `Eopt(T ts)`.

Above we defined left-to-right evaluation of arithmetic expressions, which is usual in mathematics and programming language. 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 11 The attributed parser in Example 10 can be changed to evaluate the expression from right to left as follows:

```
fun E ts =
    let val (ts1, tv) = T ts
        val (ts2, ev) = Eopt (ts1, tv)
    in (ts2, ev) end
and Eopt (ts, inval) =
    case ts of
        Sub :: tr => let val ts1       = tr
                         val (ts2, tv) = T ts1
                         val (ts3, ev) = Eopt (ts2, tv)
                     in (ts3, inval-ev) end
      | _         => (ts, inval)
and T ts =
    case ts of
        Zero :: tr => (tr, 0)
      | One  :: tr => (tr, 1)
      | _          => raise Parseerror
```

The only change is in the `Sub` branch of the `Eopt` function. 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.

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.

The following datatype is useful for representing the simple expressions from Example 2:

```
datatype expr = Zeroterm
              | Oneterm
              | Minus of expr * expr
```

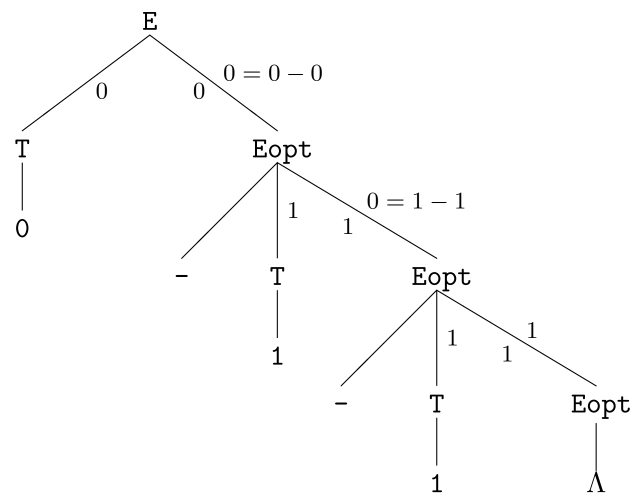


Figure 8: Parse tree with attributes for right-to-left evaluation

The declaration says: an expression is a zero, or a one, or an expression minus another expression.

For instance, the expression `0-1` can be represented as `Minus(Zeroterm, Oneterm)`. The expression `0-1-1` can be represented as either as `Minus(Minus(Zeroterm, Oneterm), Oneterm)` or as `Minus(Zeroterm, Minus(Oneterm, Oneterm))`. The first corresponds to a left-to-right reading, and the second corresponds to a right-to-left reading.

Let us make an attributed parser which computes 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 10.

Example 12 This parser builds abstract syntax trees for simple arithmetic expressions.

```
fun E ts =
    let val (ts1, tv) = T ts
        val (ts2, ev) = Eopt (ts1, tv)
    in (ts2, ev) end
and Eopt (ts, inval) =
    case ts of
        Sub :: tr => let val ts1       = tr
                         val (ts2, tv) = T ts1
                         val (ts3, ev) = Eopt (ts2, Minus(inval, tv))
                     in (ts3, ev) end
      | _         => (ts, inval)
and T ts =
    case ts of
        Zero :: tr => (tr, Zeroterm)
      | One  :: tr => (tr, Oneterm)
      | _          => raise Parseerror
```

Instead of returning an integer (0 or 1), function `T` now returns the representation (`Zeroterm` or `Oneterm`) of an expression. Instead of subtracting one number from another, returning a number, function `Eopt` now builds and returns a representation of an expression (in the `Sub` 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 10. At any point, `inval` is the representation of the expression parsed so far. □

The new attributed parsing functions have types

```
E     : terminal list        -> terminal list * expr
Eopt  : terminal list * expr -> terminal list * expr
T     : terminal list        -> terminal list * expr
parse : terminal list        -> expr
```

Typical uses of the new parser are

```
parse [One, Sub, Zero];
> val it = Minus (Oneterm,Zeroterm) : expr

parse (scan "0-1-1");
> val it = Minus (Minus (Zeroterm,Oneterm),Oneterm) : expr
```

6.3 Summary of parsers with attributes

To make parsing functions 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 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 real 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. Check this before you read on!

7.2 The parser constructed from the grammar

The terminal symbols of the grammar are the operators '+', '-', '*', '/', the parentheses '(' and ')', and the real numerals:

```
datatype terminal =
    Add | Sub | Mul | Div | Lpar | Rpar | Real of real
```

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

```
exception Parseerror
fun E ts =
    let val ts1 = T ts
        val ts2 = Eopt ts1
    in ts2 end
and Eopt ts =
    case ts of
        Add :: tr => let val ts1 = T tr
                         val ts2 = Eopt ts1
                     in ts2 end
      | Sub :: tr => let val ts1 = T tr
                         val ts2 = Eopt ts1
                     in ts2 end
      | _         => ts
and T ts =
    let val ts1 = F ts
        val ts2 = Topt ts1
    in ts2 end
and Topt ts =
    case ts of
        Mul :: tr => let val ts1 = F tr
                         val ts2 = Topt ts1
                     in ts2 end
      | Div :: tr => let val ts1 = F tr
                         val ts2 = Topt ts1
                     in ts2 end
      | _         => ts
and F ts =
    case ts of
        Real r :: tr => tr
      | Lpar :: tr   => let val ts1 = E tr
                       in case ts1 of
                           Rpar :: tr => tr
                         | _          => raise Parseerror
                       end
      | _            => raise Parseerror
```

7.3 A scanner for arithmetic expressions

An appropriate scanner is shown below. It ignores blanks and uses the scanner function `screal` for real numbers defined in Section 5.4. (File `gram/aritscan.sml` contains a copy of this scanner).

```
fun scan s =
    let fun sc cs =
        case cs of
            []              => []
          | #"+" :: cr      => Add :: sc cr
          | #"-" :: cr      => Sub :: sc cr
          | #"*" :: cr      => Mul :: sc cr
          | #"/" :: cr      => Div :: sc cr
          | #"(" :: cr      => Lpar :: sc cr
          | #")" :: cr      => Rpar :: sc cr
          | #"~" :: c :: cr => if isdigit c then
                                  let val (cs1, r) = screal(cr, realval c)
                                  in Real (~ r) :: sc cs1 end
                               else raise Scanerror
          | c :: cr         => if isblank c then sc cr
                               else if isdigit c then
                                   let val (cs1, r) = screal(cr, realval c)
                                   in Real r :: sc cs1 end
                               else raise Scanerror
    in sc (explode s) end
```

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 10, except that many subexpressions have been simplified by hand. (File `gram/ariteval.sml` contains a copy of this parser).

```
fun E ts = Eopt (T ts)
and Eopt (ts, inval) =
    case ts of
        Add :: tr => let val (ts1, tv) = T tr
                     in  Eopt (ts1, inval+tv) end
      | Sub :: tr => let val (ts1, tv) = T tr
                     in  Eopt (ts1, inval-tv) end
      | _         => (ts, inval)
and T ts = Topt (F ts)
and Topt (ts, inval) =
    case ts of
        Mul :: tr => let val (ts1, fv) = F tr
                     in  Topt (ts1, inval*fv) end
      | Div :: tr => let val (ts1, fv) = F tr
                     in  Topt (ts1, inval/fv) end
      | _         => (ts, inval)
and F ts =
    case ts of
        Real r :: tr => (tr, r)
      | Lpar :: tr   => let val (ts1, ev) = E tr
                        in case ts1 of
                               Rpar :: tr => (tr, ev)
                             | _          => raise Parseerror
                        end
      | _            => raise Parseerror
```

8 Scanner and parser facilities in the Basis Library

(This section added by Ken Friis Larsen, October 2002).

The SML Basis Library provides a framework for constructing scanners and parsers that draws characters and values from functional streams. The central type in the framework is the `reader` type in the `StringCvt` structure:

```
type ('elm, 'src) reader = 'src -> ('elm * 'src) option
```

That is, a `StringCvt.reader` is a function that given a source `src` (a functional stream) it returns `SOME(e, src')` where `e` is an element and `scr'` is the remainder of the stream; or it returns `NONE` if `src` is exhausted. For instance, a character source reader:

```
getc : (char, cs) reader
```

is used for obtaining characters from a functional character source `src` of type `cs`, one at a time. It should hold that:

$$\texttt{getc src} = \begin{cases} \texttt{SOME(c, src')} & \text{if the next character in } \texttt{src} \text{ is } \texttt{c}\text{, and } \texttt{src'} \text{ is the rest of } \texttt{src}; \\ \texttt{NONE} & \text{if } \texttt{src} \text{ contains no characters} \end{cases}$$

A character source scanner takes a character source reader `getc` as argument and uses it to scan a data value from the character source. The Basis Library already contains several character source scanners (or functions that generate character source scanners). Examples include: `Int.scan`, `Real.scan`, `Bool.scan`, and `Time.scan`.

Here we do not give a complete treatment of how to utilize the framework from the Basis Library. Instead we give examples of how to build a scanner and a parser that works with the arithmetic expressions described Section 7. (File `gram/basislib-scanpars.sml` contains a copy of the code presented in the following.)

8.1 Making a `StringCvt.reader` for scanning arithmetic expressions

The SML function `getTerminal` shown below corresponds to the scanner from Section 7.3. The type of `getTerminal` is:

```
(char, 'cs) StringCvt.reader -> (terminal, 'cs) StringCvt.reader
```

That is, `getTerminal` takes a character source reader as argument and returns a reader that reads from a character source and returns `terminals`.

```
fun getTerminal getc cs =
    case getc (StringCvt.skipWS getc cs) of
        NONE           => NONE
      | SOME(#"+", cs) => SOME(Add, cs)
      | SOME(#"-", cs) => SOME(Sub, cs)
      | SOME(#"*", cs) => SOME(Mul, cs)
      | SOME(#"/", cs) => SOME(Div, cs)
      | SOME(#"(", cs) => SOME(Lpar, cs)
      | SOME(#")", cs) => SOME(Rpar, cs)
      | _              => (case Real.scan getc cs of
                               SOME(r, cs) => SOME(Real r, cs)
                             | NONE => raise Scanerror)
```

8.2 A `StringCvt.reader` for parsing and evaluating arithmetic expressions

The SML function `evalExpr` corresponds to the parser in Section 7.4. The type of `evalExpr` is:

```
(char, 'cs) StringCvt.reader -> (real, 'cs) StringCvt.reader
```

That is, `evalExpr` takes a character source reader `getc` as argument, uses `getTerminal` to get terminals one by one, and evaluate the arithmetic expressions while parsing them.

```
fun evalExpr getc src =
    let val gett = getTerminal getc

        fun E ts = Eopt (T ts)
        and Eopt (inval, ts) =
            case gett ts of
                SOME(Add, tr) => let val (tv, ts1) = T tr
                                 in  Eopt (inval+tv, ts1) end
              | SOME(Sub, tr) => let val (tv, ts1) = T tr
                                 in  Eopt (inval-tv, ts1) end
              | _             => (inval, ts)
        and T ts = Topt (F ts)
        and Topt (inval, ts) =
            case gett ts of
                SOME(Mul, tr) => let val (fv, ts1) = F tr
                                 in  Topt (inval*fv, ts1) end
              | SOME(Div, tr) => let val (fv, ts1) = F tr
                                 in  Topt (inval/fv, ts1) end
              | _             => (inval, ts)
        and F ts =
            case gett ts of
                SOME(Real r, tr) => (r, tr)
              | SOME(Lpar, tr)   => let val (ev, ts1) = E tr
                                    in case gett ts1 of
                                           SOME(Rpar, tr) => (ev, tr)
                                         | _      => raise Parseerror
                                    end
              | _                => raise Parseerror

        val (res, src) = E src

    in  case getc (StringCvt.skipWS getc src) of
            NONE => SOME(res, src)
          | _    => raise Parseerror
    end
```

The last `case`-expression is to ensure that we don't accept input that have trailing garbage (except for spaces). That is, the character source must be exhausted.

8.3 Using a `StringCvt.reader`

Because `evalExpr` is a `StringCvt.reader` we can use it with `StringCvt.scanString` to construct a function `evalString` that can parse and evaluate arithmetic expressions given as strings:

```
fun evalString s = valOf(StringCvt.scanString evalExpr s)
```

The function `evalString` can be used as

```
evalString "3.0 * (5.0 + 9.0)";
> val it = 42.0 : real
```

We can also use `evalExpr` with `TextIO.scanStream` to construct a function `evalFile` that reads, parses, and evaluates an arithmetic expression from a file.

```
fun evalFile filename =
    let val inStrm   = TextIO.openIn filename
        fun close () = TextIO.closeIn inStrm
    in  (valOf(TextIO.scanStream evalExpr inStrm)
         before
         close())
        handle e => (close(); raise e)
    end
```

9 Some background

9.1 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.

9.2 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* $\texttt{e}_1 \ \texttt{...} \ \texttt{e}_m$ 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.

9.3 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. The Unix utility '`Yacc`' is an automatic parser generator for $LALR(1)$ grammars. The $LR(1)$ grammars are sufficiently powerful for most computing problems, but as exemplified by Exercise 5 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	

9.4 Further reading

A description (in Danish) of practical recursive descent parsing using Turbo Pascal is given by Kristensen [2], who provided Example 1 and other inspiration.

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 [3], and in Wirth [4, Chapter 5].

10 Exercises

Exercise 1 Write down a grammar for SML lists of unsigned integers. Show the derivations of `[]` and `[7, 9, 13]`. □

Exercise 2 Construct a grammar for the container data in Example 1. □

Exercise 3 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 4 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 5 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 6 Consider the grammar in Exercise 3. Left factorize it. Construct a parser skeleton for the left factorized grammar, using `datatype terminal = Add | Sub | Zero | One`. □

Exercise 7 The grammar

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

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

Exercise 8 Write a grammar and construct a parser for parenthesized expressions such as `0`, `0+(1)`, `1-(1+1)`, `(0-1)-1`, etc. □

Exercise 9 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 datatype `terminal` for 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 constructed value `Nat 123` of type `terminal`. (4) Construct a parser skeleton for the transformed grammar and test it. □

Exercise 10 Show that the requirements in Figure 6 imply that for every grammar rule, and distinct alternatives $\texttt{f}_i$ and $\texttt{f}_j$, it holds that $\mathit{Select}(\texttt{f}_i) \cap \mathit{Select}(\texttt{f}_j) = \{\}$. □

Exercise 11 The input language for the scanner in Example 9 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 12 Extend the parser constructed in Exercise 6 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 13 Extend the parser constructed in Exercise 6 to build an abstract syntax tree for the parsed expression, using the following datatype:

```
datatype expr = Zeroterm
              | Oneterm
              | Plus of expr * expr
              | Minus of expr * expr
```

What are the types of the attributed parsing functions? □

Exercise 14 Extend the scanner from Section 7.3 to recognize SML reals with exponents such as '`6.6256E~34`' or '`3E8`'. □

Exercise 15 What changes are necessary to make the parser in Example 12 build representations from right to left? □

Exercise 16 Check that the grammar at the end of Section 7.1 satisfies the grammar requirements. □

Exercise 17 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 18 The following datatype represents the expressions from Section 7:

```
datatype expr =
    CstR of real
  | Plus   of expr * expr
  | Minus  of expr * expr
  | Times  of expr * expr
  | Divide of expr * expr
```

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

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

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

Index

$\Longrightarrow$ (derivation), 5
Λ (empty sequence), 5

abstract syntax tree, 27
alternative, 5, 37
arithmetic expressions, 30
associate to the left, 25
attribute, 25
attributed parser, 25

Backus-Naur Form, 37
blank, 20
BNF, 37
bottom-up parsing, 38

context-free grammar, 38
context-sensitive grammar, 38

derivation, 5
derivation tree, 6

EBNF, 37
element, 37
`evalExpr` (function), 35
`evalFile` (function), 36
`evalString` (function), 36
Extended Backus-Naur Form, 37

factor, 30
factorize, 10
First(`f`) (first-set), 13
first-set, 12, 13
Follow(`A`) (follow-set), 13
follow-set, 12, 13

`getTerminal` (function), 34
grammar, 4, 5
grammar notation, 5
grammar requirements, 13
grammar rule, 5
grouping, 37

identifier, 21
inherited attribute, 25
`isblank` (function), 21

keywords, 22

LALR grammar, 38
language
    generated by grammar, 6
left associative, 25
left factorization, 10
left recursive, 11
lexical analysis, 20
LL grammar, 38
LR grammar, 38

name
    scanning of, 21
nonterminal symbol, 5
numeral, 22

option, 37

`parse` (function), 16, 18, 25
parse tree, 9
parser, 4
parser construction, 16
parser skeleton, 16
parser with attributes, 25
parsing, 7
    bottom-up, 38
    recursive descent, 38
    top-down, 7, 38
parsing theory, 7
precedence, 30

`readfile` (function), 24
recursive descent parsing, 38
regular grammar, 38
repetition, 37
requirements on grammar, 13
rule, 5

`scan` (function), 20, 22, 23, 32
scanner, 4, 20
scanning, 20
`scfrac` (function), 23
`scname` (function), 21
`screal` (function), 23
Select(`f`) (selection set), 12
selection set, 12
self left recursive, 11
sequence, 5

starting symbol, 5
symbol, 5
syntax, 25
syntax analysis, 7
synthesized attribute, 25

term, 30
terminal symbol, 5
top-down parsing, 7, 38
tree
  derivation, 6
  parse, 9

unrestricted grammar, 38

Yacc, 38