Grammars and parsing[1] with Haskell Using Parser Combinators

Peter Sestoft[2] and Ken Friis Larsen[3]
sestoft@itu.dk ken@friislarsen.net

DRAFT VERSION 2
2013-09-11

[1] Based on earlier versions for Standard ML, for Java, for C#, and for Python.

[2] IT University of Copenhagen, Denmark.

[3] Department of Computer Science, University of Copenhagen, 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 one or more Haskell data types, 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 layout.

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 (a list, for instance) of the data in the input file. Here we shall not be concerned with the meaning[4] 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 Haskell using parser combinators. The methods shown here are essentially independent of Haskell, and can be used with suitable modifications in any language that has recursive procedures (Java, Ada, C, ML, Python, 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.

[4] 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 ϵ 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* $G = (T, N, R, \texttt{S})$ has a set T of terminals, a set N of nonterminals, a set R of rules, and a starting symbol $\texttt{S} \in N$.

A *rule* has form `A = f`$_1$ `| ... | f`$_n$, where $\texttt{A} \in N$ 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 ϵ.

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 $\texttt{T} \Longrightarrow \texttt{"0"}$ and $\texttt{T} \Longrightarrow \texttt{"1"}$.

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 $\texttt{E} \Longrightarrow \texttt{"0"}$.

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

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

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 is 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"`.

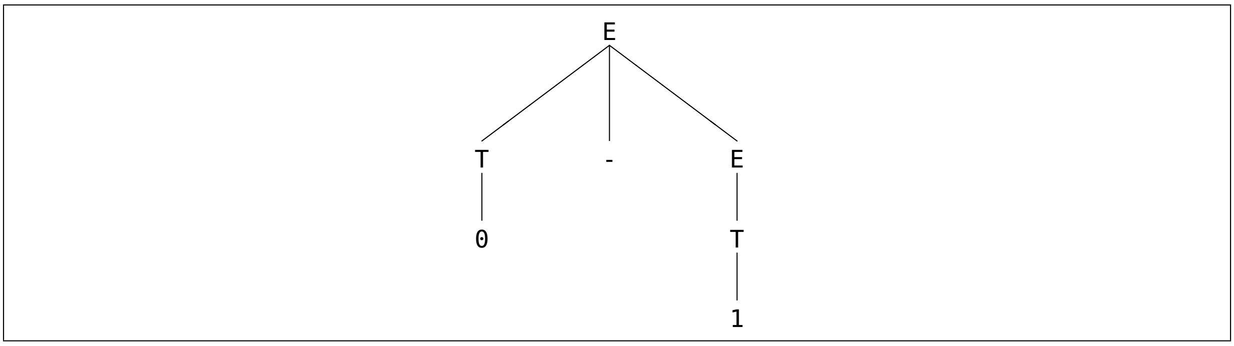


Figure 2: A derivation tree

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 ϵ. 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`, ...}, 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
            |  ϵ .
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 Haskell parser functions working according to this principle.

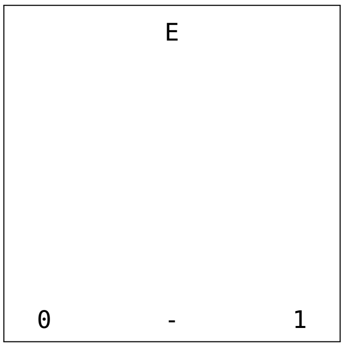
3.1 Parsing: reconstruction of a derivation tree

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:

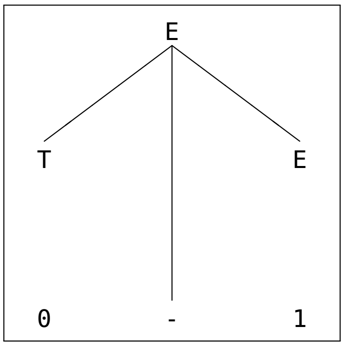
```
E   ::= T "-" E | T .
T   ::= "0" | "1" .
```

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:



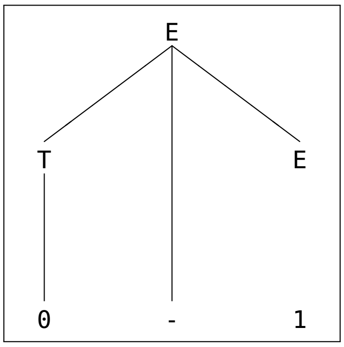
(a)

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` ⟹ `T "-" E` and `E` ⟹ `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):



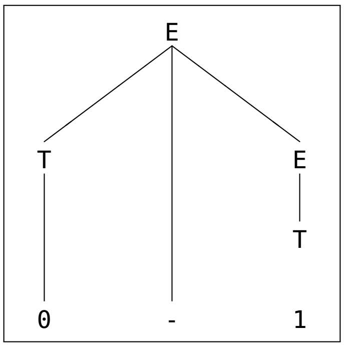
(b)

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



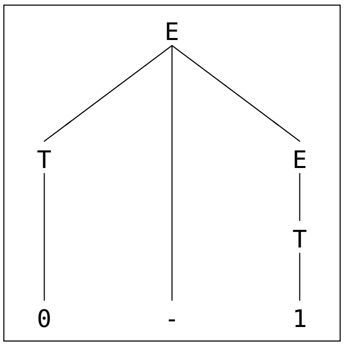
(c)

Next we must see how the remaining input symbol `"1"` can be derived from `E`. The `E` ⟹ `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` ⟹ `"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`. Second, 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, ϵ.

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 ϵ 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"
         ϵ                   ϵ
```

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 | ϵ)`', 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 | ϵ`, 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 | ϵ .
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` $\Longrightarrow$ ϵ and `Eopt` $\Longrightarrow$ `"-" T Eopt`.

Since ϵ 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, ϵ.

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
         ϵ                       Eopt
         ϵ                        ϵ
```

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.

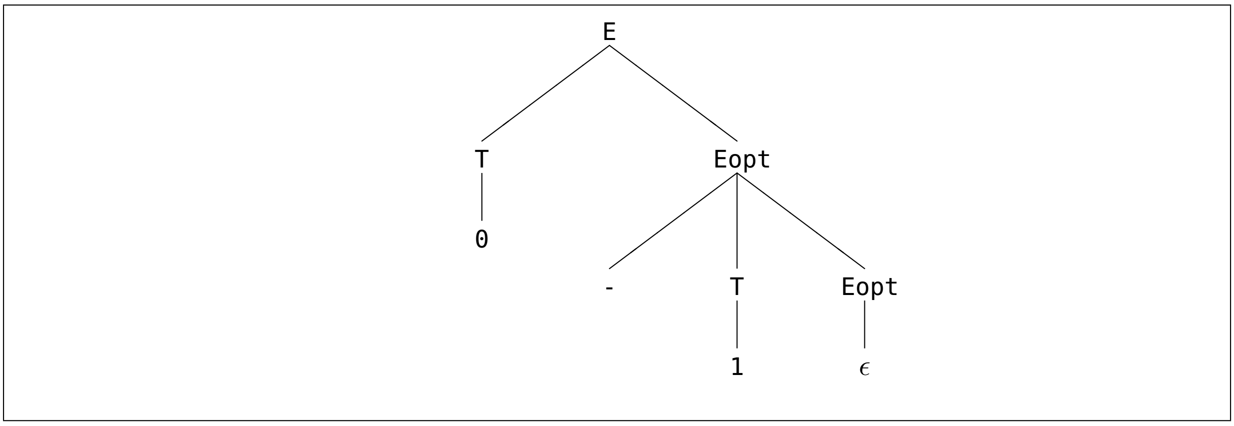


Figure 3: Derivation tree for the left factorized grammar

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 $\texttt{E} \Longrightarrow \texttt{E ...}$ 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:

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

The $\texttt{g}_i$ and $\texttt{f}_j$ stand for sequences of grammar symbols (possibly ϵ). 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 $\texttt{f}_j$, and continue with zero or more $\texttt{g}_i$'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 | ϵ .
```

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 ϵ 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 ϵ, and $\mathit{First}(\mathtt{f}_i) \cup \mathit{Follow}(\mathtt{A})$ if $\mathtt{f}_i$ can derive ϵ:

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

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 ϵ via $\mathtt{f}_i$.

How can we compute $\mathit{First}(\mathtt{f})$? If `f` is ϵ, we have $\mathit{First}(\epsilon) = \{\}$ 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 ϵ, 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 ϵ, $\mathit{First}(\mathtt{f})$ includes $\mathit{First}(\mathtt{e}_2\ \ldots\ \mathtt{e}_m)$ also.

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

How can we compute the follow-set $\mathit{Follow}(\mathtt{A})$ of a nonterminal `A`? Assume that `A` appears in the rule $\mathtt{B} ::= \ldots \mid \ldots\ \mathtt{A}\ \mathtt{f} \mid \ldots$ for nonterminal `B`, where `f` is a string of grammar symbols (possibly

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

Figure 4: Computation of first-sets

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

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

Figure 5: Computation of follow-sets

ϵ). Then $\mathit{Follow}(\texttt{A})$ must include everything that `f` can begin with, and, if `f` can derive ϵ, then also everything that can follow `B`. This is expressed by Figure 5.

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

$$
\begin{array}{lll}
\text{Form 0:} & \texttt{A ::= f}_1 & \\
\text{Form 1:} & \texttt{A ::= f}_1 \mid \ldots \mid \texttt{f}_n & n \geq 2 \\
\text{Form 2:} & \texttt{A ::= f}_1 \mid \ldots \mid \texttt{f}_n \mid \epsilon & n \geq 1
\end{array}
$$

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 ϵ.
- 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

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

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 ϵ.

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 | ϵ .
T    ::= "0" | "1" .
```

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

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

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 (ϵ). □

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

We now show a systematic way to write a *parser skeleton* (in Haskell) 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 Parser Combinator Libraries

In Haskell it is convenient to use a *parser combinator library* for writing our parser. There are several high-quality libraries to chose from. In Section 8 we shall present some of them. For now, what we need to assume is that the library implements the following interface:

```
newtype Parser a = ...

instance Monad Parser where ...
instance MonadPlus Parser where ...

runParser :: Parser a -> String -> Either Error a

reject :: Parser a
char   :: Char   -> Parser Char
string :: String -> Parser String
satisfy:: (Char -> Bool) -> Parser Char

eof :: Parser ()

(<|>)  :: Parser a -> Parser a -> Parser a

many   :: Parser a -> Parser [a]
many1  :: Parser a -> Parser [a]

option :: Parser a -> Parser (Maybe a)
```

That is, there should be a type constructor `Parser a`, which is the type of parsers returning an element of type `a`. The `Parser` type constructor must be an element of the type classes `Monad` and `MonadPlus`. Meaning there should the instances of the classes:

```
instance Monad Parser where
    (>>=)  :: Parser a -> (a -> Parser b) -> Parser b
    return :: a -> Parser a
instance MonadPlus Parser where
    mzero  :: Parser a
    mplus  :: Parser a -> Parser a -> Parser a
```

Furthermore, we shall use the infix operator `<|>` as an alternative name for `mplus` and the function `reject` as alternative name for `mzero` (both for giving a better intuition for what they are used for when constructing parsers). The combinator `eof` is for checking that we have reached the end of the input.

4.2 Constructing parsing functions in Haskell

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 `Aname` is called `aname`, that is the same name but un-capitalised. The function `aname` tries to find a string derivable from the nonterminal `Aname` at the beginning of the current input. If it succeeds, then it just returns, possibly after having consumed parts of input. If it fails, then it will use the `reject` combinator, which is polymorphic and thus always can be given the right type.

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

```
$a_1$ :: Parser ()
$a_1$ = ...
$a_2$ :: Parser ()
$a_2$ = ...
...
$a_k$ :: Parser ()
$a_k$ = ...

parseString input = runParser (s » eof) input
```

where {$\mathtt{A}_1$, …, $\mathtt{A}_k$} = N is the set of nonterminals and `S` is the starting symbol. The main function is `parseString`; it checks that no input remains after parsing. If parsing succeeds and no input remains, it just returns; otherwise it raises an exception.

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

```
a = parse code for $f_1$
```

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

```
a = parse code for alternative $f_1$
    <|> ...
    <|> parse code for alternative $f_n$
```

Rule of form 2 The parsing function for a rule of form `A ::=` $\mathtt{f}_1$ `| ... |` $\mathtt{f}_n$`|` ϵ is

```
a = parse code for alternative $f_1$
    <|> ...
    <|> parse code for alternative $f_n$
    <|> return ()
```

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

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

```
do  $\mathcal{P}(e_1)$
    $\mathcal{P}(e_2)$
   ...
    $\mathcal{P}(e_m)$
   return ()
```

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 function.

Terminal The parse code $\mathcal{P}$("c") for a terminal "c" is a call to the combinator string "c".

Note that in the parser construction for grammar rules of form 1 and 2, the grammar requirements ensure that the first-sets are disjoint, so the 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 f$_i$ alternative is never wrongly chosen instead of the last alternative (for ϵ).

Example 9 Applying this construction function to the left factorized grammar from Example 4 gives the parser below.

```
e = do t
       eopt
       return()
eopt = (do string "-"
           t
           eopt
           return ())
     <|> return ()
t = (do string "0"
        return ())
  <|> (do string "1"
          return())
parseString input = runParser (e » eof) input
```

To demonstrate the construction method we have followed it mindlessly in this example. Parts of the program may be improved, for example most of the do-expression can be simplified a lot. However, it is advisable to postpone such improvements until you have acquired more practice with parser construction.

□

4.3 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, follows the grammar. In Section 6 we show how to make the parser return more information, such as an internal representation of the input.

5 Lexical analysis

In the parser skeleton in the previous section, we just used the combinator `string` to handle lexical analysis. This is usually not adequate. In particular, we do not handle white-space (also called blanks). It is inconvenient and not elegant to deal with white-space all over our parser. Therefore parsing of text files is usually divided into two phases or layers.

In the *first* phase, we make a small set of combinators for dealing with the bare character stream This is called *scanning* or *lexical analysis* and is explained in this section.

In the *second* phase, the list 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 discard 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. Instead we define a small set of combinators to deal the lexical analysis:

```
space    :: Parser Char
space     = satisfy isSpace
spaces   :: Parser String
spaces    = many space
spaces1  :: Parser String
spaces1   = many1 space

token    :: Parser a -> Parser a
token p   = spaces >> p

symbol    :: String -> Parser String
symbol    = token . string
schar     :: Char -> Parser Char
schar     = token . char
```

First, we define the functions `space`, `spaces`, and `space1` that recognise a single blank, a sequence of zero or more blanks, or a sequence of at least one blank. Second, we define the combinator `token` that takes an other parser, `p`, as argument, and removes spaces before `p` is used. Finally, we define the functions `symbol` and `schar` for the often used case where our tokens can be described by a single string or character.

Example 10 The parser below is the same as the parser in Example 9, except that we have used the function `symbol` instead of the function `string` (we could also have used `schar` in this example), we are skipping spaces (using the function `token`) before `eof`. Thus, it ignores all blanks, and considers all characters other than ‘`-`’, ‘`0`’ and ‘`1`’ as errors.

```
e = do t
       eopt
       return()
eopt = (do symbol "-"
           t
           eopt
           return ())
     <|> return ()
t = (do symbol "0"
        return ())
  <|> (do symbol "1"
          return())
parseString input = runParser (e » token eof) input
```

□

5.1 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 | ϵ .
Letter   ::= "A" | ... | "Z" | "a" | ... | "z" .
Digit    ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

A scanning function `name` for names using the *satisfy* function, and character classification predicates from the Haskell library, is show in the following:

```
name :: Parser String
name = token (do c  <- letter
                 cs <- letdigs
                 return (c:cs))
     where letter = satisfy isLetter
           letdigs = many (letter <|> num)
           num = satisfy isDigit
```

Here we have used the `token` combinator to skip spaces before names.

5.2 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, Haskell have the keywords ‘`class`’, ‘`data`’, ‘`type`’, and so on.

Keywords are represented by other terminal symbols than names. For instance, if ‘`class`’, ‘`data`’, and ‘`type`’ are keywords, then we often just want to skip them using `symbol` or sometimes we want to return a constant such as `Class`, `Data`, or `Type` (assuming that we have declared a type with such constants).

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

```
data Keyword = Keyword String

reserved :: [String]
reserved = ["class", "data", "type"]

nameOrKeyword :: Parser (Either Keyword String)
nameOrKeyword = do n <- name
                   if n ‘elem‘ reserved then return (Left(Keyword n))
                   else return (Right n)
```

5.3 Scanning floating-point numerals

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

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

Like for names, we can define a parser for paring floating point numbers realNumber, shown in the following:

```
realNumber :: Parser Double
realNumber = token (do pre <- digits
                       char '.'
                       post <- digits
                       return $ read $ pre ++ "." ++ post)
           where digits = many1(satisfy isDigit)
```

The realNumber parser returns the matched string converted to a floating-point number using the library function read, and uses token to skip preceding whitespace, like name.

5.4 Summary of lexical analysis

Lexical analysis is the first phase in the parsing of a text. It mainly deals with the removal of whitespace.

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 `apars` has taken some input and have returned nothing (except perhaps a error). From now on, a parsing function can return a result and take more arguments beside the implicit input. We call (the new kind of) `apars` an *attributed parser*. The extra arguments are called an *inherited* attributes and the return value is called a *synthesized* attribute. One may decorate parse trees with attribute values. Inherited attributes are sent down the tree as an additional arguments 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 `apars` either returns the synthesized attribute, or return an error.

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 `parseString` of a parser now either returns a result or raises an exception:

```
parseString input = runParser (do
  res <- s
  token eof
  return res) input
```

where `s` is the starting symbol of the grammar. In the following examples and exercises, function `parseString` 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 arithmetic expressions in Example 9. 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 `-` branch of function `eopt`, its value `tv` is subtracted from `inval`, and the result is passed to `eopt` in the recursive call.

```
e = do tv <- t
       v <- eopt tv
       return v
eopt inval = (do symbol "-"
                 tv <- t
                 v <- eopt (inval - tv)
                 return v)
     <|> return inval
t = (do symbol "0"
        return 0)
  <|> (do symbol "1"
          return 1)
parseString input = ...
```

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` in which case it calls `t` again, subtracts the new `T`-value `tv` from `inval`, and passes the difference to `eopt` in the recursive call, *or* it parses ϵ, 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. □

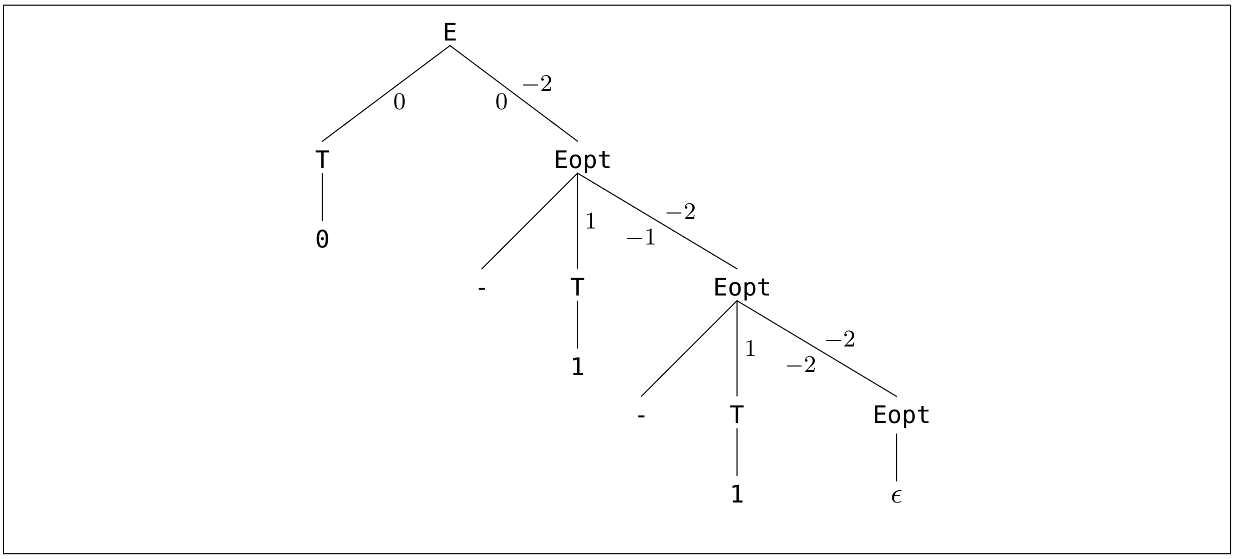


Figure 7: Parse tree with attributes for left-to-right evaluation

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.

Some of the previous parsing functions could be simplified. For instance, the function `e` could be simplified to:

```
e = do tv <- t
       eopt tv
```

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.

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 from Example 11.

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

```
e = do tv <- t
       v <- eopt tv
       return v
eopt inval = (do symbol "-"
                 tv <- t
                 ev <- eopt tv
                 return (inval - ev))
     <|> return inval
t = (do symbol "0"
        return 0)
  <|> (do symbol "1"
          return 1)
parseString input = ...
```

The only change is in the - 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. □

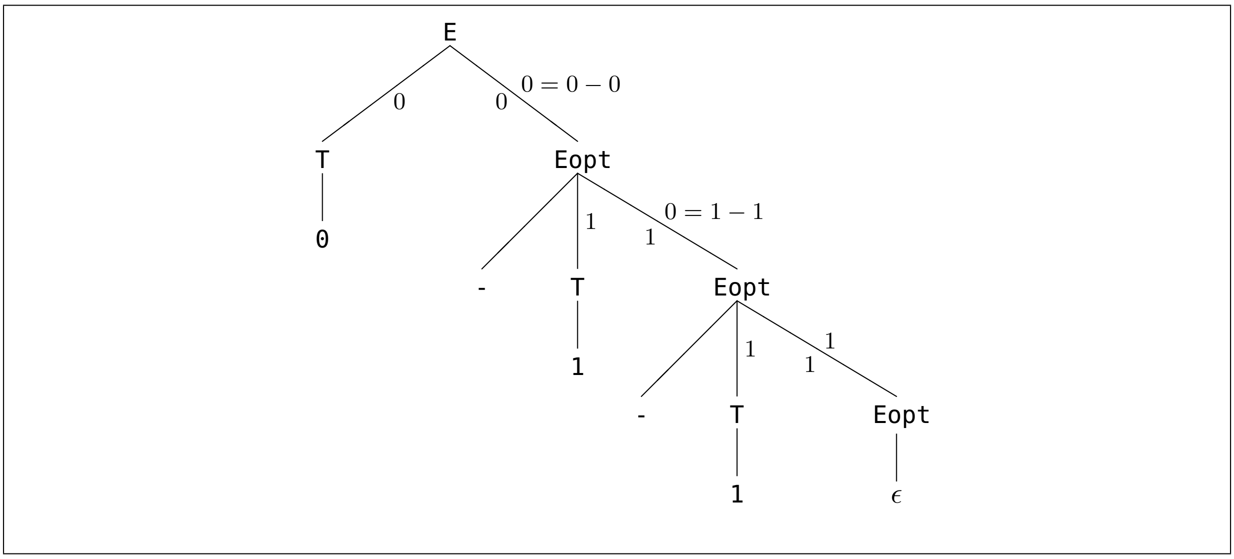


Figure 8: Parse tree with attributes for right-to-left evaluation

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 synthesised 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.

A flexible and general way to represent abstract syntax trees in Haskell is to use algebraic data types. For instance, to represent simple expressions as defined in Example 2:

```
data Expr = Zeroterm
          | Oneterm
          | Minus Expr Expr
          deriving (Eq, Show)
```

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 11.

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

```
e = do tv <- t
       ev <- eopt tv
       return ev
eopt inval = (do symbol "-"
                 tv <- t
                 ev <- eopt (Minus inval tv)
                 return ev)
     <|> return inval
t = (do symbol "0"
        return Zeroterm)
  <|> (do symbol "1"
          return Oneterm)
parseString input = ...
```

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 `-` 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 attributed parsing functions have types

```
e      :: Parser Expr
eopt   :: Expr -> Parser Expr
t      :: Parser Expr
parseString :: String -> Either Error Expr
```

□

A typical application of the attributed parser in Example 13 would look like this:

```
> parseString "0-1-1"
Right(Minus(Minus Zeroterm Oneterm) Oneterm)
```

Calling function `parseString` will scan and parse the string and build a representation of its contents, as a value of type `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 seperated by multiplications and divisions. 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 | ϵ .
T    ::= F Topt .
Topt ::= "*" F Topt | "/" F Topt | ϵ .
F    ::= Real | "(" E ")" .
```

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

7.2 The parser constructed from the grammar

Application of the construction method from Section 4.2 to the above grammar gives the parser skeleton shown in the following.

```
e = do t
       eopt
       return()
eopt = (do symbol "-"
           t
           eopt
           return ())
   <|> (do symbol "+"
           t
           eopt
           return ())
   <|> return ()
t = do f
       topt
       return()
topt = (do symbol "*"
           f
           topt
           return ())
   <|> (do symbol "/"
           f
           topt
           return ())
   <|> return ()
f = (do _ <- realNumber
        return ())
  <|> (do symbol "("
          e
          symbol ")"
          return())
parseString = e >> token eof
```

7.3 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 many subexpressions have been simplified by hand.

```
e, t, f :: Parser Double
e = do tv <- t
       eopt tv
eopt :: Double -> Parser Double
eopt inval =
       (do symbol "-"
           tv <- t
           eopt(inval - tv))
   <|> (do symbol "+"
           tv <- t
           eopt(inval + tv))
   <|> return inval
t = do fv <- f
       topt fv
topt :: Double -> Parser Double
topt inval =
       (do symbol "*"
           fv <- f
           topt(inval * fv))
   <|> (do symbol "/"
           fv <- f
           topt(inval / fv))
   <|> return inval
f = realNumber
  <|> (do symbol "("
          ev <- e
          symbol ")"
          return ev)
```

We use the parser `realNumber` from Section 5.3 to parse floating-point numbers.

8 Haskell Parser Combinator Libraries

Use either SimpleParse, Parsec, or ReadP.
Draft note: To be done

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 ϵ 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, Λ is used instead of our ϵ (in earlier versions of this document, for instance).

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* $\mathtt{e}_1$ `...` $\mathtt{e}_m$ 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 |` ϵ.
- a repetition `{ f }` is replaced by a new nonterminal `Repf` with rule `Repf = f Repf |` ϵ.
- 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 4 there are grammars for which there is no equivalent $LR(1)$ grammar (and consequently no $LALR(1)$-grammar or $LL(1)$-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 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 lists of (unsigned) integers. For instance, the empty list 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" | ϵ .
```

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 parser skeleton 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 `0`, `1`, `2`, ... (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 $Select(\mathtt{f}_i) \cap Select(\mathtt{f}_j) = \{\}$. □

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

```
input ::= "-" input | "0" input | "1" input | blank input | ϵ .
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:

```
data Expr = Zero
          | One
          | Minus Expr Expr
          | Plus
          deriving (Eq, Show)
```

What are the types of the attributed parsing functions? □

Exercise 13 Extend the lexical parser from Section 5.3 to recognize Haskell 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 data type may be used to represent the arithmetic expressions from Section 7:

```
data Expr = CstD   Double
          | Plus   Expr Expr
          | Minus  Expr Expr
          | Times  Expr Expr
          | Divide Expr Expr
          deriving (Show, Eq)
```

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

Exercise 18 S-expressions (for *symbolic expression*) are the basic building blocks for Lisp programs. An S-expression is either a literal value (numeric or symbolic constant), or a sequence of s-expressions enclosed in parentheses, a list-expression. That is, S-expressions can be described by the following grammar:

```
SExp  ::= Number
        | Symbol
        | "(" SExps ")" .
SExps ::= SExp SExps
        | ϵ .
```

Where `Number` is an integer, and `Symbol` is non-empty sequence of non-blanks.
The following data type can be used for representing S-expressions:

```
data SExp = IntVal Int
          | SymbolVal String
          | ListExp [SExp]
          deriving (Show, Eq, Ord)
```

Write an attributed parser that builds abstract syntax trees for S-expressions.

□

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