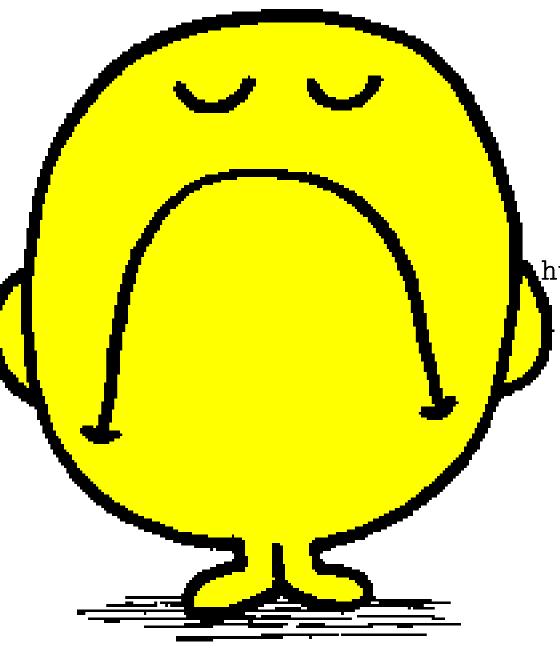
Frown

An LALR(k) Parser Generator for Haskell

version 0.6 (andromeda release)



RALF HINZE
Institut für Informatik III
Universität Bonn
Römerstraße 164
53117 Bonn
Germany
`ralf@cs.uni-bonn.de`
`http://www.cs.uni-bonn.de/~ralf/`

1st November 2005

Abstract

Frown is an LALR(k) parser generator for Haskell 98 written in Haskell 98.

Its salient features are:

- The generated parsers are time and space efficient. On the downside, the parsers are quite large.
- Frown generates four different types of parsers. As a common characteristic, the parsers are *genuinely functional* (ie 'table-free'); the states of the underlying LR automaton are encoded as mutually recursive functions. Three output formats use a typed stack representation, one format due to Ross Paterson (`code=stackless`) works even without a stack.
- Encoding states as functions means that each state can be treated individually as opposed to a table driven-approach, which necessitates a uniform treatment of states. For instance, look-ahead is only used when necessary to resolve conflicts.
- Frown comes with debugging and tracing facilities; the standard output format due to Doaitse Swierstra (`code=standard`) may be useful for teaching LR parsing.
- Common grammatical patterns such as repetition of symbols can be captured using *rule schemata*. There are several predefined rule schemata.
- Terminal symbols are arbitrary variable-free Haskell patterns or guards. Both terminal and nonterminal symbols may have an arbitrary number of synthesized attributes.
- Frown comes with extensive documentation; several example grammars are included.

Furthermore, Frown supports the use of monadic lexers, monadic semantic actions, precedences and associativity, the generation of backtracking parsers, multiple start symbols, error reporting and a weak form of error correction.

Contents

Chapter 1

Introduction

Frown is an LALR(k) parser generator for Haskell 98 written in Haskell 98.

The work on Frown started as an experiment in generating genuinely functional LR parsers. The first version was written within three days—yes, Haskell is a wonderful language for rapid prototyping. Since then Frown has gone through several cycles of reorganization and rewriting. It also grew considerably: dozens of features were added, examples were conceived and tested, and this manual was written. In the end, Frown has become a useable tool. I hope you will find it useful, too.

1.1 Obtaining and installing Frown

Obtaining Frown The parser generator is available from

`http://www.informatik.uni-bonn.de/~ralf/frown`.

The bundle includes the sources and the complete documentation (dvi, ps, PDF, and HTML).

Requirements You should be able to build Frown with every Haskell 98-compliant compiler. You have to use a not too ancient compiler as there have been some changes to the Haskell language in Sep. 2001 (GHC 5.02 and later versions will do).

The Haskell interpreter Hugs 98 is needed for running the testsuite.

Various tools are required to generate the documentation from scratch: `lhs2TeX`, LaTeX, functional METAPOST, HEVEA and HACHA. Note, however, that the bundle already includes the complete documentation.

Installation Unzip and untar the bundle. This creates a directory called `Frown`. Enter this directory.

```
ralf> tar xzf frown.tar.gz
ralf> cd Frown
```

The documentation resides in the directory `Manual`; example grammars can be found in `Examples` and `Manual/Examples` (the files ending in `.g` and `.lg`).

You can install Frown using either traditional makefiles or Cabal.

Using makefiles Optionally, edit the `Makefile` to specify destinations for the binary and the documentation (this information is only used by `make install`). Now, you can trigger

```
ralf/Frown> make
```

which compiles Frown generating an executable called `frown` (to use Frown you only need this executable). Optionally, continue with

```
ralf/Frown> make install
```

to install the executable and the documentation.

For reference, here is a list of possible targets:

`make`
: Compiles Frown generating an executable called `frown` (to use Frown you only need this executable).

`make install`
: Compiles Frown and installs the executable and the documentation.

`make test`
: Runs the testsuite.[1]

`make man`
: Generates the documentation in various formats (dvi, ps, PDF, and HTML).

`make clean`
: Removes some temporary files.

`make distclean`
: Removes all files except the ones that are included in the distribution.

Using Cabal Alternatively, you can build Frown using Cabal (version 1.1.3 or later), Haskell's Common Architecture for Building Applications and Libraries.

For a global install, type:

```
ralf/Frown> runhaskell Setup.hs configure --ghc
ralf/Frown> runhaskell Setup.hs build
ralf/Frown> runhaskell Setup.hs install
```

If you want to install Frown locally, use (you may wish to replace `$HOME` by a directory of your choice):

```
ralf/Frown> runhaskell Setup.hs configure --ghc --prefix=$HOME
ralf/Frown> runhaskell Setup.hs build
ralf/Frown> runhaskell Setup.hs install --user
```

Usage The call

```
ralf/Frown> frown -h
```

displays the various options. For more information consult this manual.

1.2 Reporting bugs

Bug reports should be send to Ralf Hinze (`ralf@cs.uni-bonn.de`). The report should include all information necessary to reproduce the bug: the compiler used to compile Frown, the grammar source file (and possibly auxiliary Haskell source files), and the command-line invocation of Frown.

Suggestions for improvements or request for features should also be sent to the above address.

[1] There are some known problems. The format `code=stackless` behaves differently for `Loop.g` (the generated parser is less strict than the standard one). Also, `Empty.g` does not work yet. Finally, error reports may differ for different formats and for optimized and unoptimized versions (as some parsers perform additional reductions before an error is reported).

1.3 License

Frown is distributed under the GNU general public licence (version 2).

1.4 Credits

Frown wouldn't have seen the light of day without the work of Ross Paterson and Doaitse Swierstra. Ross invoked my interest in LR parsing and he devised the `--code=stackless` and `--code=gvstack` output formats. Doaitse invented the `--code=standard` format, which was historically the first format Frown supported.

A big thank you goes to Andres Löh and Ross Paterson for various bug reports and suggestions for improvement.

Chapter 2

Quick start

First install Frown as described in Sec. 1.1. Then enter the directory `QuickStart`.

```
ralf/Frown> cd QuickStart
```

The file `Tiger.lg`, listed in Fig. 2.1, contains a medium-sized grammar for an imperative language. Now, type

```
ralf/Frown/QuickStart> frown -v Tiger.lg
ralf/Frown/QuickStart> hugs Tiger.hs
...
Tiger> exp [ID "a", PLUS, ID "b", TIMES, INT "2"] >>= print
Bin (Var "a") Add (Bin (Var "b") Mul (Int "2"))
Tiger> tc "fib.tig"
...
```

The call `frown -v Tiger.lg` generates a Haskell parser which can then be loaded into `hugs` (or `ghci`). The parser has type $exp :: (Monad\ m) \Rightarrow [\,Terminal\,] \rightarrow m\ Expr$, that is, the parser is a computation that takes a list of terminals as input and returns an expression.

More examples can be found in the directory `Manual/Examples`:

`Paren1.lg`
: well-balanced parentheses: a pure grammar (see Sec. 3.2.1);

`Paren2.lg`
: an extension of `Paren1.lg` illustrating the definition of attributes (see Sec. 3.2.2);

`Calc.lg`
: a simple evaluator for arithmetic expressions: a parser that interfaces with a separate lexer (see Sec. 3.2.3);

`MCalc.lg`
: a variant of the desktop calculator (`Calc.lg`) that prints all intermediate results: illustrates monadic actions (see Sec. 3.2.4);

`Let1.lg`
: an ambiguous expression grammar: illustrates backtracking parsers (see Sec. 3.2.5);

`Let2.lg`
: an expression grammar: illustrates the use of precedences and associativity (see Sec. 3.2.6);

`Let3.lg`
: a variant of the expression grammar: shows how to simulate inherited attributes using a reader monad (see Sec. 3.2.8);

A grammar file consists of two parts: the specification of the grammar, enclosed in special curly braces, and Haskell source code. The source file typically starts with a Haskell module header.

```
module Tiger where
import Lexer
import Syntax
import Prelude hiding (exp)

%{
```

The grammar part begins here. A context-free grammar consists of sets of terminal and nonterminal symbols, a set of start symbols, and set of productions or grammar rules. The declaration below introduces the terminal symbols. Each terminal is given by a Haskell pattern of type *Terminal*.

```
Terminal = DO | ELSE | END | FUNCTION | IF
         | IN | LET | THEN | VAR | WHILE
         | ASSIGN as ":=" | COLON as ":" | COMMA as "," | CPAREN as ")"
         | DIV as "/" | EQU as "=" | LST as "<=" | MINUS as "-"
         | NEG as "~" | OPAREN as "(" | PLUS as "+" | SEMI as ";"
         | TIMES as "*"
         | ID{String} | INT{String};
```

A terminal symbol may carry a semantic value or attribute. The Haskell type of the semantic value is given in curly braces. As a rule, Haskell source code is always enclosed in curly braces within the grammar part. The as-clauses define shortcuts for terminals, which may then be used in the productions.
The declaration below introduces a nonterminal symbol called *exp* followed by sixteen productions for that symbol. The asterix marks *exp* as a start symbol; *exp* has a single attribute of type *Expr*.

```
*exp{Expr};
 exp{Var v} : ID{v};
    {Block es} | "(", sepBy exp ";"{es}, ")";
    {Int i} | INT{i};
    {Un Neg e} | "-", exp{e}, prec "~";
    {Call f es} | ID{f}, "(", sepBy exp ","{es}, ")";
    {Bin e1 Eq e2} | exp{e1}, "=", exp{e2};
    {Bin e1 Leq e2} | exp{e1}, "<=", exp{e2};
    {Bin e1 Add e2} | exp{e1}, "+", exp{e2};
    {Bin e1 Sub e2} | exp{e1}, "-", exp{e2};
    {Bin e1 Mul e2} | exp{e1}, "*", exp{e2};
    {Bin e1 Div e2} | exp{e1}, "/", exp{e2};
    {Assign v e} | ID{v}, ":=", exp{e};
    {IfThen e e1} | IF, exp{e}, THEN, exp{e1};
    {IfElse e e1 e2} | IF, exp{e}, THEN, exp{e1}, ELSE, exp{e2};
    {While e e1} | WHILE, exp{e}, DO, exp{e1};
    {Let ds es} | LET, many dec{ds}, IN, sepBy exp ";"{es}, END;
```

Left-hand and right-hand side of a production are separated by a colon; symbols on the right are separated by commas and terminated by a semicolon. Alternative right-hand sides are separated by a vertical bar.
The pieces in curly braces constitute Haskell source code. Each rule can be seen as a function from the right-hand to the left-hand side. On the right-hand side, Haskell variables are used to name the values of attributes. The values of the attributes on the left-hand side are given by Haskell expressions, in which the variables of the right-hand side occur free.
The last production makes use of two (predefined) rule schemes: *many x* implements the repetition of the symbol x, and *sepBy x sep* denotes a repetition of x symbols separated by *sep* symbols.
The above productions are ambiguous as, for instance, $1 + 2 * 3$ has two derivations. The ambiguity can be resolved by assigning precedences to terminal symbols.

```
left 7 "~"; left 6 "*"; left 6 "/"; left 5 "+"; left 5 "-";
right 0 THEN; right 0 ELSE;
nonassoc 4 "<="; nonassoc 4 "="; nonassoc 0 DO; nonassoc 0 ":=";
```

The following declarations define the nonterminal *dec* and three further nonterminals.

```
dec{Decl};
dec{d} : vardec{d};
   {d} | fundec{d};

vardec{Decl};
vardec{Variable v e} : VAR, ID{v}, ":=", exp{e};

fundec{Decl};
fundec{Function f xs e} : FUNCTION, ID{f}, "(", sepBy (ID{}) ","{xs}, ")", "=", exp{e};

formal{(Ident, TyIdent)};
formal{(v, t)} : ID{v}, ":", ID{t};

}%
```

The grammar part ends here. The source file typically includes a couple of Haskell declarations. The user-defined function *frown* is the error routine invoked by the parser in case of a syntax error; its definition is mandatory.

```
frown _ = error "syntax error"

tc f = do {putStrLn "*** reading ..."; s <- readFile f; print s;
           putStrLn "*** lexing  ..."; let {ts = lexer s}; print ts;
           putStrLn "*** parsing ..."; e <- exp ts; print e}
```


Figure 2.1: A sample Frown grammar file.

`Let4.lg`
: an expression grammar: illustrates a parser that interfaces with a monadic lexer (see Sec. 3.3.1);

`Let5.lg`
: a variant of `Let4.lg` with better error reporting (see Sec. 3.3.2);

`Let6.lg`
: a variant of `Let5.lg` with even better error reporting: prints a list of expected tokens upon error (see Sec. 3.3.3);

`Let7.lg`
: yet another variant of the expression grammar: illustrates a simple form of error correction (see Sec. 3.3.4);

`Let8.lg`
: variant of `Let7.lg` that notifies the user of corrections (see Sec. 3.3.4);

`VarCalc.lg`
: a variant of the desktop calculator (`Calc.lg`) that works without a separate lexer: illustrates guards (see Sec. 3.4.2);

`Paren3.lg`
: illustrates the tracing facilities (see Sec. 3.4.4);

`VarParen.lg`
: illustrates irrefutable patterns on the right-hand side of productions (see Sec. 4.1);

`RepMin.lg`
: a solution to the rep-min problem: illustrates how to simulate inherited attributes using functional attributes (see Sec. 4.2).

Chapter 3

Tour de Frown

This chapter introduces Frown by means of example.

3.1 Preliminaries: monads

Some elementary knowledge of *monads* is helpful in order to use Frown effectively. For the most basic applications, however, one can possibly do without. This section summarizes the relevant facts.

In Haskell, the concept of a monad is captured by the following class definition.

$$
\begin{aligned}
&\textbf{class } \mathit{Monad}\ m\ \textbf{where}\\
&\quad \mathit{return} :: a \to m\ a\\
&\quad (\ggg\!=) :: m\ a \to (a \to m\ b) \to m\ b\\
&\quad (\gg) :: m\ a \to m\ b \to m\ b\\
&\quad \mathit{fail} :: \mathit{String} \to m\ a\\
&\\
&\quad m \gg n = m \ggg\!= \mathit{const}\ n\\
&\quad \mathit{fail}\ s = \mathit{error}\ s
\end{aligned}
$$

The essential idea of monads is to distinguish between *computations* and *values*. This distinction is reflected on the type level: an element of $m\ a$ represents a computation that yields a value of type a. The trivial or pure computation that immediately returns the value a is denoted $\mathit{return}\ a$. The operator $(\ggg\!=)$, commonly called 'bind', combines two computations: $m \ggg\!= k$ applies k to the result of the computation m. The derived operation $(\gg)$ provides a handy shortcut if one is not interested in the result of the first computation. Finally, the operation *fail* is useful for signaling error conditions (a common thing in parsing).

Framing the concept of a monad as a type class is sensible for at least two interrelated reasons. First, we can use the same names (*return*, '$\ggg\!=$', and *fail*) for wildly different computational structures.[1] Second, by overloading a function with the monad class we effectively parameterize the function by computational structures, that is, we can call the same function with different instances of monads obtaining different computational effects.

The following instance declaration (`Result.lhs`) defines a simple monad that we will use intensively in the sequel (the monad can be seen as a simplified term implementation of the basic

[1] In fact, we can use the same notation, the so-called **do**-*notation*, for different monads (cf Haskell Report §3.14).

monad operations).

```
module Result where

data Result a = Return a | Fail String
                deriving (Show)

instance Monad Result where
    return = Return
    Fail s >>= k = Fail s
    Return a >>= k = k a
    fail = Fail
```

In monad speak, this is an *exception monad*: a computation in *Result* either succeeds gracefully yielding a value *a* (represented by the term *Return a*) or it fails with an error message *s* (represented by *Fail s*). That's all we initially need for Frown: parsing a given input either succeeds producing a semantic value (sometimes called an attribution) or it fails (hopefully, with a clear indication of the syntax error).

3.2 Basic features

3.2.1 Pure grammars

Let's start with a simple example. The following complete Frown source file (`Paren1.lg`[2]) defines the language of well-balanced parentheses. The specification of the grammar is enclosed in special curly braces '%{ ... }%'. The remainder contains Haskell source code, that is, a module header and a function declaration.

```
module Paren where
import Result

%{

Terminal = '(' | ')';

Nonterminal = paren;

paren :;
paren : paren, '(', paren, ')';

}%

frown _ = fail "syntax error"
```

The part enclosed in special curly braces comprises the typical ingredients of a *context-free grammar*: a declaration of the *terminal symbols*, a declaration of the *nonterminal symbols*, and finally the *productions* or *grammar rules*.

In general, the terminal symbols are given by Haskell patterns of the same type. Here, we have two character patterns of type *Char*.

Nonterminals are just identifiers starting with a lower-case letter. By convention, the first nonterminal is also the start symbol of the grammar (this default can be overwritten, see Sec. 3.2.7).

Productions have the general form $n : v_1, \ldots, v_k;$ where n is a nonterminal and $v_1, \ldots, v_k$ are symbols. Note that the symbols are separated by commas and terminated by a semicolon. The

[2]The source files of the examples are located in the directory `Manual/Examples`.

mandatory trailing semicolon helps to identify so-called *ϵ-productions*, productions with an empty right-hand side, such as *paren* : ;.

As a shorthand, we allow to list several alternative right-hand sides separated by a vertical bar. Thus, the above productions could have been written more succinctly as

paren:;
    | *paren*, `'('`, *paren*, `')'`;

The two styles can be arbitrarily mixed. In fact, it is not even required that productions with the same left-hand side are grouped together (though it is good style to do so).

Now, assuming that the above grammar resides in a file called `Paren.g` we can generate a Haskell parser by issuing the command

```
frown Paren.g
```

This produces a Haskell source file named `Paren.hs` that contains among other things the function

$$paren :: (Monad\ m) \Rightarrow [\,Char\,] \rightarrow m\ (),$$

which recognizes the language generated by the start symbol of the same name. Specifically, if *inp* is a list of characters, then *paren inp* is a computation that either succeeds indicating that *inp* is a well-formed parentheses or fails indicating that *inp* isn't well-formed. Here is a short interactive session using the Haskell interpreter Hugs (type `hugs Paren.hs` at the command line).

Paren ≫ *paren* `"(())()"` :: *Result* ()
Return ()
Paren ≫ *paren* `"(())("` :: *Result* ()
Fail `"syntax error"`

Note that we have to specify the result type of the expressions in order to avoid an unresolved overloading error. Or to put it differently, we have to specify the monad, in which the parsing process takes place. Of course, we are free to assign *paren* a more constrained type by placing an appropriate type signature in the Haskell section of the grammar file:

$$paren :: [\,Char\,] \rightarrow Result\ ().$$

By the way, since the nonterminal *paren* carries no semantic value, the type of the computation is simply *Result* () where the empty tuple type '()' serves as a dummy type. In the next section we will show how to add attributes or semantic values to nonterminals.

Every once in a while parsing fails. In this case, Frown calls a user-supplied function named, well, *frown* (note that you must supply this function). In our example, *frown* has type

$$frown :: (Monad\ m) \Rightarrow [\,Char\,] \rightarrow m\ a$$

The error function *frown* is passed the remaining input as an argument, that you can give an indication of the location of the syntax error (more on error reporting in Sec. 3.3). Note that *frown* must be polymorphic in the result type.

Remark 1 *For those of you who are knowledgable and/or interested in LR parsing, Fig. 3.1 displays the Haskell file that is generated by* `frown Paren.g`[3]. *For each state i of the underlying LR(0) automaton, displayed in Fig. 3.2, there is one function called parse_i. All these functions take two arguments: the remaining input and a stack that records the transitions of the LR(0) machine. The reader is invited to trace the parse of* `"(())()"`.

[3] Actually, the file is generated using `frown --suffix Paren.g`, see Sec. 5.5.

```
module Paren where
import Result

 {- frown :-( -}

data Stack = Empty
           | T_1_2 Stack
           | T_2_3 Stack
           | T_2_5 Stack
           | T_4_5 Stack
           | T_4_6 Stack
           | T_5_4 Stack

data Nonterminal = Paren

paren tr = parse_1 tr Empty >>= (\Paren -> return ())

parse_1 ts st = parse_2 ts (T_1_2 st)

parse_2 tr@[] st = parse_3 tr (T_2_3 st)
parse_2 ('(' : tr) st = parse_5 tr (T_2_5 st)
parse_2 ts st = frown ts

parse_3 ts (T_2_3 (T_1_2 st)) = return Paren

parse_4 ('(' : tr) st = parse_5 tr (T_4_5 st)
parse_4 (')' : tr) st = parse_6 tr (T_4_6 st)
parse_4 ts st = frown ts

parse_5 ts st = parse_4 ts (T_5_4 st)

parse_6 ts (T_4_6 (T_5_4 (T_2_5 (T_1_2 st)))) =
                                    = parse_2 ts (T_1_2 st)
parse_6 ts (T_4_6 (T_5_4 (T_4_5 (T_5_4 st))))
                                    = parse_4 ts (T_5_4 st)

 {- )-: frown -}

frown _ = fail "syntax error"
```

Figure 3.1: A Frown generated parser.

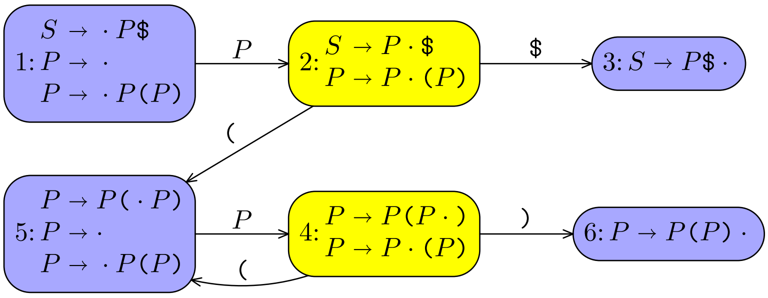


Figure 3.2: The LR(0) automaton underlying the parser of Fig. 3.1.

3.2.2 Attributes

Now, let's augment the grammar of Sec. 3.2.1 by semantic values (`Paren2.lg`). Often, the parser converts a given input into some kind of tree representation (the so-called *abstract syntax tree*). To represent nested parentheses we simply use binary trees (an alternative employing n-ary trees is given in Sec. 4.1).

```
module Paren where
import Result

data Tree = Leaf | Fork Tree Tree
            deriving (Show)

%{

Terminal = '(' | ')';

Nonterminal = paren{Tree};

paren{Leaf}:;
     {Fork t u} | paren{t}, '(', paren{u}, ')';


}%

frown _ = fail "syntax error"
```

Attributes are always given in curly braces. When we declare a nonterminal, we have to specify the types of its attributes as in *paren*{*Tree*}. The rules of the grammar can be seen as functions from the right-hand side to the left-hand side. On the right-hand side, Haskell variables are used to name the values of attributes. The values of the attributes on the left-hand side are then given by Haskell expressions, in which the variables of the right-hand side may occur free. The Haskell expressions can be arbitrary, except that they must not be layout-sensitive.

In general, a nonterminal may have an arbitrary number of attributes (see Sec. 4.4 for an example). Note that Frown only supports so-called *synthesized attributes* (*inherited attributes* can be simulated, however, with the help of a reader monad, see Sec. 3.2.8, or with functional attributes, see Sec. 4.2).

The parser generated by Frown now has type

$$paren :: (Monad\ m) \Rightarrow [Char] \to m\ Tree.$$

The following interactive session illustrates its use.

$Paren \gg paren$ `"(())()"` $:: Result\ Tree$
$Return\ (Fork\ (Fork\ Leaf\ (Fork\ Leaf\ Leaf))\ Leaf)$
$Paren \gg paren$ `"(())("` $:: Result\ Tree$
$Fail$ `"syntax error"`

3.2.3 Interfacing with a lexer

The parsers of the two previous sections take a list of characters as input. In practice, a parser usually does not work on character streams directly. Rather, it is prefaced by a lexer that first converts the characters into a list of so-called *tokens*. The separation of the lexical analysis from the syntax analysis usually leads to a clearer design and as a benevolent side-effect it also improves efficiency (Sec. 3.4.2 shows how to combine lexing and parsing in Frown, though).

A simple token type is shown in Fig 3.3 (`Terminal1.lhs`). (Note that the type comprises more constructors than initially needed.)

Fig. 3.4 (`Lexer.lhs`) displays a simple lexer for arithmetic expressions, which are built from numerals using the arithmetic operators '`+`', '`-`', '`*`', and '`/`'.

The following grammar, which builds upon the lexer, implements a simple evaluator for arithmetic expressions (`Calc.lg`).

```
module Calc where
import Lexer
import Result

%{

Terminal = Numeral{Int}
         | Addop{Op}
         | Mulop{Op}
         | LParen as "("
         | RParen as ")";

Nonterminal = expr{Int}
            | term{Int}
            | factor{Int};

expr{app op v1 v2} : expr{v1}, Addop{op}, term{v2};
    {e} | term{e};
term{app op v1 v2} : term{v1}, Mulop{op}, factor{v2};
    {e} | factor{e};
factor{n} : Numeral{n};
    {e} | "(", expr{e}, ")";

}%

frown _ = fail "syntax error"
```

The terminal declaration now lists patterns of type *Terminal*. Note that terminals may also carry semantic values. The single argument of *Numeral*, for instance, records the numerical value of the numeral.

When declaring a terminal we can optionally define a shortcut using an *as*-clause as, for example, in *LParen as* `"("`. The shortcut can be used in the productions possibly improving their readability.

```haskell
module Terminal where
import Maybe

data Op = Plus | Minus | Times | Divide
          deriving (Show)

name :: Op -> String
name Plus = "+"
name Minus = "-"
name Times = "*"
name Divide = "/"

app :: Op -> (Int -> Int -> Int)
app Plus = (+)
app Minus = (-)
app Times = (*)
app Divide = div

data Terminal = Numeral Int
              | Ident String
              | Addop Op
              | Mulop Op
              | KWLet
              | KWIn
              | Equal
              | LParen
              | RParen
              | EOF
               deriving (Show)

ident, numeral :: String -> Terminal
ident s = fromMaybe (Ident s) (lookup s keywords)
numeral s = Numeral (read s)

keywords :: [(String, Terminal)]
keywords = [("let", KWLet), ("in", KWIn)]
```

Figure 3.3: The type of terminals (`Terminal1.lhs`).

```
module Lexer (module Terminal, module Lexer) where
import Char
import Terminal

lexer :: String → [Terminal]
lexer [] = []
lexer ('+' : cs) = Addop Plus : lexer cs
lexer ('-' : cs) = Addop Minus : lexer cs
lexer ('*' : cs) = Mulop Times : lexer cs
lexer ('/' : cs) = Mulop Divide : lexer cs
lexer ('=' : cs) = Equal : lexer cs
lexer ('(' : cs) = LParen : lexer cs
lexer (')' : cs) = RParen : lexer cs
lexer (c : cs)
    | isAlpha c = let (s, cs') = span isAlphaNum cs in ident (c : s) : lexer cs'
    | isDigit c = let (s, cs') = span isDigit cs in numeral (c : s) : lexer cs'
    | otherwise = lexer cs
```

Figure 3.4: A simple lexer for arithmetic expressions (`Lexer.lhs`).

Here is an example session demonstrating the evaluator.

```
Calc ≫ lexer "4 * (7 + 1)"
[Numeral 4, Mulop Times, LParen, Numeral 7, Addop Plus, Numeral 1, RParen]
Calc ≫ expr (lexer "4711") :: Result Int
Return 4711
Calc ≫ expr (lexer "4 * (7 + 1) - 1") :: Result Int
Return 31
Calc ≫ expr (lexer "4 * (7 + 1 - 1") :: Result Int
Fail "syntax error"
```

3.2.4 Monadic actions

The expression that determines the value of an attribute is usually a pure one. It is, however, also possible to provide a monadic action that *computes* the value of the attribute. The computation lives in the underlying parsing monad. Monadic actions are enclosed in '$\{\%\lambda\ldots\}$' braces and have type $m\ t$ where m is the type of the underlying monad and t is the type of attributes.

As an example, the following variant of the desktop calculator (`MCalc.lg`) prints all intermediate results (note that we only list the changes to the preceeding example).

```
trace :: Op → (Int → Int → IO Int)
trace op v1 v2 = putStrLn s ≫ return v
    where v = app op v1 v2
          s = show v1 ++ name op ++ show v2 ++ "=" ++ show v

expr{%trace op v1 v2} : expr{v1}, Addop{op}, term{v2};
term{%trace op v1 v2} : term{v1}, Mulop{op}, factor{v2};
```

The following session illustrates its working.

$MCalc \gg expr\ (lexer$ `"4711"`$)$
4711
$MCalc \gg expr\ (lexer$ `"4 * (7 + 1) - 1"`$)$
$7 + 1 = 8$
$4 * 8 = 32$
$32 - 1 = 31$
31
$MCalc \gg expr\ (lexer$ `"4 * (7 + 1 - 1"`$)$
$7 + 1 = 8$
`Program error: user error (syntax error)`

In general, monadic actions are useful for performing 'side-effects' (for example, in order to parse `%include` directives) and for interaction with a monadic lexer (see Sec. 3.3.1).

3.2.5 Backtracking parsers

In the previous examples we have encoded the precedences of the operators ('$*$' binds more tightly than '$+$') into the productions of the grammar. However, this technique soon becomes unwieldy for a larger expression language. So let's start afresh. The grammar file shown in Fig. 3.5 (`Let1.lg`) uses only a single nonterminal for expressions (we have also extended expressions by local definitions). Also note that the grammar has no *Nonterminal* declaration. Rather, the terminal symbols are declared by supplying type signatures before the respective rules. Generally, type signatures are preferable to a *Nonterminal* declaration if the grammar is long.

Of course, the rewritten grammar is no longer LALR(k) simply because it is ambiguous. For instance, '$1 + 2 * 3$' can be parsed as $Bin\ (Const\ 1)\ Plus\ (Bin\ (Const\ 2)\ Times\ (Const\ 3))$ or as $Bin\ (Bin\ (Const\ 1)\ Plus\ (Const\ 2))\ Times\ (Const\ 3)$. Frown is also unhappy with the grammar: it reports six shift/reduce conflicts:

```
* warning: 6 shift/reduce conflicts
```

This means that Frown wasn't able to produce a deterministic parser. Or rather, it produced a deterministic parser by making some arbitrary choices to avoid non-determinism (shifts are preferred to reductions, see Sec. 3.2.6). However, we can also instruct Frown to produce a non-deterministic parser, that is, one that generates all possible parses of a given input. We do so by supplying the option `--backtrack`:

```
frown --backtrack Let.g
```

The generated parser *expr* now has type

$$expr :: (MonadPlus\ m) \Rightarrow [\,Terminal\,] \rightarrow m\ Expr.$$

Note that the underlying monad must be an instance of *MonadPlus* (defined in the standard library *Monad*). The list monad and the *Maybe* monad are both instances of *MonadPlus*. The following session shows them in action.

$Let \gg expr\ (lexer$ `"1 + 2 - 3 * 4 / 5"`$) :: [Expr]$
$[Bin\ (Const\ 1)\ Plus\ (Bin\ (Const\ 2)\ Minus\ (Bin\ (Const\ 3)\ Times\ (Bin\ (Const\ 4)\ Divide\ (Const\ 5)))), Bin\ (Co$
$Let \gg expr\ (lexer$ `"1 + - 3 * 4 / 5"`$) :: [Expr]$
$[\,]$
$Let \gg expr\ (lexer$ `"1 + 2 - 3 * 4 / 5"`$) :: Maybe\ Expr$
$Just\ (Bin\ (Const\ 1)\ Plus\ (Bin\ (Const\ 2)\ Minus\ (Bin\ (Const\ 3)\ Times\ (Bin\ (Const\ 4)\ Divide\ (Const\ 5)))))$

The list monad supports 'deep backtracking': all possible parses are returned (beware: the number grows exponentionally). The *Maybe* monad implements 'shallow backtracking': it commits to the first solution (yielding the same results as the parser generated *without* the option `--backtrack`).

```
module Let where
import Lexer
import Monad

data Expr = Const Int | Var String | Bin Expr Op Expr | Let Decl Expr
            deriving (Show)

data Decl = String :=: Expr
            deriving (Show)

%{

Terminal = Numeral{Int}
         | Ident{String}
         | Addop{Op}
         | Mulop{Op}
         | KWLet as "let"
         | KWIn as "in"
         | Equal as "="
         | LParen as "("
         | RParen as ")";

expr{Expr};
expr{Const n} : Numeral{n};
    {Var s} | Ident{s};
    {Bin e1 op e2} | expr{e1}, Addop{op}, expr{e2};
    {Bin e1 op e2} | expr{e1}, Mulop{op}, expr{e2};
    {Let d e} | "let", decl{d}, "in", expr{e};
    {e} | "(", expr{e}, ")";

decl{Decl};
decl{s :=: e} : Ident{s}, "=", expr{e};

}%

frown _ = fail "syntax error"
```

Figure 3.5: An ambiguous grammar (Let1.lg).

3.2.6 Precedences and associativity

Instead of resorting to a backtracking parser we may also help Frown to generate the 'right' deterministic parser by assigning *precedences* to terminal symbols. The understand the working of precedences it is necessary to provide some background of the underlying parsing technique.

LR parsers work by repeatedly performing two operations: *shifts* and *reductions*. A shift moves a terminal from the input onto the stack, the auxiliary data structure maintained by the parser. A reduction replaces a top segment of the stack matching the right-hand side of a production by its left-hand side. Parsing succeeds if the input is empty and the stack consists of a start symbol only. As an example, consider parsing '$N * N + N$'.

	$N * N + N$	shift
N	$* N + N$	reduce by $e : N;$
e	$* N + N$	shift
$e *$	$N + N$	shift
$e * N$	$+ N$	reduce by $e : N;$
$e * e$	$+ N$	

At this point, there are two possibilities: we can either perform a reduction (using the production $e : e, *, e;$) or shift the next input symbol. Both choices are viable.

$e * e$	$+ N$	reduce by $e : e, *, e;$
e	$+ N$	shift
$e +$	N	shift
$e + N$		reduce by $e : N;$
$e + e$		reduce by $e : e, +, e;$
e		

$e * e$	$+ N$	shift
$e * e +$	N	shift
$e * e + N$		reduce by $e : N;$
$e * e + e$		reduce by $e : e, +, e;$
$e * e$		reduce by $e : e, *, e;$
e		

Alas, the two choices also result in different parse trees. By default, Frown prefers shifts to reductions. As a consequence, $N * N + N$ is parsed as $N * (N + N)$, that is, '+' binds more tightly than '$*$'.

Now, we can direct the resolution of conflicts by assigning *precedences* and *associativity* to terminal symbols. The following declarations will do in our example (`Let2.g`).

left 6 *Addop*{ };
left 7 *Mulop*{ };
nonassoc 0 `"in"`;

Thus, '$*$' takes precedence over '+', which in turn binds more tightly than '**in**'. For instance, **let** $a = 4$ **in** $a + 2$ is parsed as **let** $a = 4$ **in** $(a + 2)$. A conflict between two symbols of equal precedence is resolved using *associativity*: the succession $1 + 2 + 3$ of left-associative operators is grouped as $(1 + 2) + 3$; likewise for right-associative operators; sequences of non-associative operators are not well-formed.

Given the fixity declarations above Frown now produces the 'right' deterministic parser, which can be seen in action below.

Let ≫ *expr* (*lexer* `"4 * (7 + 1) - 1"`) :: *Result Expr*
Return (*Bin* (*Bin* (*Const* 4) *Times* (*Bin* (*Const* 7) *Plus* (*Const* 1))) *Minus* (*Const* 1))
Let ≫ *expr* (*lexer* `"4 * 7 + 1 - 1"`) :: *Result Expr*
Return (*Bin* (*Bin* (*Bin* (*Const* 4) *Times* (*Const* 7)) *Plus* (*Const* 1)) *Minus* (*Const* 1))
Let ≫ *expr* (*lexer* `"let\n    a = 4 * (7 + 1) - 1\n in a * a"`) :: *Result Expr*
Return (*Let* (`"a"` :=: *Bin* (*Bin* (*Const* 4) *Times* (*Bin* (*Const* 7) *Plus* (*Const* 1))) *Minus* (*Const* 1)) (*Bin* (*Var*
Let ≫ *expr* (*lexer* `"let\n    a = 4 * (7 + 1 - 1\n in a * a"`) :: *Result Expr*
Fail `"syntax error"`

In general, a conflict between the actions 'reduce by rule r' and 'shift terminal t' is resolved as follows (the precedence of a rule is given by the precedence of the rightmost terminal on the right-hand side):

condition		action	example
prec $r <$ *prec* t		shift	reduce by $e : e, +, e$; versus shift $*$
	left t	reduce	reduce by $e : e, *, e$; versus shift $*$
prec $r =$ *prec* t	**right** t	shift	reduce by $e : e, +\!\!+, e$; versus shift $+\!\!+$
	nonassoc t	fail	reduce by $e : e, ==, e$; versus shift $==$
prec $r >$ *prec* t		reduce	reduce by $e : e, *, e$; versus shift $+$

Just in case you may wonder: there are no shift/shift conflicts by construction; reduce/reduce conflicts cannot be cured using precedences and associativity.

3.2.7 Multiple start symbols

A grammar may have several start symbols. In this case, Frown generates multiple parsers, one for each start symbol (actually, these are merely different entry points into the LR(0) automaton[4]). We mark a symbol as a start symbol simply by putting an asterix before its declaration (either in a *Nonterminal* declaration or in a separate type signature). Consider our previous example: most likely we want parsers both for expressions and declarations. Thus, we write

$$* \, expr\{Expr\};$$
$$* \, decl\{Decl\};$$

and get parsers of type.

$$expr :: (Monad\ m) \Rightarrow [Terminal] \to m\ Expr$$
$$decl :: (Monad\ m) \Rightarrow [Terminal] \to m\ Decl.$$

3.2.8 Monadic attributes

This section does not introduce any new features of Frown and can be safely skipped on first reading. Its purpose is to show how to simulate inherited attributes using a reader monad (see also Sec. 4.2). Generally, inherited attributes are used to pass context information down the parse tree. As an example, consider implementing an evaluator for arithmetic expressions that include variables and **let**-bindings (`Let3.lg`). To determine the value of a variable we need to pass down an environment that records the values of bound variables. The reader monad displayed in Fig. 3.6 (`Reader.lhs`) serves exactly this purpose. We need some additional helper functions for accessing and extending environments

type *Binding* = (*String*, *Int*)

type *Result* = *Reader* [*Binding*]

$$extend :: Binding \to Result\ a \to Result\ a$$
$$extend\ b\ m = getenv \ggg\!= \lambda env \to withenv\ (b : env)\ m$$

$$access :: String \to Result\ Int$$
$$access\ s = getenv \ggg\!= \lambda env \to return\ (fromMaybe\ 0\ (lookup\ s\ env))$$

[4]There is, however, a small cost involved: for each start symbol s Frown silently introduces a new symbol s' and a new rule $s' : s, EOF$. This increases the size of the automaton by a few states.

$$
\begin{array}{l}
\textbf{module } \mathit{Reader} \textbf{ where} \\[1em]
\textbf{newtype } \mathit{Reader}\ \mathit{env}\ a = \mathit{Reader}\{\ \mathit{apply} :: \mathit{env} \to a\} \\[1em]
\textbf{instance } \mathit{Monad}\ (\mathit{Reader}\ \mathit{env}) \textbf{ where} \\
\quad \mathit{return}\ a = \mathit{Reader}\ (\lambda \mathit{env} \to a) \\
\quad m \gg\!= k = \mathit{Reader}\ (\lambda \mathit{env} \to \mathit{apply}\ (k\ (\mathit{apply}\ m\ \mathit{env}))\ \mathit{env}) \\
\quad \mathit{fail}\ s = \mathit{Reader}\ (\mathit{error}\ s) \\[1em]
\mathit{getenv} :: \mathit{Reader}\ \mathit{env}\ \mathit{env} \\
\mathit{getenv} = \mathit{Reader}\ (\lambda \mathit{env} \to \mathit{env}) \\[1em]
\mathit{withenv} :: \mathit{env} \to \mathit{Reader}\ \mathit{env}\ a \to \mathit{Reader}\ \mathit{env}'\ a \\
\mathit{withenv}\ \mathit{env}\ m = \mathit{Reader}\ (\lambda \mathit{env}' \to \mathit{apply}\ m\ \mathit{env})
\end{array}
$$

Figure 3.6: The reader monad (`Reader.lhs`).

The following grammar implements the desired evaluator.

$$
\begin{array}{l}
\mathit{expr}\{\ \mathit{Result}\ \mathit{Int}\}; \\
\mathit{expr}\{\textbf{do}\ \{b \leftarrow d; \mathit{extend}\ b\ m\}\} : \texttt{"let"}, \mathit{decl}\{\ d\}, \texttt{"in"}, \mathit{expr}\{\ m\}; \\
\quad \{\ \mathit{liftM2}\ (\mathit{app}\ \mathit{op})\ m_1\ m_2\} \mid \mathit{expr}\{\ m_1\}, \mathit{Addop}\{\ \mathit{op}\}, \mathit{expr}\{\ m_2\}; \\
\quad \{\ \mathit{liftM2}\ (\mathit{app}\ \mathit{op})\ m_1\ m_2\} \mid \mathit{expr}\{\ m_1\}, \mathit{Mulop}\{\ \mathit{op}\}, \mathit{expr}\{\ m_2\}; \\
\quad \{\ \mathit{return}\ n\} \mid \mathit{Numeral}\{\ n\}; \\
\quad \{\ \mathit{access}\ s\} \mid \mathit{Ident}\{\ s\}; \\
\quad \{\ m\} \mid \texttt{"("}, \mathit{expr}\{\ m\}, \texttt{")"}; \\[1em]
\mathit{decl}\{\ \mathit{Result}\ \mathit{Binding}\}; \\
\mathit{decl}\{\textbf{do}\ \{v \leftarrow m; \mathit{return}\ (s, v)\}\} : \mathit{Ident}\{\ s\}, \texttt{"="}, \mathit{expr}\{\ m\};
\end{array}
$$

Note that there are two monads around: the parsing monad (in fact, *expr* is parametric in this monad) and the reader monad, which is embedded in the attributes. The parser returns a value of type *Reader Int*, to which we pass an empty initial environment.

$$
\begin{array}{l}
\mathit{eval} :: (\mathit{Monad}\ m) \Rightarrow [\mathit{Char}] \to m\ \mathit{Int} \\
\mathit{eval}\ \mathit{inp} = \textbf{do}\ \{f \leftarrow \mathit{expr}\ (\mathit{lexer}\ \mathit{inp}); \mathit{return}\ (\mathit{apply}\ f\ [\,])\}
\end{array}
$$

Let's see the evaluator in action.

Let ≫ *eval* `"2 + 7"` :: *IO Int*
9
Let ≫ *eval* `"a + b"` :: *IO Int*
0
Let ≫ *eval* `"let x = 4 in x * x"` :: *IO Int*
16
Let ≫ *eval* `"let x = 4 in x * x + x"` :: *IO Int*
20
Let ≫ *eval* `"(let x = 4 in x * x) + x"` :: *IO Int*
16

3.3 Error reporting and correction

3.3.1 Monadic lexers

The chances that parsing succeeds are probably smaller than the chances that it fails. Good error messages are indispensable to turn the latter into the former case. Up to now we only produced the rather uninformative message `"syntax error"`. Fortunately, we are in a good position to do better. LR parsing has the nice property that it detects a syntax error at the earliest possible moment: parsing fails as soon as the input cannot be extended to a legal sentence of the grammar. For instance, the syntax error in **let** $a = 4 * (7 + 1 - 1$ **in** $a * a$ is detected after reading the keyword '**in**'.

Now, all we have to do is to keep track of context information: the current line and column number and possibly the filename. This section prepares the ground for maintaining state information; the parser that actually keeps track of line numbers etc is only shown in the next section.

Unsurprisingly, to maintain state information we employ monads again. This time, we require a state monad. The natural place for maintaining information about line numbers etc is, of course, the lexer. Consequently, we turn the stream-based lexer of type $String \to [\,Terminal\,]$ into a monadic one of type

$$get :: M\ \ Terminal$$

where M is the state monad. The idea is that each time *get* is called it returns the next token and updates its internal state.

The first version of the monadic lexer shown in Fig. 3.7 (`MLexer1.lhs`) has no internal state apart from the input stream, that is, it provides no additional functionality compared to the stream-based lexer. Note that we use a continuation-based state monad, *Lex m*, which requires local universal quantification (a non-Haskell 98 feature). Actually, *Lex* is even a *monad transformer* so that we can freely choose a base monad (such as *Result* or *IO*). Of course, an 'ordinary' state monad would do, as well. The monadic lexer *get* incorporates more or less the stream-based lexer. We only changed the recursive calls to lexer (ie $t : lexer\ cs$) into invocations of the continuation (ie $cont\ t\ cs$). The error routine *frown* now has type

$$frown :: (Monad\ m) \Rightarrow Terminal \to Lex\ m\ a,$$

that is, *frown* is no longer passed the remaining input but only the look-ahead token.

The changes to the grammar are minor: we have to declare an 'end of file' token marked by a star (`Let4.lg`)

```
Terminal = Numeral{ Int }
         | Ident{ String }
         | Addop{ Op }
         | Mulop{ Op }
         | KWLet as "let"
         | KWIn as "in"
         | Equal as "="
         | LParen as "("
         | RParen as ")"
         | *EOF;
```

and we have to provide a type signature for the generated parser (in the Haskell section).

$$expr :: (Monad\ m) \Rightarrow Lex\ m\ Expr$$

The type signature is necessary to avoid an 'unresolved top-level overloading' error (the monomorphism restriction strikes again).

When we generate the Haskell parser we must supply the option `--lexer` to inform Frown that we use a monadic lexer.

```
module MLexer (module Terminal, module MLexer) where
import Terminal
import Char

type CPS a answer = (a → answer) → answer

newtype Lex m a = Lex{ unLex :: ∀ans . CPS a (String → m ans)}

instance (Monad m) ⇒ Monad (Lex m) where
      return a = Lex (λcont → cont a)
      m ≫= k = Lex (λcont → unLex m (λa → unLex (k a) cont))
      fail s = lift (fail s)

lift :: (Monad m) ⇒ m a → Lex m a
lift m = Lex (λcont inp → m ≫= λa → cont a inp)

run :: (Monad m) ⇒ Lex m a → (String → m a)
run parser inp = unLex parser (λa rest → return a) inp

get :: (Monad m) ⇒ Lex m Terminal
get =
   Lex (λcont inp →
      let lexer [] = cont (EOF) []
          lexer ('+' : cs) = cont (Addop Plus) cs
          lexer ('-' : cs) = cont (Addop Minus) cs
          lexer ('*' : cs) = cont (Mulop Times) cs
          lexer ('/' : cs) = cont (Mulop Divide) cs
          lexer ('=' : cs) = cont (Equal) cs
          lexer ('(' : cs) = cont (LParen) cs
          lexer (')' : cs) = cont (RParen) cs
          lexer (c : cs)
              | isSpace c = lexer cs
              | isAlpha c = let (s, cs′) = span isAlphaNum cs in cont (ident (c : s)) cs′
              | isDigit c = let (s, cs′) = span isDigit cs in cont (numeral (c : s)) cs′
              | otherwise = lexer cs
      in lexer inp)

frown :: (Monad m) ⇒ Terminal → Lex m a
frown t = Lex (λcont inp →
                                    fail ("\n*** syntax error:\n" ++ context 4 inp))

context :: Int → String → String
context n inp = unlines (take n (lines inp ++ ["<end of input>"]))
```

Figure 3.7: A monadic lexer for the **let** language (`MLexer1.lhs`).

```
frown --lexer Let.g
```

For completeness, here is an interactive session (note that in the case of error the look-ahead token is *not* displayed).

```
Let ≫ run expr "4 * (7 + 1) - 1" :: IO Expr
Bin (Bin (Const 4) Times (Bin (Const 7) Plus (Const 1))) Minus (Const 1)
Let ≫ run expr "let\n    a = 4 * (7 + 1) - 1\n in a * a" :: IO Expr
Let ("a" :=: Bin (Bin (Const 4) Times (Bin (Const 7) Plus (Const 1))) Minus (Const 1)) (Bin (Var "a") Ti
Let ≫ run expr "let\n    a = 4 * (7 + 1 - 1\n in a * a" :: IO Expr

Program error: user error (
*** syntax error:
 a * a
<end of input>
)
```

3.3.2 Error reporting

The monadic lexer shown in Fig. 3.8 (`MLexer2.lhs`) builds upon the one given in the previous section. The state monad *Lex m* has been extended to keep track of the current line number and the current line itself. The current line is displayed in case of a lexical or syntax error. As an aside, note that the column number can be recreated from the rest of the input and the length of the current line.

The following session shows the new lexer in action.

```
Let ≫ run expr "4 * (7 + 1) - 1" :: IO Expr
Bin (Bin (Const 4) Times (Bin (Const 7) Plus (Const 1))) Minus (Const 1)
Let ≫ run expr "let\n    a = 4 * (7 + 1) - 1\n in a * a" :: IO Expr
Let ("a" :=: Bin (Bin (Const 4) Times (Bin (Const 7) Plus (Const 1))) Minus (Const 1)) (Bin (Var "a") Ti
Let ≫ run expr "let\n    a = 4 * [7 + 1 - 1)\n in a * a" :: IO Expr

Program error: user error (
*** lexical error at (line 2, column 13):
    a = 4 * [7 + 1 - 1)
            ^

 in a * a
<end of input>
)
Let ≫ run expr "let\n    a = 4 * (7 + 1 - 1\n in a * a" :: IO Expr

Program error: user error (
*** syntax error at (line 3, column 3):
 in a * a
  ^
<end of input>
)
```

In the case of a lexical error the cursor '^' points at the offending character. In the case of a syntax error the cursor points at the *last* character of the offending token (recall that the part of the input up to and including this token is the shortest prefix of the input that cannot be extended to a legal sentence of the grammar).

```
module MLexer (module Terminal, module MLexer) where
import Terminal
import Char

type CPS a answer = (a → answer) → answer

newtype Lex m a = Lex{ unLex :: ∀ans . CPS a (String → Int → String → m ans)}

instance (Monad m) ⇒ Monad (Lex m) where
    return a = Lex (λcont → cont a)
    m ≫= k = Lex (λcont → unLex m (λa → unLex (k a) cont))
    fail s = lift (fail s)

lift :: (Monad m) ⇒ m a → Lex m a
lift m = Lex (λcont inp line cur → m ≫= λa → cont a inp line cur)

run :: (Monad m) ⇒ Lex m a → (String → m a)
run parser inp = unLex parser (λa rest line cur → return a) inp 1 (current inp)

current :: String → String
current s = takeWhile (≠ '\n') s

get :: (Monad m) ⇒ Lex m Terminal
get =
  Lex (λcont inp line cur →
    let lexer [ ] n x = cont (EOF) [ ] n x
        lexer ('\n' : cs) n x = lexer cs (n + 1) (current cs)
        lexer ('+' : cs) n x = cont (Addop Plus) cs n x
        lexer ('-' : cs) n x = cont (Addop Minus) cs n x
        lexer ('*' : cs) n x = cont (Mulop Times) cs n x
        lexer ('/' : cs) n x = cont (Mulop Divide) cs n x
        lexer ('=' : cs) n x = cont (Equal) cs n x
        lexer ('(' : cs) n x = cont (LParen) cs n x
        lexer (')' : cs) n x = cont (RParen) cs n x
        lexer (c : cs) n x
            | isSpace c = lexer cs n x
            | isAlpha c = let (s, cs') = span isAlphaNum cs in cont (ident (c : s)) cs' n x
            | isDigit c = let (s, cs') = span isDigit cs in cont (numeral (c : s)) cs' n x
            | otherwise = fail ("\n*** lexical error at "
                                ++ position cs n x ++ ":\n"
                                ++ context 4 cs x)
    in lexer inp line cur)

frown :: (Monad m) ⇒ Terminal → Lex m a
frown t = Lex (λcont inp line cur →
                                      fail ("\n*** syntax error at "
                                            ++ position inp line cur ++ ":\n"
                                            ++ context 4 inp cur))

position :: String → Int → String → String
position inp line cur = "(line " ++ show line ++ ", column " ++ show col ++ ")"
    where col = length cur − length (current inp)

context :: Int → String → String → String
context n inp cur = unlines ([cur, replicate col' ' ' ++ "^"]
                             ++ take n (lines (drop 1 (dropWhile (≠ '\n') inp))
                                        ++ ["<end of input>"]))
    where col' = length cur − length (current inp) − 1
```

Figure 3.8: A monadic lexer for the **let** language featuring good error reports (`MLexer2.lhs`).

3.3.3 Expected tokens

We can do even better! We can instruct Frown to pass a list of *expected* tokens to the error routine *frown* (by supplying the option `--expected`).

```
frown --lexer --expected Let.g
```

Frown uses the shortcuts given in the terminal declaration for generating lists of expected tokens. This means, in particular, that a token is *not* included in such a list if it does not have a shortcut. In our running example, we want every token to be listed. Therefore, we add shortcuts for every terminal symbol (`Let6.lg`).

```
Terminal = Numeral{Int} as "<numeral>"
         | Ident{String} as "<identifier>"
         | Addop{Op} as "+ or -"
         | Mulop{Op} as "* or /"
         | KWLet as "let"
         | KWIn as "in"
         | Equal as "="
         | LParen as "("
         | RParen as ")"
         | *EOF as "<end of input>";
```

The error routine *frown* now takes an additional argument of type [*String*] (`MLexer3.lhs`).

```
frown :: (Monad m) => [String] -> Terminal -> Lex m a
frown la t = Lex (\cont inp line cur ->
                    fail ("\n*** syntax error at "
                          ++ position inp line cur ++ ":\n"
                          ++ context 4 inp cur
                          ++ "* expected: " ++ concat (intersperse ", " la)))
```

The interactive session listed in Fig. 3.9 is a bit longer than usual to illustrate the quality of the error messages.

3.3.4 Error correction

So far we have content ourselves with reporting syntax errors. To a limited extent it is also possible to *correct* errors. Consider the last rule of the following grammar (`Let7.lg`).

```
expr{Expr};
expr{Const n} : Numeral{n};
     {Var s} | Ident{s};
     {Bin e1 op e2} | expr{e1}, Addop{op}, expr{e2};
     {Bin e1 op e2} | expr{e1}, Mulop{op}, expr{e2};
     {Let d e} | "let", decl{d}, "in", expr{e};
     {e} | "(", expr{e}, ")";
     {e} | "(", expr{e}, insert ")";
```

The symbol *insert* `")"` instructs Frown to automatically insert a `")"` token *if parsing would otherwise fail.* The special symbol *insert* `")"` can be seen as being defined by the ϵ-production *insert* `")"` : ;. The difference to an 'ordinary' user-defined ϵ-production is that the rule is only applied if every other action would fail.

```
Let ≫ run expr "let\n    a = 4 * [7 + 1 - 1)\n in a * a" :: IO Expr

Program error: user error (
*** lexical error at (line 2, column 13):
    a = 4 * [7 + 1 - 1)
            ^
 in a * a
<end of input>
)
Let ≫ run expr "let\n    a = 4 * (7 + 1 - 1\n in a * a" :: IO Expr

Program error: user error (
*** syntax error at (line 3, column 3):
 in a * a
  ^
<end of input>
* expected: + or -, * or /, ))
Let ≫ run expr "let\n    a = 4 * (7 + 1 - 1)\n a * a" :: IO Expr

Program error: user error (
*** syntax error at (line 3, column 2):
 a * a
 ^
<end of input>
* expected: + or -, * or /, in)
Let ≫ run expr "\n    a = 4 * (7 + 1 - 1)\n in a * a" :: IO Expr

Program error: user error (
*** syntax error at (line 2, column 7):
    a = 4 * (7 + 1 - 1)
      ^
 in a * a
<end of input>
* expected: + or -, * or /, <end of input>)
Let ≫ run expr "let\n    a = 4 * (7 + - 1)\n in a * a" :: IO Expr

Program error: user error (
*** syntax error at (line 2, column 18):
    a = 4 * (7 + - 1)
                 ^
 in a * a
<end of input>
* expected: <numeral>, <identifier>, let, ()
Let ≫ run expr "let\n    a = 4  (7 + 1 - 1)\n in a * a" :: IO Expr

Program error: user error (
*** syntax error at (line 2, column 12):
    a = 4  (7 + 1 - 1)
           ^
 in a * a
<end of input>
* expected: + or -, * or /, in)
```

Figure 3.9: A session full of syntax errors.

The following session shows the error correction in action.

```
Let ≫ run expr "4 * (7 + 1) - 1" :: IO Expr
Bin (Bin (Const 4) Times (Bin (Const 7) Plus (Const 1))) Minus (Const 1)
Let ≫ run expr "let\n     a = 4 * (7 + 1) - 1\n in a * a" :: IO Expr
Let ("a" :=: Bin (Bin (Const 4) Times (Bin (Const 7) Plus (Const 1))) Minus (Const 1)) (Bin (Var "a") Tir
Let ≫ run expr "let\n     a = 4 * (7 + 1 - 1\n in a * a" :: IO Expr
Let ("a" :=: Bin (Const 4) Times (Bin (Bin (Const 7) Plus (Const 1)) Minus (Const 1))) (Bin (Var "a") Tir
```

In the last query the missing parenthesis ')' is inserted just before the keyword '**in**'. This may or may not what the user intended!

It is generally a good idea to notify the user if a token is inserted. This is relatively easy to accomplish using monadic actions (`Let8.lg`). The parsing monad is now *Lex IO*; the monad transformer *Lex* proves its worth.

```
expr :: Lex IO Expr

expr{Expr};
expr{Const n} : Numeral{n};
      {Var s} | Ident{s};
      {Bin e1 op e2} | expr{e1}, Addop{op}, expr{e2};
      {Bin e1 op e2} | expr{e1}, Mulop{op}, expr{e2};
      {Let d e} | "let", decl{d}, "in", expr{e};
      {e} | "(", expr{e}, close{_};

close{()};
close{()} : ")";
      {%insert ")"} | insert ")";

insert :: String → Lex IO ()
insert s = lift (putStrLn ("Warning: " ++ s ++ " inserted"))
```

Let's repeat the last query of the previous session.

```
Let ≫ run expr "let\n     a = 4 * (7 + 1 - 1\n in a * a" :: IO Expr
Warning: ) inserted
Let ("a" :=: Bin (Const 4) Times (Bin (Bin (Const 7) Plus (Const 1)) Minus (Const 1))) (Bin (Var "a") Tir
```

The reader is invited to extend the code so that the current source location is additionally printed (informing the user *where* the token has been inserted).

3.4 Advanced features

3.4.1 Rule schemes

When we define grammars we often find ourselves repeatedly writing similar rules. A common pattern is the *repetition* of symbols. As an example, the following rules define a repetition of t symbols.

```
ts:;
ts : ts, t;
```

As an aside, note that the second rule is intentionally *left-recursive*. LR parsers prefer left to right recursion: the rules above use constant stack space whereas the right-recursive variant requires space linear in the length of the input.

Now, Frown allows to capture recurring patterns using so-called *rule schemes*. Here is the scheme for a repetition of symbols (of arity 0).

$$\begin{aligned}&\mathit{many}\ x \leftarrow x;\\ &\mathit{many}\ x{:};\\ &\qquad\quad \mid \mathit{many}\ x, x;\end{aligned}$$

The first line contains *many*'s type signature: it simply says that neither *many* nor *many*'s argument x possess attributes. Given this scheme we can simply write *many* t for a repetition of t symbols.

The rule for repetition becomes more interesting if the argument possesses an attribute (is of arity 1). In this case, *many* returns a list of semantic values.

$$\begin{aligned}&\mathit{many}\ x\{[a]\} \leftarrow x\{a\};\\ &\mathit{many}\ x\{[\,]\} \leftarrow;\\ &\mathit{many}\ x\{as + [a]\} \leftarrow \mathit{many}\ as\{as\}; x\{a\};\end{aligned}$$

(The use of list concatenation '++' in the second rule incurs a runtime penalty which we will cure later.) The first line contains again the type signature, which we may read as a conditional clause: if x has one attribute of type a, then *many* x has one attribute of type $[a]$. This schemes comes in handy if we extend our little expression language by applications and abstractions (we assume that the abstract syntax has been extended suitably; *aexpr* denotes atomic expressions).

$$\begin{aligned}&\mathit{expr}\{\mathit{App}\ e\ es\} : \mathit{aexpr}\{e\}, \mathit{many}\ \mathit{aexpr}\{es\};\\ &\mathit{expr}\{\mathit{Abs}\ (i : is)\ e\} : \texttt{"\\\\"}, \mathit{Ident}\{i\}, \mathit{many}\ (\mathit{Ident}\{\})\{is\}, \texttt{"."}, \mathit{expr}\{e\};\end{aligned}$$

Note that if we pass terminal symbols as arguments to rule schemes they must be written with (empty) curly braces—Frown can only identify terminal symbols, ie patterns, if they have exactly the same syntactic form as in the terminal declaration. Think of '{ }' as a placeholder.

In the above definition of *many* we have used list concatenation to append an element to a list. The following improved definition does away with this linear-time operation employing Hughes' efficient sequence type [3].

$$\begin{aligned}&\mathit{many}\ x\{[a]\} \leftarrow x\{a\};\\ &\mathit{many}\ x\{s\ [\,]\} : \mathit{many}'\ x\{s\};\\[1em] &\mathit{many}'\ x\{[a] \rightarrow [a]\} \leftarrow x\{a\};\\ &\mathit{many}'\ x\{\lambda as \rightarrow as\}{:};\\ &\qquad\quad \{\lambda as \rightarrow s\ (a : as)\} \mid \mathit{many}'\ x\{s\}, x\{a\};\end{aligned}$$

These schemata are predefined in Frown. There is a caveat, however: the singleton production *many* x : *many*$'$ x may introduce a shift/reduce conflict, see Sec. 4.3.

Actually, both the *many* scheme with no attributes and the scheme above with one attribute are predefined. In general, it is possible to use the same name for schemes and nonterminals of different arity. The only restriction is that the arity of the scheme must determine the arity of its arguments.

Another useful variation of *many* is *sepBy* x *sep* which denotes a list of x symbols separated by *sep* symbols (*sepBy* and *sepBy1* are predefined, as well).

$$\begin{aligned}&\mathit{sepBy}\ x\ \mathit{sep}\{[a]\} \leftarrow x\{a\}, \mathit{sep};\\ &\mathit{sepBy}\ x\ \mathit{sep}\{[\,]\}{:};\\ &\qquad\qquad \{as\} \mid \mathit{sepBy1}\ x\ \mathit{sep}\{as\};\\[1em] &\mathit{sepBy1}\ x\ \mathit{sep}\{[a]\} \leftarrow x\{a\}, \mathit{sep};\\ &\mathit{sepBy1}\ x\ \mathit{sep}\{[a]\} : x\{a\};\\ &\qquad\qquad \{as + [a]\} \mid \mathit{sepBy1}\ x\ \mathit{sep}\{as\}, \mathit{sep}, x\{a\};\end{aligned}$$

This scheme is useful for adding tuples to our expression language.

$expr\{\,Tuple\ es\,\}$: `"("`, $sepBy\ expr$ `","`$\{\,es\,\}$, `")"`;

For a complete list of predefined schemes see Sec. 5.3.

3.4.2 A second look at terminal symbols

The terminal symbols of a grammar are given by Haskell *patterns*. Up to now we have seen only simple patterns. Patterns, however, may also be nested or even overlapping. In the latter case, one should be careful to list specific patterns before general ones in a *Terminal* declaration (Frown preserves the relative ordering of patterns when generating **case** expressions). Here is a simple example.

```
Terminal = Ident "if" as "if"
         | Ident "then" as "then"
         | Ident "else" as "else"
         | Ident{String};
         | λ ...
```

Note that keywords are declared just by listing them before the general pattern for identifiers.

Alternatively, terminal symbols can be specifed using so-called *guards*, Boolean functions of type *Terminal* $\rightarrow$ *Bool*. Guards are most useful for defining character classes as in the following example.

```
Terminal = guard{isAlpha} as "alpha"
         | λ ...
```

A guard is introduced by the keyword *guard*, followed by its Haskell definition, followed by the mandatory shortcut. The shortcut can then be used as a terminal symbol *of arity* 1: its attribute of type *Terminal* is the very input symbol that matched the guard.

```
ident{String};
ident{c : cs} : "alpha"{c}, many "alpha"{cs};
```

Using guards one can quite easily define character-based grammars that include lexical syntax (that is, whose parsers combine lexing and parsing). Fig. 3.10 lists a variant of the desktop calculator that works without a separate lexer. Note that the type *Terminal* must be defined in the Haskell section. The reader may wish to extend the grammar so that two tokens can be separated by white space.

3.4.3 Look-ahead

⟨**To do:** type grammar.⟩

3.4.4 Debugging and tracing

⟨**To do:** `--prefix` und `--suffix`.⟩

```
module Calc where
import Result
import Char

type Terminal = Char

%{

Terminal = guard{isDigit}as "digit"
         | '+'
         | '*'
         | '('
         | ')';

Nonterminal = expr{Integer}
            | term{Integer}
            | factor{Integer}
            | numeral{Integer};

expr{v1 + v2} : expr{v1}, '+', term{v2};
    {e} | term{e};
term{v1 * v2} : term{v1}, '*', factor{v2};
    {e} | factor{e};
factor{n} : numeral{n};
      {e} | '(', expr{e}, ')';
numeral{encode c} : "digit"{c};
       {n * 10 + encode c} | numeral{n}, "digit"{c};

}%

encode c = toInteger (fromEnum c - fromEnum '0')

frown _ = fail "syntax error"
```

Figure 3.10: A variant of the desktop calculator that includes lexical syntax (`VarCalc.lhs`).

⟨**To do:** `--debug` und `--pagewidth`.⟩

```
module Paren
where

%{

Terminal = '(' | ')';

paren{IO ()};
paren{reduce "p : ;"}
    :;
paren{do t1; shift '('; t2; shift ')'; reduce "p : p, '(', p, ')';"}
    : paren{t1}, '(', paren{t2}, ')';

}%

frown _ = fail "*** syntax error"

shift :: Char → IO ()
shift c = putStrLn ("shift " ++ show c)

reduce :: String → IO ()
reduce p = putStrLn ("reduce by " ++ p)
```

3.4.5 Output formats and optimizations

⟨**To do:** optimizations (`--optimize`).⟩
⟨**To do:** which format benefits from GHC extensions (`--ghc`)?⟩
⟨**To do:** `NOINLINE` pragmas (`--noinline`).⟩
⟨**To do:** `--signature`.⟩

Chapter 4

Tips and tricks

4.1 Irrefutable patterns

Irrefutable patterns on the RHS (`VarParen.lg`):

```
module VarParen where
import Result

newtype Tree = Node [Tree]
               deriving (Show)

%{

Terminal = '(' | ')';

Nonterminal = paren{Tree};

paren{Node []}:;
    {Node (x : xs)} | paren{x}, '(', paren{Node xs}, ')';

}%

frown ts = fail "syntax error"
```

4.2 Inherited attributes

Shows how to simulate inherited attributes: *expr* has type $Integer \rightarrow (Tree\ Integer, Integer)$, it takes the global minimum to the rep-min tree (with all elements replaced by the minimum) and the local minimum (`RepMin.lg`).

```
module RepMin where

data Tree a = Leaf a | Fork (Tree a) (Tree a)
              deriving (Show)

data Terminal = Num Integer | LPar | RPar

%{

Terminal = Num{Integer}
         | LPar as "("
         | RPar as ")";

Nonterminal = *start{Tree Integer}
            | expr{Integer → (Tree Integer, Integer)};

start{let (t, m) = f m in t}
  : expr{f};

expr{λm → (Leaf m, i)}
  : Num{i};
expr{λm → let {(tl, ml) = l m
              ;(tr, mr) = r m}
          in (Fork tl tr, ml `min` mr)}
  : expr{l}, "(", expr{r}, ")";

}%

frown ts = fail "syntax error"
```

!avoid layout-sensitive code!

4.3 Dealing with conflicts

$many'\ x : many\ x;$

4.4 Multiple attributes

Chapter 5

Reference manual

5.1 Lexical syntax of **Frown**

⟨**To do:** that of Haskell including comments.⟩
⟨**To do:** Literate grammar file (Bird tracks)⟩.

5.2 Syntax of **Frown**

Grammar file.

file : *many* `"not special"`,
    `"%{"`,
    many decl;
    `"}%"`,
    many `"not special"`;

Note that `"not special"` matches every token except the special curly braces `"%{"` and `"}%"`.
Declaration.

decl : *terminals*;
  | *nonterminals*;
  | *fixity*;
  | *signature*;
  | *productions*;

Terminal declaration.

terminals : `"Terminal"`, `"="`, *sepBy term* `"|"`, `";"`;

term : *opt* `"*"`, *assoc*, *terminal*;
  | *opt* `"*"`, *assoc*, *literal*, `"="`, *terminal*;   -- deprecated
  | *opt* `"*"`, *assoc*, *terminal*, `"as"`, *literal*;
  | *opt* `"*"`, *assoc*, `"guard"`, *haskell*, `"as"`, *literal*;

assoc:;
  | `"left"`, *Numeral*;
  | `"right"`, *Numeral*;
  | `"nonassoc"`, *Numeral*;

Nonterminal declaration.

nonterminals : `"Nonterminal"`, `"="`, *sepBy nonterm* `"|"`, `";"`;

nonterm : *opt* `"*"`, *nonterminal*;

Fixity declaration.

> *fixity* : "left", *Numeral*, *terminal*, ";";
> | "right", *Numeral*, *terminal*, ";";
> | "nonassoc", *Numeral*, *terminal*, ";";

Type signature.

> *signature* : "::", *nonterminal*, *premise*, ";"; -- deprecated
> | *nonterminal*, *premise*, ";";
> | "::", "*", *nonterminal*, ";"; -- deprecated
> | "*", *nonterminal*, ";";
>
> *premise*:;
> | "<-", *sepBy1 nonterminal* ",";

Productions.

> *productions* : *nonterminal*, ":", *sepBy symbol* ",", ";", *alts*;
>
> *alts*:;
> | *attributes*, "|", *sepBy symbol* ",", ";", *alts*;
>
> *symbol* : "insert", *terminal*;
> | "delete", *terminal*;
> | "prec", *terminal*;
> | *terminal*;
> | *nonterminal*;

Nonterminal symbols (*expr0* is a variant of *expr* lacking the embedded Haskell production).

> *nonterminal* : *expr0*, *attributes*;
>
> *expr0* : *Varid*, *many aexpr0*;
>
> *aexpr0* : *Varid*;
> | *Conid*;
> | *literal*;
> | "(", *sepBy expr* ",", ")";
> | "[", *sepBy expr* ",", "]";
>
> *expr* : *aexpr*;
> | *Varid*, *many1 aexpr*;
> | *Conid*, *many1 aexpr*;
>
> *aexpr* : *Varid*;
> | *Conid*;
> | *literal*;
> | "(", *sepBy expr* ",", ")";
> | "[", *sepBy expr* ",", "]";
> | *haskell*; -- embedded Haskell

Terminal symbols.

```
terminal : pat;
         | literal, haskell, attributes;        -- shortcut

pat : apat;
    | Conid, many1 apat;

apat : Conid;
     | literal;        -- either literal or shortcut
     | "(", sepBy pat ",", ")";
     | "[", sepBy pat ",", "]";
     | haskell;

literal : String;
        | Numeral;
        | Char;
```

Embedded Haskell (types, patterns, and expressions).

```
attributes:;
          | haskell, attributes;

haskell : "{", many hs, "}";

hs : "not brace";
   | "{", many hs, "}";
```

Note that `"not brace"` matches every token except the curly braces `"{"` and `"}"`.

5.3 Predefined schemes

Note that the predefined rules are left-recursive and ‘run’ using constant stack space. Also note that we define rules for arity zero and arity one (the arity specifies the number of attributes/semantic values). The primed versions of the rules work on Hughes’s efficient sequence type (a sequence of a’s is represented by a function of type $[a] \to [a]$).

5.3.1 Optional elements

Arity zero.

```
opt x ← x;
opt x:;
     | x;
```

Arity one.

```
opt x{Maybe a} ← x{a};
opt x{Nothing}:;
     {Just a} | x{a};
```

5.3.2 Repetition of elements

Arity zero.

$$\begin{array}{l} \textit{many}\ x \leftarrow x; \\ \textit{many}\ x :; \\ \qquad\quad \mid \textit{many}\ x, x; \end{array}$$

$$\begin{array}{l} \textit{many1}\ x \leftarrow x; \\ \textit{many1}\ x : x, \textit{many}\ x; \end{array}$$

Arity one.

$$\begin{array}{l} \textit{many}\ x\{[a]\} \leftarrow x\{a\}; \\ \textit{many}\ x\{s\ []\} : \textit{many}'\ x\{s\}; \end{array}$$

$$\begin{array}{l} \textit{many}'\ x\{[a] \rightarrow [a]\} \leftarrow x\{a\}; \\ \textit{many}'\ x\{\lambda as \rightarrow as\} :; \\ \qquad\quad \{\lambda as \rightarrow s\ (a : as)\} \mid \textit{many}'\ x\{s\}, x\{a\}; \end{array}$$

$$\begin{array}{l} \textit{many1}\ x\{[a]\} \leftarrow x\{a\}; \\ \textit{many1}\ x\{a : as\} : x\{a\}, \textit{many}\ x\{as\}; \end{array}$$

5.3.3 Repetition of elements separated by a second element

Arity zero.

$$\begin{array}{l} \textit{sepBy}\ x\ \textit{sep} \leftarrow x, \textit{sep}; \\ \textit{sepBy}\ x\ \textit{sep} :; \\ \qquad\qquad \mid \textit{sepBy1}\ x\ \textit{sep}; \end{array}$$

$$\begin{array}{l} \textit{sepBy1}\ x\ \textit{sep} \leftarrow x, \textit{sep}; \\ \textit{sepBy1}\ x\ \textit{sep} : x; \\ \qquad\qquad\quad \mid \textit{sepBy1}\ x\ \textit{sep}, \textit{sep}, x; \end{array}$$

Arity one.

$$\begin{array}{l} \textit{sepBy}\ x\ \textit{sep}\{[a]\} \leftarrow x\{a\}, \textit{sep}; \\ \textit{sepBy}\ x\ \textit{sep}\{[]\} :; \\ \qquad\qquad \{as\} \mid \textit{sepBy1}\ x\ \textit{sep}\{as\}; \end{array}$$

$$\begin{array}{l} \textit{sepBy1}\ x\ \textit{sep}\{[a]\} \leftarrow x\{a\}, \textit{sep}; \\ \textit{sepBy1}\ x\ \textit{sep}\{s\ []\} : \textit{sepBy1}'\ x\ \textit{sep}\{s\}; \end{array}$$

$$\begin{array}{l} \textit{sepBy1}'\ x\ \textit{sep}\{[a] \rightarrow [a]\} \leftarrow x\{a\}, \textit{sep}; \\ \textit{sepBy1}'\ x\ \textit{sep} \\ \qquad \{\lambda as \rightarrow a : as\} : x\{a\}; \\ \qquad \{\lambda as \rightarrow s\ (a : as)\} \mid \textit{sepBy1}'\ x\ \textit{sep}\{s\}, \textit{sep}, x\{a\}; \end{array}$$

TODO: also versions where *sep* has arity one.

5.3.4 Repetition of possibly empty elements separated by a second element

⟨**To do:** better name.⟩

Arity zero.

$$
\begin{aligned}
&\mathit{optSepBy}\ x\ \mathit{sep} \leftarrow x, \mathit{sep};\\
&\mathit{optSepBy}\ x\ \mathit{sep}{:};\\
&\qquad\qquad\qquad \mid x;\\
&\qquad\qquad\qquad \mid \mathit{optSepBy}\ x\ \mathit{sep}, \mathit{sep};\\
&\qquad\qquad\qquad \mid \mathit{optSepBy}\ x\ \mathit{sep}, \mathit{sep}, x;
\end{aligned}
$$

Arity one.

$$
\begin{aligned}
&\mathit{optSepBy}\ x\ \mathit{sep}\{[a]\} \leftarrow x\{a\}, \mathit{sep};\\
&\mathit{optSepBy}\ x\ \mathit{sep}\{s\ []\} : \mathit{optSepBy}'\ x\ \mathit{sep}\{s\};\\
&\\
&\mathit{optSepBy}'\ x\ \mathit{sep}\{[a] \to [a]\} \leftarrow x\{a\}, \mathit{sep};\\
&\mathit{optSepBy}'\ x\ \mathit{sep}\\
&\quad \{\lambda as \to as\}{:};\\
&\quad \{\lambda as \to a : as\} \mid x\{a\};\\
&\quad \{\lambda as \to s\ as\} \mid \mathit{optSepBy}'\ x\ \mathit{sep}\{s\}, \mathit{sep};\\
&\quad \{\lambda as \to s\ (a : as)\} \mid \mathit{optSepBy}'\ x\ \mathit{sep}\{s\}, \mathit{sep}, x\{a\};
\end{aligned}
$$

5.4 Output formats

⟨**To do:** Used type names: *Result* and *Terminal*.⟩

⟨**To do:** Used function names: *frown*. For each start symbol a parser.⟩

The `code=standard` format is due to Doaitse Swierstra [1].

The `code=stackless` format is due to Ross Paterson [2].

The `code=gvstack` format is also due to Ross Paterson.

5.5 Invocation and options

`Usage: frown [option ...] file.g ...`

`-b` **or** `--backtrack`
: generate a backtracking parser (see Sec. 3.2.5)

`-cc`, `-ccompact` **or** `--code=compact`
: (see Sec. 3.4.5 and 5.4)

`-cg`, `-cgvstack` **or** `--code=gvstack`
: (see Sec. 3.4.5 and 5.4)

`-cs`, `-cstackless` **or** `--code=stackless`
: (see Sec. 3.4.5 and 5.4)

`-cstandard` **or** `--code=standard`
: (see Sec. 3.4.5 and 5.4)

`--copying`
: display details of copying

`-d` **or** `--debug`
: emit debugging information (see Sec. 3.4.4)

`-e` **or** `--expected`
: pass a list of expected terminals to '*frown*' (see Sec. 3.3.3)

```
module Paren where
import Result

 {- frown :-( -}

data Stack = Empty | T_1 State Stack

data State = S_1 | S_2 | S_3 | S_4 | S_5 | S_6

data Nonterminal = Paren

paren tr = parse_1 tr Empty >>= (\Paren -> return ())

parse_1 ts st = reduce_2 ts S_1 st

parse_2 tr@[] st = parse_3 tr (T_1 S_2 st)
parse_2 ('(' : tr) st = parse_5 tr (T_1 S_2 st)
parse_2 ts st = frown ts

parse_3 ts st = reduce_1 ts st

parse_4 ('(' : tr) st = parse_5 tr (T_1 S_4 st)
parse_4 (')' : tr) st = parse_6 tr (T_1 S_4 st)
parse_4 ts st = frown ts

parse_5 ts st = reduce_2 ts S_5 st

parse_6 ts st = reduce_3 ts st

reduce_1 ts (T_1 _ (T_1 s st)) = return Paren

reduce_2 ts s st = goto_5 s ts (T_1 s st)

reduce_3 ts (T_1 _ (T_1 _ (T_1 _ (T_1 s st))))
                                        = goto_5 s ts (T_1 s st)

goto_5 S_1 = parse_2
goto_5 S_5 = parse_4

 {- )-: frown -}

frown _ = fail "syntax error"
```

Figure 5.1: `frown --code=compact Paren.g`.

```
module Paren where
import Result

 {- frown :-( -}

paren tr = state_1 (λ_ → return ()) tr

state_1 k_1_0 ts = let {goto_paren = state_2 k_1_0 (reduce_3 goto_paren)}
                   in reduce_2 goto_paren ts

state_2 k_1_1 k_3_1 ts = case ts of {tr@[] → state_3 k_1_1 tr;
                                     '(' : tr → state_5 k_3_1 tr;
                                     _ → frown ts}

state_3 k_1_2 ts = k_1_2 ts

state_4 k_3_1 k_3_3 ts = case ts of {'(' : tr → state_5 k_3_1 tr;
                                     ')' : tr → state_6 k_3_3 tr;
                                     _ → frown ts}

state_5 k_3_2 ts = let {goto_paren = state_4 (reduce_3 goto_paren) k_3_2}
                   in reduce_2 goto_paren ts

state_6 k_3_4 ts = k_3_4 ts

reduce_2 g ts = g ts

reduce_3 g ts = g ts

 {- )-: frown -}

frown _ = fail "syntax error"
```

Figure 5.2: `frown --code=stackless Paren.g`.

```
module Paren where
import Result

 {- frown :-( -}

data Nonterminal = Paren′ | Paren

type Parser = [Terminal] → Result Nonterminal

type VStack vs v = ((vs, Nonterminal → Parser), v)

paren tr = state_1 () tr ≫= (λParen′ → return ())

state_1 :: vs → Parser
state_1 = state action_1 goto_1
action_1 t = reduce_2
goto_1 Paren = goto state_2 ()


state_2 :: VStack vs () → Parser
state_2 = state action_2 ⊥
action_2 t = case t of { '(' → shift state_5 ();
                         '$' → shift state_3 ();
                         _ → error }

state_3 :: VStack (VStack vs ()) () → Parser
state_3 = state action_3 ⊥
action_3 t = reduce_1

state_4 :: VStack (VStack (VStack vs ()) ()) () → Parser
state_4 = state action_4 ⊥
action_4 t = case t of { '(' → shift state_5 ();
                         ')' → shift state_6 ();
                         _ → error }

state_5 :: VStack (VStack vs ()) () → Parser
state_5 = state action_5 goto_5
action_5 t = reduce_2
goto_5 Paren = goto state_4 ()

state_6 :: VStack (VStack (VStack (VStack vs ()) ()) ()) () → Parser
state_6 = state action_6 ⊥
action_6 t = reduce_3


reduce_1 (((((_, g), ()), _), ()), _) ts
                            = accept Paren′ ts

reduce_2 (_, g) ts = g Paren ts

reduce_3 (((((((((_, g), ()), _), ()), _), ()), _), ()), _) ts
                            = g Paren ts

state action goto vs ts = let {gs = (vs, g); g v = goto v gs} in action (head ts) gs ts

shift state v vs ts = state (vs, v) (tail ts)

shift′ state v vs ts = state (vs, v) ts

accept v _ = return v
```

`-g` **or** `--ghc`
: use GHC extensions (see Sec. 3.4.5)

`-h`, `-?` **or** `--help`

`-i` **or** `--info`
: put additional information into generated file (see Sec. 3.4.4)

`-k[nat]` **or** `--lookahead[=nat]`
: use k tokens of look-ahead (see Sec. 3.4.3)

`-l` **or** `--lexer`
: use a monadic lexer ($get :: M\ \ Terminal$) (see Sec. 3.3.1)

`-n` **or** `--noinline`
: generate `NOINLINE` pragmas (see Sec. 3.4.5)

`-O` **or** `--optimize`
: optimize parser (see Sec. 3.4.5)

`-p[nat]` **or** `--pagewidth[=nat]`
: use the specified pagewidth for pretty printing (see Sec. 3.4.4)

`--prefix[=string]`
: use prefix for Frown-generated variables (see Sec. 3.4.4)

`-sm`, `-smono` **or** `--signature=mono`
: add monomorphic type signatures (see Sec. 3.4.5)

`-sp`, `-spoly` **or** `--signature=poly`
: add polymorphic type signatures (see Sec. 3.4.5)

`--suffix[=string]`
: use suffix for frown generated variables (see Sec. 3.4.4)

`-t` **or** `--trace`
: insert calls to tracing routines ('*shift*', '*reduce*' and '*accept*') (see Sec. 3.4.4)

`-v` **or** `--verbose`
: be verbose

`--version`
: print version information

`--warranty`
: display details of warranty

Bibliography

[1] Luc Duponcheel and Doaitse Swierstra. A functional program for generating efficient functional LALR(1) parsers, September 2000. unpublished note.

[2] Ralf Hinze and Ross Paterson. Derivation of a typed functional LR parser, 2005. in submission.

[3] R. John Muir Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, March 1986.