Chapter 1

The Core language

All our implementations take some program written in a simple *Core language* and execute it. The Core language is quite impoverished, and one would not want to write a large program in it. Nevertheless it has been carefully chosen so that it is possible to translate programs in a rich functional language (such as Miranda) into the Core language without losing expressiveness or efficiency. The Core language thus serves as a clean interface between the 'front end' of the compiler, which is concerned with high-level language constructs, and the 'back end', which is concerned with implementing the Core language in various different ways.

We begin with an informal introduction to the Core language (Section 1.1). Following this, we define the Core language more formally, by giving:

- Its syntax (Section 1.2).
- Miranda data types `coreProgram` and `coreExpr` for Core-language programs and expressions respectively (Section 1.3). These will serve as the input data types for the compilers we build.
- Definitions for a small *standard prelude* of Core-language functions, which will be made available in any Core program (Section 1.4).
- A pretty-printer, which transforms a Core-language program into a character string which, when printed, is a formatted version of the program (Section 1.5).
- A parser, which parses a character string to produce a Core-language program (Section 1.6).

This chapter has a second purpose: it introduces and uses many of the features of Miranda which we will use throughout the book before we get involved with any of our functional-language implementations.

1.1 An overview of the Core language

Here is an example Core program[1], which evaluates to 42:

```
main = double 21
double x = x + x
```

A Core program consists of a set of *supercombinator definitions*, including a distinguished one, `main`. To execute the program, we evaluate `main`. Supercombinators can define functions, such as the definition of `double`. `double` is a function of one argument, `x`, which returns twice its argument. The program looks quite similar to the top level of a Miranda script, except that no pattern matching is permitted for function arguments. Pattern matching is performed by a separate Core language construct, the `case` expression, which is discussed below. Each supercombinator is defined by a single equation whose arguments are all simple variables.

Notice that not all supercombinators have arguments. Some, such as `main`, take no arguments. Supercombinators with no arguments are also called *constant applicative forms* or CAFs and, as we shall see, often require special treatment in an implementation.

1.1.1 Local definitions

Supercombinators can have local definitions, using the `let` construct of the Core language:

```
main = quadruple 20 ;
quadruple x = let twice_x = x+x
              in twice_x + twice_x
```

Here `twice_x` is defined locally within the body of `quadruple` to be `x+x`, and `quadruple` returns `twice_x + twice_x`. Like Miranda `where` clauses, local definitions are useful both to name intermediate values, and to save recomputing the same value twice; the programmer can reasonably hope that only two additions are performed by `quadruple`.

A `let` expression is *non-recursive*. For recursive definitions, the Core language uses the `letrec` construct, which is exactly like `let` except that its definitions can be recursive. For example:

```
infinite n = letrec ns = cons n ns
             in ns
```

The reason that we distinguish `let` from `letrec` in the Core language (rather than providing only `letrec`) is that `let` is a bit simpler to implement than `letrec`, and we may get slightly better code.

`let` and `letrec` are similar to the Miranda `where` clause, but there are a number of important differences:

- The `where` clause always defines a recursive scope. There is no non-recursive form.

[1]We use typewriter fount for Core programs, but without the initial > sign which distinguishes executable Miranda code.

- A `where` clause can be used to define local functions, and to perform pattern matching:

```
... where f x = x+y
          (p,q) = zip xs ys
```

  Neither of these facilities is provided by the Core language `let` and `letrec` expressions. Functions can only be defined at the top level, as supercombinators, and pattern matching is done only by `case` expressions.

  In short, *the left-hand side of a* `let` *or* `letrec` *binding must be a simple variable.*

- The `let`/`letrec` construct is an *expression*. It is therefore quite legal to write (for example):

```
quad_plus_one x = 1 + (let tx = x+x in tx+tx)
```

  In contrast a `where` clause in Miranda can only be attached to *definitions*. (One reason for this is that it allows the definitions in a Miranda `where` clause to range over several guarded right-hand sides.)

1.1.2 Lambda abstractions

Functions are usually expressed in the Core language using top-level supercombinator definitions, and for most of the book this is the *only* way in which functions can be denoted. However, it is sometimes convenient to be able to denote functions using explicit *lambda abstractions*, and the Core language provides a construct to do so. For example, in the program

```
double_list xs = map (\ x. 2*x) xs
```

the lambda abstraction `(\ x. 2*x)` denotes the function which doubles its argument.

It is possible to transform a program involving explicit lambda abstractions into an equivalent one which uses only top-level supercombinator definitions. This process is called *lambda lifting*, and is discussed in detail in Chapter 6. Throughout the other chapters we assume that this lambda lifting process has been done, so they make no use of explicit lambda abstractions.

The final major construct in the Core language is the `case` expression, which expresses pattern matching. There are several ways of handling pattern matching, so we begin with a review of structured data types.

1.1.3 Structured data types

A universal feature of all modern functional programming languages is the provision of *structured types*, often called *algebraic data types*. For example, here are a few algebraic type definitions, written in Miranda:

```
colour ::= Red | Green | Blue
```

```
complex ::= Rect num num | Polar num num

numPair ::= MkNumPair num num

tree * ::= Leaf * | Branch (tree *) (tree *)
```

Each definition introduces a new *type* (such as `colour`), together with one or more *constructors* (such as `Red`, `Green`). They can be read as follows: 'A value of type `colour` is either `Red` or `Green` or `Blue`', and 'A `complex` is either a `Rect` containing two `num`s, or a `Polar` containing two `num`s'.

The type `tree` is an example of a *parameterised* algebraic data type; the type `tree` is parameterised with respect to the type variable `*`. It should be read as follows: 'a `tree` of `*`'s is either a `Leaf` containing a `*`, or a `Branch` containing two `tree` of `*`'s'. Any particular tree must have leaves of uniform type; for example, the type `tree num` is a tree with `num`s at its leaves, and the type `tree colour` is a tree with `colour`s at its leaves.

Structured values are *built* with these constructors; for example the following expressions denote structured values:

```
Green
Rect 3 4
Branch (Leaf num) (Leaf num)
```

Structured values are *taken apart* using *pattern matching*. For example:

```
isRed Red   = True
isRed Green = False
isRed Blue  = False

first (MkNumPair n1 n2) = n1

depth (Leaf n)       = 0
depth (Branch t1 t2) = 1 + max (depth t1) (depth t2)
```

Several data types usually thought of as 'built in' are just special cases of structured types. For example, booleans are a structured type: they can be defined by the algebraic data type declaration

```
bool ::= False | True
```

Apart from their special syntax, the lists and tuples provided by Miranda are further examples of structured types. If we use `Cons` and `Nil` as constructors rather than the special syntax of `:` and `[]`, we could define lists like this:

```
list * ::= Nil | Cons * (list *)
```

Chapter 4 of [Peyton Jones 1987] gives a fuller discussion of structured types.

The question arises, therefore: *how are we to represent and manipulate structured types in our small Core language?* In particular, our goal is to avoid having data type declarations in the Core language altogether. The approach we take breaks into two parts:

- Use a simple, uniform representation for constructors.
- Transform pattern matching into simple `case` expressions.

1.1.4 Representing constructors

Instead of allowing user-defined constructors such as `Red` and `Branch` in our Core language, we provide a *single* family of constructors

$$\texttt{Pack}\{\mathit{tag}, \mathit{arity}\}$$

Here, *tag* is an integer which uniquely identifies the constructor, and *arity* tells how many arguments it takes. For example, we could represent the constructors of `colour`, `complex`, `tree` and `numPair` as follows:

```
Red        =  Pack{1,0}
Green      =  Pack{2,0}
Blue       =  Pack{3,0}

Rect       =  Pack{4,2}
Polar      =  Pack{5,2}

Leaf       =  Pack{6,1}
Branch     =  Pack{7,2}

MkNumPair  =  Pack{8,2}
```

So in the Core language one writes

```
Pack{7,2} (Pack{6,1} 3) (Pack{6,1} 4)
```

instead of

```
Branch (Leaf 3) (Leaf 4)
```

The tag is required so that objects built with different constructors can be distinguished from one another. In a well-typed program, objects of different type will never need to be distinguished at run-time, so tags only need to be unique *within a data type.* Hence, we can start the tag at 1 afresh for each new data type, giving the following representation:

```
Red        =  Pack{1,0}
Green      =  Pack{2,0}
Blue       =  Pack{3,0}

Rect       =  Pack{1,2}
Polar      =  Pack{2,2}

Leaf       =  Pack{1,1}
Branch     =  Pack{2,2}

MkNumPair  =  Pack{1,2}
```

1.1.5 `case` expressions

In general, the pattern matching allowed by modern functional programming languages can be rather complex, with multiple nested patterns, overlapping patterns, guards and so on. For the Core language, we eliminate these complications by outlawing all complex forms of pattern matching! We do this by providing only `case` *expressions* in the Core language. Their formal syntax is given in Section 1.2, but here are some examples:

```
isRed c = case c of
                <1> -> True ;
                <2> -> False ;
                <3> -> False

depth t = case t of
                <1> n -> 0 ;
                <2> t1 t2 -> 1 + max (depth t1) (depth t2)
```

The important thing about `case` expressions is that each alternative consists only of a tag followed by a number of variables (which should be the same as the arity of the constructor). No nested patterns are allowed.

`case` expressions have a very simple operational interpretation, rather like a multi-way jump: evaluate the expression to be analysed, get the tag of the constructor it is built with and evaluate the appropriate alternative.

1.2 Syntax of the Core language

Figure 1.1 gives the syntax for the Core language. The grammar allows infix binary operators, but (for brevity) is not explicit about their precedence. Instead we give the following table of precedences, where a higher precedence means tighter binding:

<table>
<tr><th>Precedence</th><th>Associativity</th><th>Operator</th></tr>
<tr><td>6</td><td>Left</td><td>Application</td></tr>
<tr><td>5</td><td>Right</td><td>*</td></tr>
<tr><td></td><td>None</td><td>/</td></tr>
<tr><td>4</td><td>Right</td><td>+</td></tr>
<tr><td></td><td>None</td><td>-</td></tr>
<tr><td>3</td><td>None</td><td>== ~= > >= < <=</td></tr>
<tr><td>2</td><td>Right</td><td>&</td></tr>
<tr><td>1</td><td>Right</td><td>|</td></tr>
</table>

An operator's associativity determines when parentheses may be omitted around repetitions of the operator. For example, `+` is right-associative, so `x+y+z` means the same as `x+(y+z)`. On the other hand, `/` is non-associative, so the expression `x/y/z` is illegal.

There is no special operator symbol for unary negation. Instead, the `negate` function is provided, which behaves syntactically like any normal function. For example:

```
f x = x + (negate x)
```

The boolean negation operator, `not`, is handled in the same way.

1.3 Data types for the Core language

For each of the implementations discussed in this book we will build a compiler and a machine interpreter. The compiler takes a Core program and translates it into a form suitable for execution by the machine interpreter. To do this we need a Miranda data type to represent Core programs, and that is what we will define in this section. In fact we will define a type for Core programs, one for Core expressions and a few other auxiliary types.

The data type of Core-language expression, `expr`, is defined as follows:

```
> module Language where
> import Utils


> data Expr a
>   =  EVar Name                     -- Variables
>    | ENum Int                      -- Numbers
>    | EConstr Int Int               -- Constructor tag arity
>    | EAp (Expr a) (Expr a)         -- Applications
>    | ELet                          -- Let(rec) expressions
>          IsRec                     --   boolean with True = recursive,
>          [(a, Expr a)]             --   Definitions
>          (Expr a)                  --   Body of let(rec)
>    | ECase                         -- Case expression
>          (Expr a)                  --   Expression to scrutinise
>          [Alter a]                 --   Alternatives
>    | ELam [a] (Expr a)             -- Lambda abstractions
>     deriving (Text)
```

<table>
<tr><td>Programs</td><td>program</td><td>$\rightarrow$</td><td>sc_1 ; ... ; sc_n</td><td>$n \geq 1$</td></tr>
<tr><td>Supercombinators</td><td>sc</td><td>$\rightarrow$</td><td>var $var_1 \ldots var_n$ = expr</td><td>$n \geq 0$</td></tr>
<tr><td>Expressions</td><td>expr</td><td>$\rightarrow$</td><td>expr aexpr</td><td>Application</td></tr>
<tr><td></td><td></td><td>$\mid$</td><td>$expr_1$ binop $expr_2$</td><td>Infix binary application</td></tr>
<tr><td></td><td></td><td>$\mid$</td><td><code>let</code> defns <code>in</code> expr</td><td>Local definitions</td></tr>
<tr><td></td><td></td><td>$\mid$</td><td><code>letrec</code> defns <code>in</code> expr</td><td>Local recursive definitions</td></tr>
<tr><td></td><td></td><td>$\mid$</td><td><code>case</code> expr <code>of</code> alts</td><td>Case expression</td></tr>
<tr><td></td><td></td><td>$\mid$</td><td><code>\</code> $var_1 \ldots var_n$ <code>.</code> expr</td><td>Lambda abstraction ($n \geq 1$)</td></tr>
<tr><td></td><td></td><td>$\mid$</td><td>aexpr</td><td>Atomic expression</td></tr>
<tr><td></td><td>aexpr</td><td>$\rightarrow$</td><td>var</td><td>Variable</td></tr>
<tr><td></td><td></td><td>$\mid$</td><td>num</td><td>Number</td></tr>
<tr><td></td><td></td><td>$\mid$</td><td><code>Pack{</code>num<code>,</code>num<code>}</code></td><td>Constructor</td></tr>
<tr><td></td><td></td><td>$\mid$</td><td><code>(</code> expr <code>)</code></td><td>Parenthesised expression</td></tr>
<tr><td>Definitions</td><td>defns</td><td>$\rightarrow$</td><td>$defn_1$; ... ; $defn_n$</td><td>$n \geq 1$</td></tr>
<tr><td></td><td>defn</td><td>$\rightarrow$</td><td>var = expr</td><td></td></tr>
<tr><td>Alternatives</td><td>alts</td><td>$\rightarrow$</td><td>alt_1 ; ... ; alt_n</td><td>$n \geq 1$</td></tr>
<tr><td></td><td>alt</td><td>$\rightarrow$</td><td><code><</code>num<code>></code> $var_1 \ldots var_n$ <code>-></code> expr</td><td>$n \geq 0$</td></tr>
<tr><td>Binary operators</td><td>binop</td><td>$\rightarrow$</td><td>arithop $\mid$ relop $\mid$ boolop</td><td></td></tr>
<tr><td></td><td>arithop</td><td>$\rightarrow$</td><td><code>+</code> $\mid$ <code>-</code> $\mid$ <code>*</code> $\mid$ <code>/</code></td><td>Arithmetic</td></tr>
<tr><td></td><td>relop</td><td>$\rightarrow$</td><td><code><</code> $\mid$ <code><=</code> $\mid$ <code>==</code> $\mid$ <code>~=</code> $\mid$ <code>>=</code> $\mid$ <code>></code></td><td>Comparison</td></tr>
<tr><td></td><td>boolop</td><td>$\rightarrow$</td><td><code>&</code> $\mid$ <code>|</code></td><td>Boolean</td></tr>
<tr><td>Variables</td><td>var</td><td>$\rightarrow$</td><td>alpha $varch_1 \ldots varch_n$</td><td>$n \geq 0$</td></tr>
<tr><td></td><td>alpha</td><td>$\rightarrow$</td><td>an alphabetic character</td><td></td></tr>
<tr><td></td><td>varch</td><td>$\rightarrow$</td><td>alpha $\mid$ digit $\mid$ <code>_</code></td><td></td></tr>
<tr><td>Numbers</td><td>num</td><td>$\rightarrow$</td><td>$digit_1 \ldots digit_n$</td><td>$n \geq 1$</td></tr>
</table>

Figure 1.1: BNF syntax for the Core language

We choose to parameterise the data type of `expr` with respect to its *binders*. A binder is the name used at the binding occurrence of a variable; that is, on the left-hand side of a `let(rec)` definition, or in a lambda abstraction. The declaration can be read 'An `expr` of `*` is either an `EVar` containing a `name`, or ..., or an `ELam` containing a list of values of type `*` and an `expr` of `*`'.

For the most of the book we always use `name` in these binding positions, so we use a *type synonym* to define the type of `coreExpr`, which is the type we will normally use:

```
> type CoreExpr = Expr Name
```

The ability to use types other than `name` in binding positions is only used in Chapter 6.

Apart from this, the data type follows fairly directly from the syntax given in the previous section, except that various superficial differences are discarded. The biggest difference is that infix operators are expressed in prefix form in the data type. For example, the expression

```
x + y
```

is represented by

```
EAp (EAp (EVar "+") (EVar "x")) (EVar "y")
```

Variables are represented by an `EVar` constructor containing the variable's name. A variable's name is represented simply by a list of characters, which we express using another type synonym:

```
> type Name = String
```

Constructors are identified by their arity and tag, as described in Section 1.1.4.

`let` and `letrec` expressions are represented by an `ELet` constructor containing: a flag of type `isRec` to distinguish the recursive case from the non-recursive one; a list of definitions; and the expression which is the body of the `let(rec)`. We choose to represent `isRec` as a boolean variable, and we define the two boolean values as follows:

```
> type IsRec = Bool
> recursive, nonRecursive :: IsRec
> recursive    = True
> nonRecursive = False
```

Each definition is just a pair of the variable name being bound and the expression to which it is bound. We define two useful functions which each take a list of definitions: `bindersOf` picks out the list of variables bound by the definitions, and `rhssOf` (short for 'right-hand sides of') extracts the list of right-hand sides to which they are bound.

```
> bindersOf :: [(a,b)] -> [a]
> bindersOf defns =  [name | (name, rhs) <- defns]
```

```
> rhssOf        :: [(a,b)] -> [b]
> rhssOf defns  =  [rhs  | (name, rhs) <- defns]
```

`case` expressions have an expression to analyse, and a list of alternatives. Each alternative contains a tag, a list of the bound variables and the expression to the right of the arrow.

```
> type Alter a = (Int, [a], Expr a)
> type CoreAlt = Alter Name
```

We take the opportunity to define a useful function on expressions, a boolean-valued function, `isAtomicExpr`, which identifies 'atomic' expressions; that is, expressions with no internal structure:

```
> isAtomicExpr :: Expr a -> Bool
> isAtomicExpr (EVar v) = True
> isAtomicExpr (ENum n) = True
> isAtomicExpr e        = False
```

Finally, a Core-language program is just a list of supercombinator definitions:

```
> type Program a = [ScDefn a]
> type CoreProgram = Program Name
```

A supercombinator definition contains the name of the supercombinator, its arguments and its body:

```
> type ScDefn a = (Name, [a], Expr a)
> type CoreScDefn = ScDefn Name
```

The argument list might be empty, in the case of a supercombinator with no arguments.

We conclude with a small example. Consider the following small program.

```
        main = double 21 ;
        double x = x+x
```

This program is represented by the following Miranda expression, of type `coreProgram`:

```
        [("main",   [],    (EAp (EVar "double") (ENum 21))),
         ("double", ["x"], (EAp (EAp (EVar "+") (EVar "x")) (EVar "x")))
        ]
```

1.4 A small standard prelude

Miranda has a *standard prelude* which contains definitions of various useful functions (such as `map`, `foldr` and so on) which are always available. We will do the same for the Core language, by providing the following standard definitions:

```
I x = x ;
K  x y = x ;
K1 x y = y ;
S f g x = f x (g x) ;
compose f g x = f (g x) ;
twice f = compose f f
```

This 'standard prelude' is necessarily rather small, because we want it to work for *all* of our implementations, including the most primitive ones which will lack arithmetic and facilities for manipulating data structures. All that is available in the simplest implementations is function application!

The following definition for `preludeDefs`, which will be used throughout the book, embodies these definitions:

```
> preludeDefs :: CoreProgram
> preludeDefs
>   = [ ("I", ["x"], EVar "x"),
>       ("K", ["x","y"], EVar "x"),
>       ("K1",["x","y"], EVar "y"),
>       ("S", ["f","g","x"], EAp (EAp (EVar "f") (EVar "x"))
>                                (EAp (EVar "g") (EVar "x"))),
>       ("compose", ["f","g","x"], EAp (EVar "f")
>                                       (EAp (EVar "g") (EVar "x"))),
>       ("twice", ["f"], EAp (EAp (EVar "compose") (EVar "f")) (EVar "f")) ]
```

1.5 A pretty-printer for the Core language

Once we have a value of type `coreProgram` it is often convenient to be able to display it. Miranda's built-in features are not much help here. For example, if one types `preludeDefs` in response to the Miranda prompt, the output produced is rather hard to understand. (Try it.)

What we require is a 'pretty-printing' function `pprint`, with type

```
> pprint :: CoreProgram -> String
```

Then we could type `pprint preludeDefs`, and expect to get a list of characters which, when printed, looks like a nicely formatted version of `preludeDefs`. Our goal in this section is to write such a function.

When the result of a program is a list, Miranda usually prints out the list items separated by commas and surrounded by brackets. But in the special case when the result of the program is of type `[char]`, Miranda displays the list 'all squashed up', without square brackets and commas. For example, the value `"Hi\nthere"` is displayed as

```
Hi
there
```

and not as

```
['H', 'i', '\n', 't', 'h', 'e', 'r', 'e']
```

In this way, `pprint` can have complete control over the output format.

We will need some of the utility functions defined in Appendix A, so we import them using the `%include` directive:

1.5.1 Pretty-printing using strings

Let us first concentrate on Core-language expressions. It looks as though we require a pretty-printing function, `pprExpr`, defined something like this:

```
> pprExpr :: CoreExpr -> String
> pprExpr (ENum n) = show n
> pprExpr (EVar v) = v
> pprExpr (EAp e1 e2) = pprExpr e1 ++ " " ++ pprAExpr e2
```

(We have deliberately left out many of the cases for `pprExpr` for the moment.) `pprAExpr` has the same type as `pprExpr`, but differs from it by placing parentheses around the expression unless it is a variable or number.

```
> pprAExpr :: CoreExpr -> String
> pprAExpr e = isAtomicExpr e | pprExpr e
> pprAExpr e = otherwise | "(" ++ pprExpr e ++ ")"
```

One can proceed in this fashion, but there is a serious problem with doing so. The pretty-printer uses the list append function, `++`, a great deal. This can give very nasty performance, as the following example shows. Consider the expression

```
(xs1 ++ xs2) ++ xs3
```

The inner `++` takes time proportional to `#xs1`[2], but then the outer `++` takes time proportional to the length of `xs1++xs2`, so the total time taken is $(2 * \texttt{\#xs1}) + \texttt{\#xs2}$. In general, if we added more lists to this nested append, the cost can be quadratic in the length of the result! Of course, if we bracket the expression the other way, the cost is linear in the length of the result, but unfortunately we cannot guarantee this in a pretty-printer.

To demonstrate this effect, we will first write a function `mkMultiAp`, which makes it easy for us to build sample expressions of a given size. The call (`mkMultiAp` n e_1 e_2) generates a `coreExpr` representing the expression

$$e_1 \ \underbrace{e_2 \ e_2 \ \ldots \ e_2}_{n}$$

[2]The `#` function is a standard Miranda function for taking the length of a list.

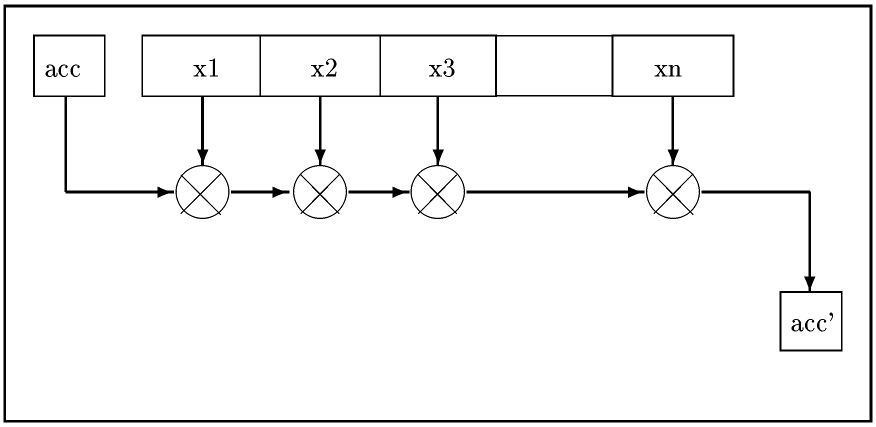

Figure 1.2: An illustration of `foldll` $\otimes$ acc $[x_1, \ldots, x_n]$

```
> mkMultiAp :: Int -> CoreExpr -> CoreExpr -> CoreExpr
> mkMultiAp n e1 e2 = foldll EAp e1 (take n e2s)
>                     where
>                     e2s = e2 : e2s
```

In this definition, `take` is a Miranda standard function which takes the first n elements of a list, discarding the rest of the list. The function `foldll` is a standard function, defined in Appendix A[3]. Given a dyadic function $\otimes$, a value acc and a list $xs = [x_1, ..., x_n]$, `foldll` $\otimes$ acc xs computes acc', where

$$acc' = (\ldots((acc \ \otimes \ x_1) \ \otimes \ x_2) \ \otimes \ \ldots x_n)$$

This is illustrated by Figure 1.2. In `mkMultiAp`, `foldll` is used to build a left-branching chain of `EAp` nodes. The initial accumulator acc is `e1`, and the combining function $\otimes$ is the `EAp` constructor. Finally, `e2s` is the infinite list `[e2, e2, ...]`; only its first `n` elements are used by `take`.

Exercise 1.1. Measure the number of Miranda steps required to compute

```
# (pprExpr (mkMultiAp n (EVar "f") (EVar "x")))
```

for various values of `n`. (You can use the Miranda directive `/count` to tell Miranda to print execution statistics. We take the length of the result so that the screen does not fill up with a huge printout.) Sketch a graph which shows how the execution cost rises with `n` and check that it is roughly quadratic in `n`.

1.5.2 An abstract data type for pretty-printing

A pretty-printer whose cost is quadratic in the size of the program to be printed is clearly unacceptable, so we had better find a way around it.

We can separate this problem into two parts: *'what operations do we want to perform?'*, and *'what is an efficient way to perform them?'*. In common with other languages, Miranda provides a way to make this distinction clear by introducing an *abstract data type*.

[3]We use `foldll` rather than the Miranda standard function `foldl` because different versions of Miranda have different definitions for `foldl`.

```
> iNil     :: Iseq                    -- The empty iseq
> iStr     :: String -> Iseq          -- Turn a string into an iseq
> iAppend  :: Iseq -> Iseq -> Iseq    -- Append two iseqs
> iNewline :: Iseq                    -- New line with indentation
> iIndent  :: Iseq -> Iseq            -- Indent an iseq
> iDisplay :: Iseq -> String          -- Turn an iseq into a string
```

The `abstype` keyword introduces an abstract data type, `iseq`. It is followed by the *interface* of the data type; that is, the operations which can be performed on the data type `iseq` and their type of each operation.

Given such a data type, we rewrite `pprExpr` to return an `iseq` instead of a list of characters:

```
> pprExpr :: CoreExpr -> Iseq
> pprExpr (EVar v) = iStr v
> pprExpr (EAp e1 e2) = (pprExpr e1) `iAppend` (iStr " ") `iAppend` (pprAExpr e2)
```

We have simply replaced `++` by `iAppend`[4], and added an `iStr` around literal strings.

What are the differences between an `iseq` and a list of characters? Firstly, we aim to produce an implementation of `iAppend` which does not have the unexpected quadratic behaviour of list append. Secondly, `iseq` provides new operations `iIndent` and `iNewline` which will be useful for controlling indentation. The idea is that `iIndent` indents its argument to line up with the current column; it should work even if its argument spreads over many lines, and itself contains calls to `iIndent`. `iNewline` stands for a newline followed by a number of spaces determined by the current level of indentation.

As an example of how `iIndent` and `iNewline` might be used, let us extend `pprExpr` to handle `let` and `letrec` expressions:

```
> pprExpr (ELet isrec defns expr)
>   = iConcat [  iStr keyword, iNewline,
>                iStr "  ",iIndent (pprDefns defns),iNewline,
>                iStr "in ",pprExpr expr ]
>     where
>     keyword | not isrec = "let"
>             | isrec = "letrec"


> pprDefns :: [(Name,CoreExpr)] -> Iseq
> pprDefns defns = iInterleave sep (map pprDefn defns)
>                  where
>                  sep = iConcat [ iStr ";", iNewline ]


> pprDefn :: (Name, CoreExpr) -> Iseq
> pprDefn (name, expr)
>   = iConcat [ iStr name, iStr " = ", iIndent (pprExpr expr) ]
```

[4]In Miranda, writing a dollar sign in front of an identifier turns it into an infix operator, allowing us to write `iAppend` between its arguments, instead of in front of them. Such infix operators are right-associative.

To make the definitions more legible, we have used two new functions, `iConcat` and `iInterleave`, with the types

```
> iConcat     :: [Iseq] -> Iseq
> iInterleave :: Iseq -> [Iseq] -> Iseq
```

`iConcat` takes a list of `iseq`s and uses `iAppend` to concatenate them into a single `iseq`. `iInterleave` is similar to `iConcat` except that it interleaves a specified `iseq` between each adjacent pair.

Exercise 1.2. Define `iConcat` and `iInterleave` in terms of `iAppend` and `iNil`.

In general, all our pretty-printing functions will return an `iseq`, and we apply `iDisplay` just once at the top level, to the `iseq` representing the entire thing we want to display:

```
> pprint prog = iDisplay (pprProgram prog)
```

Exercise 1.3. Add a further equation to `pprExpr` to handle `case` and lambda expressions, and write definitions for `pprAExpr` and `pprProgram` in the same style.

1.5.3 Implementing `iseq`

Now we come to the *implementation* of the `iseq` type. We begin by making an implementation that ignores all indentation. To implement the abstract data type we must say what type is used to represent an `iseq`:

```
> data Iseq = INil
>           | IStr String
>           | IAppend Iseq Iseq
```

The first declaration says that the type `iseqRep` is used to represent an `iseq`, while the second declares `iseqRep` to be an algebraic data type with the three constructors `INil`, `IStr` and `IAppend`.

The general idea of this particular representation is to postpone all the work until the eventual call of `iDisplay`. The operations `iNil`, `iStr` and `iAppend` all just use the relevant constructor:

```
> iNil              = INil
> iAppend seq1 seq2 = IAppend seq1 seq2
> iStr str             = IStr str
```

Since we are ignoring indentation, `iIndent` and `iNewline` are defined trivially. We will improve them in the next section.

```
> iIndent seq = seq
> iNewline = IStr "\n"
```

All the interest lies in the operation `iDisplay` which turns an `iseq` into a list of characters. The goal is that it should only take time linear in the size of the `iseq`. It turns out to be convenient to define `iDisplay` in terms of a more general function, `flatten`:

```
> flatten :: [Iseq] -> String
>
> iDisplay seq = flatten [seq]
```

The function `flatten` takes a *list* of `iseqReps`, and returns the result of concatenating each of the `iseqReps` in the list. The reason for having this list is that is allows us to accumulate a list of pending work, as we will soon see. Notice that `flatten` manipulates the *representation* type `iseqRep`, rather than the *abstract* type `iseq`.

We define `flatten` by case analysis on its argument, which we call the *work-list*. If the work-list is empty, we are done:

```
> flatten [] = ""
```

Otherwise, we work by doing case analysis on the first element of the work-list. The `INil` case just pops an item from the work-list:

```
> flatten (INil : seqs) = flatten seqs
```

The `IStr` case works by appending the specified string with the result of flattening the rest of the work-list:

```
> flatten (IStr s : seqs) = s ++ (flatten seqs)
```

So far, the fact that `flatten` takes a list has not helped us much. The justification for the list argument can be seen more clearly when we deal with `IAppend`; all that need be done is to push one more item onto the front of the work-list:

```
> flatten (IAppend seq1 seq2 : seqs)  = flatten (seq1 : seq2 : seqs)
```

Exercise 1.4. What is the cost of `flatten` in terms of the size of the `iseq`?

Change `pprExpr` to use `iseq` as indicated above, and measure the effect of the new implementation using the same experiment as in the previous exercise. Remember to apply `iDisplay` to the result of `pprExpr`.

Exercise 1.5. The key advantage of using an abstract data type is that one can change the *implementation* of the ADT without affecting its *interface*. As an example of this, redefine `iAppend` so that it returns a simplified result if either of its arguments is `INil`.

1.5.4 Layout and indentation

So far we have only given a rather trivial interpretation to the `iIndent` operation, and we now turn to improving it. In the same spirit as before, we first expand the `iseqRep` type with an extra two constructors, `IIndent` and `INewline`, and redefine their operations to use these constructors:

```
> data Iseq = INil
>           | IStr String
>           | IAppend Iseq Iseq
>           | IIndent Iseq
>           | INewline
>
> iIndent seq = IIndent seq
> iNewline    = INewline
```

We must then make `flatten` more powerful. Firstly, it needs to keep track of the current column, and secondly, its work-list must consist of `(iseq, num)` pairs, where the number gives the indentation required for the corresponding `iseq`:

```
> flatten :: Int                        -- Current column; 0 for first column
>            -> [(Iseq, Int)]           -- Work list
>            -> String                  -- Result
```

We need to change `iDisplay` to initialise `flatten` appropriately:

```
> iDisplay seq = flatten 0 [(seq,0)]
```

The interesting case for `flatten` is when we deal with `INewline`, because this is where we need to perform indentation[5]:

```
> flatten col ((INewline, indent) : seqs)
>  = '\n' : (space indent) ++ (flatten indent seqs)
```

Notice that the recursive call to flatten has a current-column argument of `indent` since we have now moved on to a new line and added `indent` spaces.

The `IIndent` case simply sets the current indentation from the current column:

```
> flatten col ((IIndent seq, indent) : seqs)
>  = flatten col ((seq, col) : seqs)
```

Exercise 1.6. Add equations for `flatten` for `IAppend`, `IStr` and `INil`.
Try `pprExpr` on an expression involving an `ELet`, and check that the layout works properly.

Exercise 1.7. The pretty-printer will go wrong if a newline character `'\n'` is embedded in a string given to `IStr`. Modify `iStr` to check for this, replacing the newline character by a use of `INewline`.

1.5.5 Operator precedence

As discussed in Section 1.3, the `coreExpr` type has no construct for infix operator applications. Instead, such applications are expressed in prefix form, just like any other function application. It would be nice if our pretty-printer recognised such applications, and printed them in infix form. This is easily done by adding extra equations to `pprExpr` of the form

[5] `spaces` is a standard Miranda function which returns a list of a specified number of space characters.

```
pprExpr (EAp (EAp (EVar "+") e1) e2)
= iConcat [   pprAExpr e1, iStr " + ", pprAExpr e2 ]
```

This still does not do a very good job, because it inserts too many parentheses. Would you prefer to see the expression

```
x + y > p * length xs
```

or the fully parenthesised version?

```
(x + y) > (p * (length xs))
```

The easiest way to achieve this is to give `pprExpr` an extra argument which indicates the precedence level of its context, and then use this to decide whether to add parentheses around the expression it produces. (The function `pprAExpr` now becomes redundant.)

Exercise 1.8. Make these changes to `pprExpr` and test them.

1.5.6 Other useful functions on `iseq`

Later on it will be useful to have a few more functions which work on `iseq`s. They are all defined in terms of the `iseq` interface functions, so the implementation can be changed without altering any of these definitions.

`iNum` maps a number to an `iseq` and `iFWNum` does the same except that the result is left-padded with spaces to a specified width:

```
> iNum :: Int -> Iseq
> iNum n = iStr (show n)


> iFWNum :: Int -> Int -> Iseq
> iFWNum width n
>   = iStr (space (width - length digits) ++ digits)
>     where
>     digits = show n
```

(If the number is wider than the width required, a negative number will be passed to `spaces`, which then returns the empty list. So the net effect is to return a field just wide enough to contain the number.) `iLayn` lays out a list, numbering the items and putting a newline character after each, just as the standard function `layn` does.

```
> iLayn :: [Iseq] -> Iseq
> iLayn seqs = iConcat (map lay_item (zip [1..] seqs))
>              where
>              lay_item (n, seq)
>                = iConcat [ iFWNum 4 n, iStr ") ", iIndent seq, iNewline ]
```

1.5.7 Summary

Our pretty-printer still has its shortcomings. In particular, a good pretty-printer will lay things out on one line if they fit, and over many lines if they do not. It is quite possible to elaborate the `iseq` data type so that it can do this, but we will not do so here.

The `iseq` type is useful for pretty-printing data other than programs, and we will use it for a variety of purposes throughout the book.

There are two general points we would like to bring out from this section:

- It is very often helpful to separate the *interface* of an abstract data type from its *implementation*. Miranda provides direct support for this abstraction, by ensuring the functions over the abstract type do not inspect the representation.
- The definition of `iDisplay` in terms of `flatten` exemplifies a very common technique called *generalisation*. We often define the function we really want in terms of a simple call to a more general function. This is usually because the more general function carries around some extra arguments which it needs to keep the book-keeping straight.

  It is hard to make general statements about when generalisation is an appropriate technique; indeed, working out a good generalisation is often the main creative step in writing any program. However, there are plenty of examples of generalisation in this book, which we hope will help to convey the idea.

1.6 A parser for the Core language

We will want to run each of our implementations on a variety of Core programs. This means that we want a way of taking a file containing the Core program in its concrete syntax, and parsing it to a value of type `coreProgram`.

Writing parsers is generally rather tiresome, so much so that great effort has been devoted to building tools which accept a grammar and write a parser for you. The Unix Yacc utility is an example of such a parser generator. In a functional language, however, it is quite easy to write a simple parser, and we will do so in this section for the Core language. We split the task into three stages:

- First, we obtain the contents of the named file, as a list of characters. This is done by the built-in Miranda function `read`.
- Next, the *lexical analysis* function `lex` breaks the input into a sequence of small chunks, such as identifiers, numbers, symbols and so on. These small chunks are called *tokens*:

  ```
  > clex :: String -> [Token]
  ```

- Finally, the *syntax analysis* function `syntax` consumes this sequence of tokens and produces a `coreProgram`:

  ```
  > syntax :: [Token] -> CoreProgram
  ```

The full parser is just the composition of these three functions:

```
> parse :: String -> CoreProgram
> parse = syntax . clex
> -- In Gofer I propose to compose this with some function
> -- CoreProgram -> String, which will illustrate some sort of
> -- execution machine, and then give this composition to catWith
> -- from my utils
```

The symbol '.' is Miranda's infix composition operator, which can be defined thus:

```
        (f . g) x  =  f (g x)
```

We could equivalently have defined `parse` without using composition, like this:

```
  parse filename = syntax (lex (read filename))
```

but it is nicer style to use composition, because it makes it particularly easy to see that we are defining `parse` as a pipeline of three functions.

1.6.1 Lexical analysis

We begin with the lexical analyser. We have not yet defined the type of a token. The easiest thing to begin with is to do no processing at all on the tokens, leaving them as (non-empty) strings:

```
> type Token = String           -- A token is never empty
```

Now the lexical analysis itself. It should throw away white space (blanks, tabs, newlines):

```
> clex (c:cs) | isWhiteSpace c = clex cs
```

It should recognise numbers as a single token:

```
> clex (c:cs) | isDigit c = num_token : clex rest_cs
>              where
>              num_token = c : takeWhile isDigit cs
>              rest_cs   = dropWhile isDigit cs
```

The standard function `digit` takes a character and returns `True` if and only if the character is a decimal digit. `takewhile` and `dropwhile` are both also standard functions; `takewhile` takes elements from the front of a list while a predicate is satisfied, and `dropwhile` removes elements from the front of a list while the predicate is satisfied. For example,

```
        takewhile digit "123abc456"
```

is the list "123".

The lexical analyser should also recognise variables, which begin with an alphabetic letter, and continue with a sequence of letters, digits and underscores:

```
> clex (c:cs) | isAlpha c = var_tok : clex rest_cs
>              where
>              var_tok = c : takeWhile isIdChar cs
>              rest_cs = dropWhile isIdChar cs
```

Here `letter` is a standard function like `digit` which returns `True` on alphabetic characters, and `isIdChar` is defined below.

If none of the above equations applies, the lexical analyser returns a token containing a single character.

```
> clex (c:cs) = [c] : clex cs
```

Lastly, when the input string is empty, `lex` returns an empty token list.

```
> clex [] = []
```

We conclude with the definitions of the auxiliary functions used above. (The operator '\/' is Miranda's boolean 'or' operation.)

```
> isIdChar, isWhiteSpace :: Char -> Bool
> isIdChar c = isAlpha c || isDigit c || (c == '_')
> isWhiteSpace c = c `elem` " \t\n"
```

Exercise 1.9. Modify the lexical analyser so that it ignores comments as well as white space. Use the same convention that a comment is introduced by a double vertical bar, `||`, and extend to the end of the line.

Exercise 1.10. The lexical analyser does not currently recognise two-character operators, such as `<=` and `==`, as single tokens. We define such operators by giving a list of them:

```
> twoCharOps :: [String]
> twoCharOps = ["==", "~=", ">=", "<=", "->"]
```

Modify `lex` so that it recognises members of `twoCharOps` as tokens. (The standard function `member` may be useful.)

Exercise 1.11. Since the lexical analysis throws away white space, the parser cannot report the line number of a syntax error. One way to solve this problem is to attach a line number to each token; that is, the type `token` becomes

```
token == (num, [char])
```

Alter the lexical analyser so that it does this. To do this you will need to add an extra parameter to `lex`, being the current line number.

1.6.2 Basic tools for parsing

In preparation for writing a parser for the Core language, we now develop some general-purpose functions to use when writing parsers. The techniques described below are well known [Fairbairn 1986, Wadler 1985], but make a rather nice demonstration of what can be done with functional programming. As a running example, we will use the following small grammar:

$$\begin{array}{lcl} \textit{greeting} & \rightarrow & \textit{hg person } \texttt{!} \\ \textit{hg} & \rightarrow & \texttt{hello} \\ & | & \texttt{goodbye} \end{array}$$

where *person* is any token beginning with a letter.

Our general approach, which is very common in functional programming, is to try to build a big parser by glueing together smaller parsers. The key question is: what should the type of a parser be? It is a function which takes a list of tokens as its argument, and at first it appears that it should just return the parsed value. But this is insufficiently general, for two reasons.

1. Firstly, it must also return the remaining list of tokens. If, for example, we want to parse two items from the input, one after the other, we can apply the first parser to the input, but we must then apply the second parser to the remaining input returned by the first.
2. Secondly, the grammar may be ambiguous, so there is more than one way to parse the input; or the input may not conform to the grammar, in which case there is no way to successfully parse the input. An elegant way to accommodate these possibilities is to return a list of possible parses. This list is empty if there is no way to parse the input, contains one element if there is a unique way to parse it, and so on.

We can summarise our conclusion by defining the type of parsers using a type synonym, like this:

```
> type Parser a = [Token] -> [(a, [Token])]
```

That is, a parser for values of type `*` takes a list of tokens and returns a list of parses, each of which consists of a value of type `*` paired with the remaining list of tokens.

Now we are ready to define some small parsers. The function `pLit` ('lit' is short for 'literal') takes a string and delivers a parser which recognises only tokens containing that string, returning the string as the value of the parse:

```
> pLit :: String -> Parser String
```

How does `pLit` work? It looks at the first token on the input and compares it with the desired string. If it matches, `pLit` returns a singleton list, indicating a single successful parse; if it does not match, `pLit` returns an empty list, indicating failure to parse[6]:

[6]This definition of `pLit` assumes that a token is just a string. If you have added line numbers to your tokens, as suggested in Exercise 1.11, then `pLit` will need to strip off the line number before making the comparison.

```
> pLit s (tok:toks) = s == tok | [(s, toks)]
>                   = otherwise | []
> pLit s []         = []
```

The second equation takes care of the case where the input stream is empty. We can use `pLit` to define parsers which look for particular tokens in the input. For example, the expression

```
pLit "hello" ["hello", "John", "!"]
```

evaluates to

```
[("hello", ["John", "!"])]
```

Similarly, we define a parser `pVar` to parse a variable from the beginning of the input:

```
> pVar :: Parser String
> pVar []         = []
```

`pVar` decides whether a token is a variable or not by looking at its first character. (The lexical analyser ensures that no token is empty.) Actually, this is not quite right, because it should not treat keywords as variables, but we will fix this problem later (Exercise 1.17).

The whole point of this development is to build bigger parsers by gluing together smaller ones, and we are now ready to do so. We will define a function `pAlt` ('alt' is short for 'alternative') which combines two parsers, say `p1` and `p2`. First it uses `p1` to parse the input, and then it uses `p2` to parse the *same* input; it returns all the successful parses returned by either `p1` or `p2`. So the type of `pAlt` is

```
> pAlt :: Parser a -> Parser a -> Parser a
```

The actual definition of `pAlt` is delightfully simple. All it needs to is append the lists of parses returned by `p1` and `p2`:

```
> pAlt p1 p2 toks = (p1 toks) ++ (p2 toks)
```

For example, `pHelloOrGoodbye` is a parser which recognises either the token `"hello"` or `"goodbye"`:

```
> pHelloOrGoodbye :: Parser String
> pHelloOrGoodbye = (pLit "hello") `pAlt` (pLit "goodbye")
```

It is easy to see that `pAlt` corresponds directly to the vertical bar, |, of a BNF grammar (see Figure 1.1, for example). We need one other fundamental parser-combining function, `pThen`, which corresponds to the *sequencing* of symbols in a BNF grammar.

Like `pAlt`, `pThen` combines two parsers, say `p1` and `p2`, returning a bigger parser which behaves as follows. First, it uses `p1` to parse a value from the input, and then it uses `p2` to parse a second value from the remaining input. What value should `pThen` return from a successful parse? Presumably some combination of the values returned by `p1` and `p2`, so the right thing to do is to give `pThen` a third argument which is the value-combining function. So the type of `pThen` is:

```
> pThen :: (a -> b -> c) -> Parser a -> Parser b -> Parser c
```

The definition of `pThen` makes use of a *list comprehension*:

```
> pThen combine p1 p2 toks
>   = [ (combine v1 v2, toks2) | (v1,toks1) <- p1 toks,
>                                (v2,toks2) <- p2 toks1]
```

The right-hand side of this equation should be read as follows:

> 'the list of pairs `(combine v1 v2, toks2)`,
> where `(v1,toks1)` is drawn from the list `p1 toks`,
> and `(v2,toks2)` is drawn from the list `p2 toks1`'.

With the aid of `pThen` we can make a parser for greetings:

```
> pGreeting :: Parser (String, String)
> pGreeting = pThen mk_pair pHelloOrGoodbye pVar
>             where
>             mk_pair hg name = (hg, name)
```

For example, the expression

```
        pGreeting ["goodbye", "James", "!"]
```

would evaluate to

```
        [(("goodbye", "James"), ["!"])]
```

Notice that when writing `pGreeting` we did not need to think about the fact that `pHelloOrGoodbye` was itself a composite parser. We simply built `pGreeting` out of its component parsers, each of which has the same standard interface. We could subsequently change `pHelloOrGoodbye` without having to change `pGreeting` as well.

1.6.3 Sharpening the tools

We have now completed the basic tools for developing parsers. In this section we will develop them in a number of ways.

The definition of `pGreeting` given above is not quite right, because the grammar demands an exclamation mark after the person's name. We could fix the problem like this:

```
  pGreeting = pThen keep_first
                        (pThen mk_pair pHelloOrGoodbye pVar)
                        (pLit "!")
              where
              keep_first hg_name exclamation = hg_name
              mk_pair hg name = (hg, name)
```

Since the final exclamation mark is always present, we have chosen not to return it as part of the parsed value; it is discarded by keep_first. This definition is rather clumsy, however. It would be more convenient to define a new function pThen3, so that we could write:

```
  pGreeting = pThen3 mk_greeting
                      pHelloOrGoodbye
                      pVar
                      (pLit "!")
              where
              mk_greeting hg name exclamation = (hg, name)
```

Exercise 1.12. Give the type of pThen3, write down its definition, and test the new version of pGreeting. Similarly, write pThen4, which we will need later.

Another very common feature of grammars is to require zero or more repetitions of a symbol. To reflect this we would like a function, pZeroOrMore, which takes a parser, p, and returns a new parser which recognises zero or more occurrences of whatever p recognises. The value returned by a successful parse can be the list of the values returned by the successive uses of p. So the type of pZeroOrMore is

```
> pZeroOrMore :: Parser a -> Parser [a]
```

For example, a parser to recognise zero or more greetings is

```
> pGreetings :: Parser [(String, String)]
> pGreetings = pZeroOrMore pGreeting
```

We can define pZeroOrMore by observing that it must either see one or more occurrences, or zero occurrences:

```
> pZeroOrMore p = (pOneOrMore p) `pAlt` (pEmpty [])
```

Here, pEmpty is a parser which always succeeds, removing nothing from the input, returning the value it is given as its first argument:

```
> pEmpty :: a -> Parser a
```

The function pOneOrMore has the same type as pZeroOrMore.

```
> pOneOrMore :: Parser a -> Parser [a]
```

Exercise 1.13. Write definitions for pOneOrMore and pEmpty. (Hint: you will find it convenient to call pZeroOrMore from pOneOrMore.) Test your definitions by using them to define a parser to recognise one or more greetings.

It is often convenient to process the values returned by successful parses. For example, suppose we wanted pGreetings to return the *number* of greetings rather than their content. To do this we would like to apply the length function, #, to the value returned by pZeroOrMore:

```
> pGreetingsN :: Parser Int
> pGreetingsN = (pZeroOrMore pGreeting) `pApply` length
```

Here `pApply` is a new parser-manipulation function, which takes a parser and a function, and applies the function to the values returned by the parser:

```
> pApply :: Parser a -> (a -> b) -> Parser b
```

Exercise 1.14. Write a definition for `pApply`, and test it. (Hint: use a list comprehension.)

Another very common pattern in grammars is to look for one or more occurrences of a symbol, separated by some other symbol. For example, a *program* in Figure 1.1 is a sequence of one or more supercombinator definitions, separated by semicolons. We need yet another parser-building function, `pOneOrMoreWithSep`, whose type is

```
> pOneOrMoreWithSep :: Parser a -> Parser b -> Parser [a]
```

The second argument is the parser which recognises the separators, which are not returned as part of the result; that is why there is only one occurrence of `**` in the type.

Exercise 1.15. Define and test `pOneOrMoreWithSep`. It may help to think of the following grammar for *program*:

$$\begin{array}{lcl} program & \rightarrow & sc\ programRest \\ programRest & \rightarrow & ;\ program \\ & | & \epsilon \end{array}$$

where ϵ is the empty string (corresponding to the `pEmpty` parser).

The parsers `pLit` and `pVar` are quite similar to each other: they both test for some property of the first token, and either fail (if it does not have the property) or succeed, returning the string inside the token (if it does have the property). We could generalise this idea by writing a parser `pSat` (where 'sat' is short for 'satisfies'), with the type

```
> pSat :: (String -> Bool) -> Parser String
```

`pSat` takes a function which tells whether or not the string inside the token has the desired property, and returns a parser which recognises a token with the property. Now we can write `pLit` in terms of `pSat`[7]:

```
> pLit s = pSat (== s)
```

Exercise 1.16. Define `pSat` and test it. Write `pVar` in terms of `pSat` in a similar way to `pLit`.

[7]The expression `(= s)` is called a *section*. It is the partial application of the equality operator `=` to one argument `s`, producing a function which tests whether its argument is equal to `s`.

`pSat` adds a useful level of modularity. For example, `pVar` currently recognises all alphabetic tokens as variables, but ultimately we might want it not to recognise language keywords (such as `let` and `case`) as variables.

Exercise 1.17. Modify the function passed to `pSat` in the definition of `pVar` above so that it does not treat strings in the list `keywords` as variables.

```
> keywords :: [String]
> keywords = ["let", "letrec", "case", "in", "of", "Pack"]
```

Exercise 1.18. As another example, use `pSat` to define a parser for numbers, with the type

```
> pNum :: Parser Int
```

`pNum` should use `pSat` to identify numeric tokens, and then `pApply` to convert the string to a number. (Miranda provides a standard function `numval` with type `[char] -> num` which can be used to do the hard work.)

There is an interesting performance problem associated with `pOneOrMore` and its related functions. Consider the following Core program:

```
f x = let x1 = x; x2 = x; ...; xn = x
      of x1
```

The idea is that we have a big `let` expression with definitions for `x1`, `x2`, ..., `xn` (the definitions are rather trivial, but they serve the purpose). This program has a syntax error: we have written '`of`' instead of '`in`' after the `let` expression.

Exercise 1.19. †Count how many Miranda steps it takes before the syntax error is reported, for `n` $= 5, 10, 15, 20$ and so on. (Use Miranda's `/count` directive to get a display of execution statistics.) How fast does the parsing cost rise, in terms of `n`?

To get an idea why this happens, try evaluating:

```
pOneOrMore (pLit "x") ["x", "x", "x", "x", "x", "x"]
```

You should get a list of six possible parses. Based on this, can you work out why the parsing cost in the previous example rises so fast?

How can this problem be solved? (Hint: apart from the first one, are any of the parses returned by `pOneOrMore` useful? How could the extra ones be eliminated?)

1.6.4 Parsing the Core language

We are finally ready to define a parser for the Core language. First we deal with the 'wrapper' function, `syntax`. Recall that it takes a list of tokens and delivers a result of type `coreProgram`. It can do this by calling the parser `pProgram` which parses the non-terminal *program* (Figure 1.1), and then selecting the first complete parse it returns. If it returns no complete parse — that is, one in which the sequence of tokens remaining after the parse is empty — `syntax` produces a (horribly uninformative) error message.

```
> syntax = take_first_parse . pProgram
>          where
>          take_first_parse ((prog,[]) : others) = prog
>          take_first_parse (parse     : others) = take_first_parse others
>          take_first_parse other                = error "Syntax error"
```

The beauty of our parsing tools is that *we can write parsers by merely transliterating the grammar into Miranda.* For example, consider the productions for *program* and *sc* in Figure 1.1:

$$\begin{array}{lcll} program & \rightarrow & sc_1 \texttt{;} \; \ldots \texttt{;} \; sc_n & (n \geq 1) \\ sc & \rightarrow & var\ var_1 \ldots var_n \texttt{ = } expr & (n \geq 0) \end{array}$$

We can transliterate these directly into Miranda:

```
> pProgram :: Parser CoreProgram
> pProgram = pOneOrMoreWithSep pSc (pLit ";")


> pSc :: Parser CoreScDefn
> pSc = pThen4 mk_sc pVar (pZeroOrMore pVar) (pLit "=") pExpr
```

Exercise 1.20. Write the function `mk_sc`. It takes four arguments returned by the four parsers used in `pSc`, and builds a value of type:

```
(name, [name], coreExpr)
```

It is a straightforward matter to complete the definitions for the rest of the grammar, apart from the productions for application and infix operators.

Exercise 1.21. Leaving these two productions out, complete the parser. A little care is needed for the parser `pAexpr`, which should have type `parser coreExpr`. The `pApply` function is required to wrap an `EVar` constructor around the value returned by `pVar`, and an `ENum` constructor around that returned by `pNum`.

Test your parser on the following program

```
f = 3 ;
g x y = let z = x in z ;
h x = case (let y = x in y) of
        <1> -> 2 ;
        <2> -> 5
```

You will find that the output becomes illegible as you run the parser on larger programs. To solve this, use the pretty-printing function `pprint` to format the output of your parser.

Exercise 1.22. Consider the program

```
f x y = case x of
        <1> -> case y of
               <1> -> 1;
        <2> -> 2
```

Does the alternative starting with `<2>` attach to the inner `case` or the outer one? Work out your answer, and see if your parser behaves as you expect. This is known as the 'dangling else' question.

Now we turn our attention to the two problems mentioned above.

1.6.5 Left recursion

The problem with applications is relatively easy to solve. The production for applications looks like this:

$$expr \;\rightarrow\; expr \; aexpr$$

If we simply transliterate this to

```
pExpr = pThen EAp pExpr pAexpr
```

then unfortunately `pExpr` will never terminate, because it keeps calling itself indefinitely. The problem is that *expr* appears as the first symbol in a production of *expr*; this is called *left recursion*. Our parsing tools simply cannot cope with left-recursive grammars. Fortunately, it is usually possible to transform the grammar so that it is no longer left-recursive, though the resulting grammar does not then reflect the structure of the result we are trying to construct. In this case, for example, we can simply use repetition, transforming the offending production to

$$expr \;\rightarrow\; aexpr_1 \ldots aexpr_n \qquad (n \geq 1)$$

and now the parser `(pOneOrMore pAexpr)` can be used. The trouble is that this returns a list of expressions, rather than a single expression built with `EAp` constructors. We can solve this using `pApply`, giving the parser

```
(pOneOrMore pAexpr) $pApply mk_ap_chain
```

Exercise 1.23. Define the appropriate function `mk_ap_chain` with type `[coreExpr] -> coreExpr`. Add the production for applications to your parser and test it.

1.6.6 Adding infix operators

The first problem with infix operators is that their precedence is implicit in the grammar of Figure 1.1. The standard way to make this explicit is to have several sorts of expression, as shown in Figure 1.3.

Notice the way that this grammar expresses the fact that `|` and `&` are right-associative, whereas relational operators are non-associative. Having to write out so many rules is rather tiresome, but we are only making explicit what we meant all along. But now the second problem arises: a parser implemented directly from these rules would be horribly inefficient! Consider the productions for *expr*1. A naive parser would attempt to recognise an *expr*2, and then look for a vertical bar `|`. If it did not find one (as will often be the case), *it will laboriously reparse the original input to look for an expr*2 *again.* Worse, each attempt to parse an *expr*2 may involve two attempts to parse an *expr*3, and hence four attempts to parse an *expr*4, and so on.

We want to share the parsing of the *expr*2 between the two productions, and this is not hard to do, by splitting the *expr*1 production into two:

$$\begin{array}{lcl} expr1 & \rightarrow & expr2\ expr1c \\ expr1c & \rightarrow & \texttt{|}\ expr1 \\ & | & \epsilon \end{array}$$

$$
\begin{array}{lcll}
expr & \rightarrow & \texttt{let}\ defns\ \texttt{in}\ expr & \\
 & | & \texttt{letrec}\ defns\ \texttt{in}\ expr & \\
 & | & \texttt{case}\ expr\ \texttt{of}\ alts & \\
 & | & \texttt{\\}\ var_1 \ldots var_n\ \texttt{.}\ expr & \\
 & | & expr1 & \\
 & & & \\
expr1 & \rightarrow & expr2\ \texttt{|}\ expr1 & \\
 & | & expr2 & \\
expr2 & \rightarrow & expr3\ \texttt{\&}\ expr2 & \\
 & | & expr3 & \\
expr3 & \rightarrow & expr4\ relop\ expr4 & \\
 & | & expr4 & \\
expr4 & \rightarrow & expr5\ \texttt{+}\ expr4 & \\
 & | & expr5\ \texttt{-}\ expr5 & \\
 & | & expr5 & \\
expr5 & \rightarrow & expr6\ \texttt{*}\ expr5 & \\
 & | & expr6\ \texttt{/}\ expr6 & \\
 & | & expr6 & \\
expr6 & \rightarrow & aexpr_1 \ldots aexpr_n & (n \geq 1)
\end{array}
$$

Figure 1.3: Grammar expressing operator precedence and associativity

Here ϵ stands for the empty string; the productions for *expr*1*c* say that an *expr*1*c* is either a vertical bar, `|`, followed by an *expr*1, or it is empty. We are almost there! The last question is: what is the type of a parser for *expr*1*c*. It cannot be of type `parser coreExpr`, because the phrase `|` *expr*1 is only part of an expression, and the empty string ϵ is not an expression either. As usual, transforming the grammar has destroyed the structure.

The solution is fairly easy. We define a new data type `partialExpr`, like this

```
> data PartialExpr = NoOp | FoundOp Name CoreExpr
```

Now we can define the parser for *expr*1*c* like this:

```
> pExpr1c :: Parser PartialExpr
> pExpr1c = (pThen FoundOp (pLit "|") pExpr1) `pAlt` (pEmpty NoOp)
```

The parser for *expr*1 takes apart the intermediate result returned by `pExpr1c`:

```
> pExpr1 :: Parser CoreExpr
> pExpr1 = pThen assembleOp pExpr2 pExpr1c


> assembleOp :: CoreExpr -> PartialExpr -> CoreExpr
> assembleOp e1 NoOp = e1
> assembleOp e1 (FoundOp op e2) = EAp (EAp (EVar op) e1) e2
```

Exercise 1.24. Transform the grammar along the lines suggested, transliterate the changes into Miranda code, and test the resulting parser.

1.6.7 Summary

The grammars that can be handled efficiently by our library of parser-building functions are called LL(1) grammars, exactly the same class that can be dealt with by conventional recursive-descent parsers [Aho *et al.* 1986].

Using the library we can easily write very concise parsers. This is an important and useful property, because almost any program has an input language of some sort, which has to be parsed by the program.

There are various things we have to take care about (left recursion, operator precedence, sharing), but exactly the same issues arise in any recursive-descent parser, regardless of the language in which it is implemented.

```
> module Template where
> import Language
> import Utils
```