9
Writing Interpreters for the λ-Calculus

This chapter brings together all the concepts we have learned so far. For an extended example, it presents a collection of modules to implement the λ-calculus as a primitive functional programming language. Terms of the λ-calculus can be parsed, evaluated and the result displayed. It is hardly a practical language. Trivial arithmetic calculations employ unary notation and take minutes. However, its implementation involves many fundamental techniques: parsing, representing bound variables and reducing expressions to normal form. These techniques can be applied to theorem proving and computer algebra.

Chapter outline

We consider parsing and two interpreters for λ-terms, with an overview of the λ-calculus. The chapter contains the following sections:

A functional parser. An ML functor implements top-down recursive descent parsing. Parsers can be combined using infix operators that resemble the symbols for combining grammatical phrases.

Introducing the λ-calculus. Terms of this calculus can express functional programs. They can be evaluated using either the call-by-value or the call-by-name mechanism. Substitution must be performed carefully, avoiding variable name clashes.

Representing λ-terms in ML. Substitution, parsing and pretty printing are implemented as ML structures.

The λ-calculus as a programming language. Typical data structures of functional languages, including infinite lists, are encoded in the λ-calculus. The evaluation of recursive functions is demonstrated.

A functional parser

Before discussing the λ-calculus, let us consider how to write scanners and parsers in a functional style. The parser described below complements the pretty printer of the previous chapter. Using these tools, ML programs can read and write λ-terms, ML types and logical formulæ.

9.1 *Scanning, or lexical analysis*

A parser seldom operates directly on a string of characters. The characters are first ***scanned***: processed into ***tokens*** such as keywords, identifiers, special symbols and numbers. The parser is supplied a list of tokens.

This two-level approach simplifies the grammar used for parsing. The scanner removes spaces, line breaks and comments in some uniform fashion, leaving the parser to deal with more complex matters of syntax. Scanning can be performed by a finite-state machine. Such a machine, driven by character-indexed arrays, can run extremely fast. For small inputs, the scanning functions in the library structure *Substring* do the job quite well.

A lexical analyser, or lexer, is a structure with the following signature:

```
signature LEXICAL =
  sig
  datatype token = Id of string | Key of string
  val scan : string -> token list
  end;
```

A *token* is either an identifier or a keyword. This simple scanner does not recognize numbers. Calling *scan* performs lexical analysis on a string and returns the resulting list of tokens.

Before we can parse a language, we must specify its vocabulary. To classify tokens as identifiers or keywords, the scanner must be supplied with an instance of the signature *KEYWORD*:

```
signature KEYWORD =
  sig
  val alphas   : string list
  and symbols : string list
  end;
```

The list *alphas* defines the alphanumeric keywords like `"if"` and `"let"`, while *symbols* lists symbolic keywords like `"("` and `")"`. The two kinds of keywords are treated differently:

- A string of alphanumeric characters is scanned as far as possible — until it is not followed by another letter or digit. It is classified as a keyword if it belongs to *alphas*, and as an identifier otherwise.
- A string of symbolic characters is scanned until it matches some element of *symbols*, or until it is not followed by another symbolic character. It is always classified as a keyword. For instance, if `"("` belongs to *symbols* then the string `"(("` is scanned as two `"("` tokens, and as one `"(("` token otherwise.

Figure 9.1 *The lexical analysis functor*

```
functor Lexical (Keyword: KEYWORD) : LEXICAL =
  struct
  datatype token = Key of string  |  Id of string;

  fun member (x:string, l) = List.exists (fn y => x=y) l;

  fun alphaTok a =
      if member(a, Keyword.alphas) then Key(a) else Id(a);

  (*scanning of a symbolic keyword*)
  fun symbolic (sy, ss) =
      case Substring.getc ss of
          NONE         => (Key sy, ss)
        | SOME (c,ss1) =>
              if member(sy, Keyword.symbols)
                  orelse not (Char.isPunct c)
              then  (Key sy, ss)
              else  symbolic (sy ^ String.str c, ss1);

  (*Scanning a substring into a list of tokens*)
  fun scanning (toks, ss) =
      case Substring.getc ss of
          NONE         => rev toks     (*end of substring*)
        | SOME (c,ss1) =>
            if Char.isAlphaNum c
                then (*identifier or keyword*)
                 let val (id, ss2) = Substring.splitl Char.isAlphaNum ss
                     val tok       = alphaTok (Substring.string id)
                 in  scanning (tok::toks, ss2)
                 end
            else if Char.isPunct c
            then (*special symbol*)
                 let val (tok, ss2) = symbolic (String.str c, ss1)
                 in  scanning (tok::toks, ss2)
                 end
            else (*ignore spaces, line breaks, control characters*)
                 scanning (toks, Substring.dropl (not o Char.isGraph) ss);

  fun scan a = scanning([], Substring.all a);
  end;
```

Functor *Lexical* (Figure 9.1 on the preceding page) implements the scanner using several *Substring* functions: *getc*, *splitl*, *string*, *dropl* and *all*. The function *getc* splits a substring into its first character, paired with the rest of the substring; if the substring is empty, the result is *NONE*. In Section 8.9 we met the functions *all* and *string*, which convert between strings and substrings. We also met *splitl*, which scans a substring from the left, splitting it into two parts. The function *dropl* is similar but returns only the second part of the substring; the scanner uses it to ignore spaces and other non-printing characters. The library predicate *Char.isAlphaNum* recognizes letters and digits, while *Char.isGraph* recognizes all printing characters and *Char.isPunct* recognizes punctuation symbols.

The code is straightforward and efficient, as fast as the obvious imperative implementation. The *Substring* functions yield functional behaviour, but they work by incrementing indices. This is better than processing lists of characters.

The functor declares function *member* for internal use. It does not depend upon the infix *mem* declared in Chapter 3, or on any other top level functions not belonging to the standard library. The membership test is specific to type *string* because polymorphic equality can be slow.

The lexical analyser is implemented as a functor because the information in signature *KEYWORD* is static. We only need to change the list of keywords or special symbols when parsing a new language. Applying the functor to some instance of *KEYWORD* packages that information into the resulting structure. We could have implemented the lexer as a curried function taking similar information as a record, but this would complicate the lexer's type in exchange for needless flexibility.

Exercise 9.1 Modify the scanner to recognize decimal numerals in the input. Let a new constructor *Num* : *integer* → *token* return the value of a scanned integer constant.

Exercise 9.2 Modify the scanner to ignore comments. The comment brackets, such as `"(*"` and `"*)"`, should be supplied as additional components of the structure *Keyword*.

9.2 *A toolkit for top-down parsing*

A top-down recursive descent parser closely resembles the grammar that it parses. There are procedures for all the syntactic phrases, and their mutually recursive calls precisely mirror the grammar rules.

The resemblance is closer in functional programming. Higher-order functions can express syntactic operations such as concatenation of phrases, alternative phrases and repetition of a phrase. With an appropriate choice of infix operators, a functional parser can be coded to look almost exactly like a set of grammar rules. Do not be fooled; the program has all the limitations of top-down parsing. In particular, a ***left-recursive*** grammar rule such as

$$exp = exp \texttt{ "*"}$$

makes the parser run forever! Compiler texts advise on coping with these limitations.

Outline of the approach. Suppose that the grammar includes a certain class of phrases whose meanings can be represented by values of type τ. A ***parser*** for such phrases must be a function of type

$$token\ list \to \tau \times token\ list,$$

henceforth abbreviated as type $\tau\ phrase$. When the parser is given a list of tokens that begins with a valid phrase, it removes those tokens and computes their meaning as a value of type τ. The parser returns the pair of this meaning and the remaining tokens. If the token list does not begin with a valid phrase, then the parser rejects it by raising exception $SyntaxErr$.

Not all functions of type $\tau\ phrase$ are parsers. A parser must only remove tokens from the front of the token list; it must not insert tokens, or modify the token list in any other way.

To implement complex parsers, we define some primitive parsers and some operations for combining parsers.

Parsing primitive phrases. The trivial parsers recognize an identifier, a specific keyword, or the empty phrase. They remove no more than one token from their input:

- The parser id, of type $string\ phrase$, removes an Id token from its input and returns this identifier as a string (paired with the tail of the token list).
- The parser $\$a$ has type $string\ phrase$ if a is a string. It removes the keyword token $Key\ a$ from its input and returns a paired with the tail of the token list.
- The parser $empty$ has the polymorphic type $(\alpha\ list)\ phrase$. It returns [] paired with the original token list.

The first two of these reject their input unless it begins with the required token, while *empty* always succeeds.

Alternative phrases. If $ph1$ and $ph2$ have type τ *phrase* then so does $ph1$ `||` $ph2$. The parser $ph1$ `||` $ph2$ accepts all the phrases that are accepted by either of the parsers $ph1$ or $ph2$. This parser, when supplied with a list of tokens, passes them to $ph1$ and returns the result if successful. If $ph1$ rejects the tokens then $ph2$ is attempted.

The parser `!!` ph is the same as ph, except that if ph rejects the tokens then the entire parse fails with an error message. This prevents an enclosing `||` operator from attempting to parse the phrase in another way. The operator `!!` is typically used in phrases that start with a distinguishing keyword, and therefore have no alternative parse; see `$--` below.

Consecutive phrases. The parser $ph1$`--`$ph2$ accepts a $ph1$ phrase followed by a $ph2$ phrase. This parser, when supplied with a list of tokens, passes them to $ph1$. If $ph1$ parses a phrase and returns $(x, toks2)$ then the remaining tokens $(toks2)$ are passed to $ph2$. If $ph2$ parses a phrase and returns $(y, toks3)$ then $ph1$`--`$ph2$ returns $((x, y), toks3)$. Note that $toks3$ consists of the tokens remaining after both parses. If either parser rejects its input then so does $ph1$`--`$ph2$.

Thus, the meaning of $ph1$`--`$ph2$ is the pair of the meanings of $ph1$ and $ph2$, applied to consecutive segments of the input. If $ph1$ has type τ_1 *phrase* and $ph2$ has type τ_2 *phrase* then $ph1$`--`$ph2$ has type $(\tau_1 \times \tau_2)$ *phrase*.

The operator `$--` covers a common case. The parser a `$--` ph resembles `$`a `--` ph; it parses a phrase that begins with the keyword token a and continues as ph. But it returns the meaning of ph, not paired with a. Moreover, if ph rejects its tokens then (using `!!`) it fails with an error message. The operator `$--` is appropriate if only one grammar rule starts with the symbol a.

Modifying the meaning. The parser ph`>>`f accepts the same inputs as ph, but returns $(f(x), toks)$ when ph returns $(x, toks)$. Thus, it assigns the meaning $f(x)$ when ph assigns the meaning x. If ph has type σ *phrase* and f has type $\sigma \to \tau$ then ph`>>`f has type τ *phrase*.

Repetition. To illustrate these operators, let us code a parsing functional. If ph is any parser then *repeat* ph will parse zero or more repetitions of ph:

```
fun repeat ph toks = (    ph -- repeat ph >> (op::)
                      || empty    ) toks;
```

The precedences of the infix operators are `--`, `>>`, `||` from highest to lowest. The body of *repeat* consists of two parsers joined by `||`, resembling the obvious grammatical definition: a repetition of ph is either a ph followed by a repetition of ph, or is empty.

The parser ph `--` *repeat* ph returns $((x, xs), toks)$, where xs is a list. The operator `>>` applies a list 'cons' (the operator ::), converting the pair (x, xs) to $x :: xs$. In the second line, *empty* yields [] as the meaning of the empty phrase. In short, *repeat* ph constructs the list of the meanings of the repeated phrases. If ph has type τ *phrase* then *repeat* ph has type $(\tau\ list)\ phrase$.

Beware of infinite recursion. Can the declaration of *repeat* be simplified by omitting *toks* from both sides? No — calling *repeat* ph would immediately produce a recursive call to *repeat* ph, resulting in disaster:

```
fun repeat ph = ph -- repeat ph >> (op::)  ||  empty;
```

Mentioning the formal parameter *toks* is a device to delay evaluation of the body of *repeat* until it is given a token list; the inner *repeat* ph is normally given a shorter token list and therefore terminates. Lazy evaluation would eliminate the need for this device.

9.3 *The ML code of the parser*

Infix directives for the operators `$--`, `--`, `>>` and `||` assign appropriate precedences to them. The exact numbers are arbitrary, but `$--` must have higher precedence than `--` in order to attract the string to its left. Also `>>` must have lower precedence than `--` in order to encompass an entire grammar rule. Finally, `||` must have the lowest precedence so that it can combine grammar rules.

```
infix 6 $--;
infix 5 --;
infix 3 >>;
infix 0 ||;
```

These directives have global effect because they are made at top level. We should also `open` the structure containing the parsing operators: compound names cannot be used as infix operators.

Functor *Parsing* (Figure 9.3 on page 371) implements the parser. The functor declaration has the primitive form that takes exactly one argument structure, in this case *Lex*. Its result signature is *PARSE* (Figure 9.2).

You may notice that many of the types in this signature differ from those given in the previous section. The type abbreviation

$$\alpha\ phrase = token\ list \to \alpha \times token\ list$$

Figure 9.2 *Signature for functional parsers*

```
signature PARSE =
  sig
  exception SyntaxErr of string
  type token
  val id     : token list -> string * token list
  val $      : string -> token list -> string * token list
  val empty  : 'a -> 'b list * 'a
  val ||     : ('a -> 'b) * ('a -> 'b) -> 'a -> 'b
  val !!     : ('a -> 'b * 'c) -> 'a -> 'b * 'c
  val --     : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> ('b * 'd) * 'e
  val $--    : string * (token list -> 'a * 'b) -> token list -> 'a * 'b
  val >>     : ('a -> 'b * 'c) * ('b -> 'd) -> 'a -> 'd * 'c
  val repeat : ('a -> 'b * 'a) -> 'a -> 'b list * 'a
  val infixes :
      (token list -> 'a * token list) * (string -> int) *
      (string -> 'a -> 'a -> 'a) -> token list -> 'a * token list
  val reader : (token list -> 'a * 'b list) -> string -> 'a
  end;
```

is not used; more importantly, some of the types in the signature are more general than is necessary for parsing. They are not restricted to token lists.

ML often assigns a function a type that is more polymorphic than we expect. If we specify the signature prior to coding the functor — which is a disciplined style of software development — then any additional polymorphism is lost. When designing a signature, it is sometimes useful to consult the ML top level and note what types it suggests.

Signature *PARSE* specifies the type $token$ in order to specify the types of id and other items. Accordingly, $Parsing$ declares the type $token$ to be equivalent to $Lex\,.\,token$.

The function $reader$ packages a parser for outside use. Calling $reader\ ph\ a$ scans the string a into tokens and supplies them to the parsing function ph. If there are no tokens left then $reader$ returns the meaning of the phrase; otherwise it signals a syntax error.

Parsing infix operators. The function $infixes$ constructs a parser for infix operators, when supplied with the following arguments:

- ph accepts the atomic phrases that are to be combined by the operators.

Figure 9.3 *The parsing functor*

```
functor Parsing (Lex: LEXICAL) : PARSE =
  struct
  type token = Lex.token;

  exception SyntaxErr of string;

  fun id (Lex.Id a :: toks) = (a,toks)
    | id toks               = raise SyntaxErr "Identifier expected";

  fun $a (Lex.Key b :: toks) = if a=b then (a,toks)
                                      else raise SyntaxErr a
    | $a _                   = raise SyntaxErr "Symbol expected";

  fun empty toks = ([],toks);

  fun (ph1 || ph2) toks = ph1 toks  handle SyntaxErr _ => ph2 toks;

  fun !! ph toks = ph toks  handle SyntaxErr msg =>
                            raise Fail ("Syntax error: " ^ msg);

  fun (ph1 -- ph2) toks =
      let val (x,toks2) = ph1 toks
          val (y,toks3) = ph2 toks2
      in  ((x,y), toks3)  end;

  fun (ph>>f) toks =
      let val (x,toks2) = ph toks
      in  (f x, toks2)  end;

  fun (a $-- ph) = ($a -- !!ph >> #2);

  fun repeat ph toks = (   ph -- repeat ph >> (op::)
                       || empty    ) toks;

  fun infixes (ph,precof,apply) =
    let fun over k toks = next k (ph toks)
        and next k (x, Lex.Key(a)::toks) =
              if precof a < k then (x, Lex.Key a :: toks)
              else next k ((over (precof a) >> apply a x) toks)
          | next k (x, toks) = (x, toks)
    in  over 0  end;

  (*Scan and parse, checking that no tokens remain*)
  fun reader ph a =
         (case ph (Lex.scan a) of
              (x, [])   => x
            | (_, _::_) => raise SyntaxErr "Extra characters in phrase");
  end;
```

- $prec\_of$ gives the precedences of the operators, returning -1 for all keywords that are not infix operators.
- $apply$ combines the meanings of phrases; $apply\ a\ x\ y$ applies the operator a to operands x and y.

The resulting parser recognizes an input like

$$ph \oplus ph \otimes ph \ominus ph \oslash ph$$

and groups the atomic phrases according to the precedences of the operators. It employs the mutually recursive functions $over$ and $next$.

Calling $over\ k$ parses a series of phrases, separated by operators of precedence k or above. In $next\ k\ (x, toks)$ the argument x is the meaning of the preceding phrase and k is the governing precedence. The call does nothing unless the next token is an operator a of precedence k or above; in this case, tokens are recursively parsed by $over(prec\_of\ a)$ and their result combined with x. The result and the remaining tokens are then parsed under the original precedence k.

The algorithm does not handle parentheses; this should be done by ph. Section 10.6 demonstrates the use of $infixes$.

Writing a backtracking parser. A grammar is ***ambiguous*** if some token list admits more than one parse. Our method is easily modified such that each parsing function returns a sequence (lazy list) of successful outcomes. Inspecting elements of this sequence causes backtracking over all parses of the input.

The parser $ph1 -- ph2$ returns the sequence of all possible ways of parsing a $ph1$ followed by a $ph2$. It applies $ph1$ to the input, which yields a sequence of $(x, toks2)$ pairs. For each element of this sequence it applies $ph2$ to $toks2$, obtaining a sequence of $(y, toks3)$ pairs. Finally it returns the sequence of all successful outcomes $((x, y), toks3)$. For each outcome, the meaning (x, y) consists of a pair of meanings returned by $ph1$ and $ph2$.

A parser rejects its input by returning the empty sequence rather than by raising an exception. Note that if $ph1$ rejects its input or if $ph2$ rejects each of the outcomes of $ph1$ then $ph1 -- ph2$ yields the empty sequence, rejecting its input.

This is an enjoyable exercise in sequence processing, but it suffers from the drawbacks of backtracking parsers: it is slow and handles errors poorly. It can take exponential time to parse the input; bottom-up parsing would be much faster. If the input contains a syntax error, a backtracking parser returns no information other than an empty sequence. Our parser can easily be made to pinpoint syntax errors. Modify type $token$ so that each token carries its position in the input string, and make `!!` include that information in its error messages.

Backtracking is valuable in theorem proving. A 'tactic' for finding proofs can be expressed as a function that takes a goal and returns a sequence of solutions. Tactics can be combined to form effective search procedures. The next chapter presents this technique, which is related to our treatment of parsing functions.

Exercise 9.3 Give an example of a parser ph such that, for all inputs, ph terminates successfully but $repeat\ ph$ runs forever.

Exercise 9.4 A ***parse tree*** is a tree representing the structure of a parsed token list. Each node stands for a phrase, with branches to its constituent symbols and sub-phrases. Modify our parsing method so that it constructs parse trees. Declare a suitable type $partree$ of parse trees such that each parsing function can have type

$$token\ list \to partree \times token\ list.$$

Code the operators `||`, `--`, id, `$`, $empty$ and $repeat$; note that `>>` no longer serves any purpose.

Exercise 9.5 Modify the parsing method to generate a sequence of successful results, as described above.

Exercise 9.6 Code the parsing method in a procedural style, where each parsing 'function' has type $unit \to \alpha$ and updates a reference to a token list by removing tokens from it. Does the procedural approach have any drawbacks, or is it superior to the functional approach?

Exercise 9.7 Modify signature *PARSE* to specify a substructure Lex of signature *LEXICAL* rather than a type $token$, so that other signature items can refer to the type $Lex\,.\,token$. Modify the functor declaration accordingly.

Exercise 9.8 When an expression contains several infix operators of the same precedence, does $infixes$ associate them to the left or to the right? Modify this function to give the opposite association. Describe an algorithm to handle a mixture of left and right-associating operators.

9.4 *Example: parsing and displaying types*

The parser and pretty printer will now be demonstrated using a grammar for ML types. For purposes of the example, ML's type system can be simplified by dropping record and product types. There are two forms of type to consider:

Types such as int, $bool\ list$ and $(\alpha\ list) \to (\beta\ list)$ consist of a ***type constructor*** applied to zero or more ***type arguments***. Here the type constructor int is applied to zero arguments; $list$ is applied to the type $bool$; and $\to$ is applied to the types $\alpha\ list$ and $\beta\ list$. ML adopts a postfix syntax for most type constructors, but $\to$ has an infix syntax. Internally, such types can be represented by a string paired with a list of types.

A type can consist merely of a type variable, which can be represented by a string. Our structure for types has the following signature:

```
signature TYPE =
  sig
  datatype t = Con of string * t list | Var of string
  val pr   : t -> unit
  val read : string -> t
  end;
```

It specifies three components:

- The datatype t comprises the two forms of type, with Con for type constructors and Var for type variables.
- Calling $pr\ ty$ prints the type ty at the terminal.
- The function $read$ converts a string to a type.

We could implement this signature using a functor whose formal parameter list would contain the signatures *PARSE* and *PRETTY*. But it is generally simpler to avoid writing functors unless they will be applied more than once. Let us therefore create structures for the lexical analyser and the parser, for parsing types. They will be used later to implement the λ-calculus.

Structure $LamKey$ defines the necessary symbols. Structure $LamLex$ provides lexical analysis for types and the λ-calculus, while $LamParsing$ provides parsing operators.

```
structure LamKey =
    struct val alphas  = []
           and symbols = ["(", ")", "'", "->"]
    end;
structure LamLex     = Lexical  (LamKey);
structure LamParsing = Parsing  (LamLex);
```

Structure $Type$ (Figure 9.4) matches signature *TYPE*. For simplicity, it only treats the $\to$ symbol; the other type constructors are left as an exercise. The grammar defines types and atomic types in mutual recursion. An atomic type is

Figure 9.4 *Parsing and displaying* ML *types*

```
structure Type : TYPE =
  struct

  datatype t = Con of string * t list
             | Var of string;

  local (** Parsing **)
    fun makeFun (ty1,ty2) = Con("->",[ty1,ty2]);
    open LamParsing

    fun typ toks =
     (    atom -- "->" $-- typ            >> makeFun
      || atom
     ) toks
    and atom toks =
      (    $"'" -- id                      >> (Var o op^)
       || "(" $-- typ -- $")"              >> #1
      ) toks;
  in
    val read = reader typ;
  end;

  local (** Display **)
    fun typ (Var a)                 = Pretty.str a
      | typ (Con("->",[ty1,ty2])) = Pretty.blo(0, [atom ty1,
                                                   Pretty.str " ->",
                                                   Pretty.brk 1,
                                                   typ ty2])

     and atom (Var a) = Pretty.str a
       | atom ty      = Pretty.blo(1, [Pretty.str"(",
                                       typ ty,
                                       Pretty.str")"]);
  in
    fun pr ty = Pretty.pr (TextIO.stdOut, typ ty, 50)
  end

end;
```

either a type variable or any type enclosed in parentheses:

$$\begin{aligned} Type &= Atom \texttt{ -> } Type \\ &\mid Atom \\ Atom &= \texttt{'} \; Id \\ &\mid \texttt{(} \; Type \; \texttt{)} \end{aligned}$$

This grammar treats → as an infix operator that associates to the right. It interprets `'a->'b->'c` as `'a->('b->'c)` rather than `('a->'b)->'c` because `'a -> 'b` is not an *Atom*.

The structure contains two `local` declarations, one for parsing and one for pretty printing. Each declares mutually recursive functions *typ* and *atom* corresponding to the grammar. Opening structure *LamParsing* makes its operations available at top level; recall that the infix directives were global.

Parsing of types. Using the top-down parsing operators, the function definitions in the parser are practically identical to the grammar rules. The operator >>, which applies a function to a parser's result, appears three times. Function *typ* uses >> to apply *makeFun* to the result of the first grammar rule, combining two types to form a function type. Using `$--` in the rule prevents the arrow symbol from being returned as a constituent of the phrase.

Both cases of *atom* involve >>, with two mysterious functions. During parsing of the type variable `'a`, in the first case, >> applies *Var o op^* to the pair (`"'"`, `"a"`). This function consists of *Var* composed with string concatenation; it concatenates the strings to `"'a"` and returns the type *Var* `"'a"`.

In the second case of *atom*, parsing the phrase `(` *Type* `)` calls the function `#1`, which selects the first component of its argument. Here it takes the pair (*ty*, `")"`) and yields *ty*. Had we not used `$--` to parse the left parenthesis, we should have needed the even more mysterious function `(#2 o #1)`.

The parsing functions mention the argument *toks* to avoid looping (like *repeat* above) and because a `fun` declaration must mention an argument.

Pretty printing of types. The same mutual recursion works for displaying as for parsing. Functions *typ* and *atom* both convert a type into a symbolic expression for the pretty printer, but *atom* encloses its result in parentheses unless it is just an identifier. Parentheses appear only when necessary; too many parentheses are confusing.

The functions *blo*, *str* and *brk* of structure *Pretty* are used in typical fashion to describe blocks, strings and breaks. Function *atom* calls *blo* with an inden-

tation of one to align subsequent breaks past the left parenthesis. Function *typ* calls *blo* with an indentation of zero, since it includes no parentheses; after the string `" ->"`, it calls *brk* 1 to make a space or a line break.

The function *pr* writes to the terminal (stream *TextIO.stdOut*), with a right margin of fifty.

Trying some examples. We can enter types, note their internal representations (as values of *Type.t*) after parsing, and check that they are displayed correctly:

```
Type.read"'a->'b->'c";
> Con ("->", [Var "'a",
>             Con ("->", [Var "'b", Var "'c"])])
> : Type.t
Type.pr it;
> 'a -> 'b -> 'c
Type.read"('a->'b)->'c";
> Con ("->", [Con ("->", [Var "'a", Var "'b"]),
>             Var "'c"])
> : Type.t
Type.pr it;
> ('a -> 'b) -> 'c
```

Our parsing of types is naïve. A string of the form `(` *Type* `)` must be parsed twice. The first grammar rule for *Type* fails: there is no `->` token after the right parenthesis. The second grammar rule parses it successfully as an *Atom*. We could modify the grammar to remove the repeated occurrence of *Atom*.

More on parsing. LR parsing is the method of choice for complicated grammars, like those of programming languages. This bottom-up technique is reliable, efficient and general; it supports good error recovery. LR parsers are not written by hand but generated using a tool such as Yacc (yet another compiler-compiler). The tool accepts a grammar, constructs parsing tables and outputs the parser in source form. Each syntax rule may be augmented with a *semantic action*: code to be executed whenever that rule applies. Most parser generators are based upon the C language.

ML-Yacc (Tarditi and Appel, 1994) uses ML for semantic actions and for the generated parser. ML-Yacc is fairly complicated to set up, but it is worth considering for any non-trivial grammar. You must supply ML-Yacc with a lexical analyser, which might be hand-coded or generated using a tool such as ML-Lex (Appel *et al.*, 1994).

The functional approach to top-down parsing has been understood for a long time. Burge (1975) contains one of the earliest published descriptions, including the use of lazy lists for backtracking. Reade (1989) gives a more modern account. Frost and Launchbury (1989) use the method to parse a subset of English for a question-answering system. Tobias Nipkow, who suggested `!!` and `$--`, has used the approach to parse Isabelle theory files.

Aho *et al.* (1986) describes lexical analysis and parsing extremely well. It covers both the top-down approach implemented here using functions, and the bottom-up approach that underlies ML-Yacc.

Exercise 9.9 Implement parsing and pretty printing of arbitrary type constructors. First, define a grammar for ML's postfix syntax, as in the examples

```
'c list list              (string,int) sum
('a -> 'b) list           'a list -> 'b list
```

Parentheses are optional when a type constructor is applied to one argument not involving the arrow; thus `'a -> 'b list` stands for `'a -> (('b) list)` rather than `('a -> 'b) list`.

Exercise 9.10 Use the parsing primitives, or alternatively ML-Yacc, to implement a parser for propositions — type *prop* of Section 4.17.

Introducing the λ-calculus

Turing machines, recursive functions and register machines are formal models of computation. The λ-calculus, developed by Alonzo Church, is one of the earliest models and perhaps the most realistic. It can express computations over pairs, lists, trees (even infinite ones) and higher-order functions. Most functional languages are nothing more than elaborated forms of the λ-calculus, and their implementations are founded in λ-calculus theory.

Church's thesis asserts that the effectively computable functions are precisely those functions that can be computed in the λ-calculus. Because 'effective' is a vague notion, Church's thesis cannot be proved, but the λ-calculus is known to have the same power as the other models of computation. Functions coded in these models can be computed effectively, given sufficient time and space, and nobody has exhibited a computable function that cannot be coded in these models.

9.5 λ-terms and λ-reductions

The λ-calculus is a simple formal theory of functions. Its terms, called λ-terms, are constructed recursively from variables x, y, z, ... and other λ-terms. Let t, u, ... stand for arbitrary λ-terms. They may take one of three

forms:

$$\begin{array}{rl} x & \text{a variable} \\ (\lambda x.t) & \text{functional } \textbf{\textit{abstraction}} \\ (t\ u) & \text{function } \textbf{\textit{application}} \end{array}$$

A term t_1 is a ***subterm*** of t_2 if t_1 is contained in t_2 or is identical to it. For instance, y is a subterm of $(\lambda z.(z\ y))$.

In the abstraction $(\lambda x.t)$, we call x the ***bound variable*** and t the ***body***. Every occurrence of x in t is ***bound*** by the abstraction. Conversely, an occurrence of a variable y is ***free*** if it is not bound — if it is not contained within the body of some abstraction $(\lambda y.u)$. For example, x occurs bound and y occurs free in $(\lambda z.(\lambda x.(y\ x))$. From now on, let a, b, c, ... denote free variables.

The names of bound variables have little significance. If they are renamed consistently in an abstraction, the new abstraction is essentially the same as the old. This principle is known throughout mathematics. In the integral $\int_a^b f(x)\,dx$, the variables a and b are free while x is bound. In the product $\prod_{k=0}^{n} p(k)$, the variable n is free while k is bound.

The abstraction $(\lambda x.t)$ represents the function f with $f(x) = t$ for all x. Applying $(\lambda x.t)$ to an argument u yields the term that results when u is substituted for all free occurrences of x in t. Write the result of this substitution as $t[u/x]$. Substitution involves some delicate points, but let us leave them for later.

λ-conversions. These are rules for transforming a λ-term while preserving its intuitive meaning. Conversions should not be confused with equations such as $x + y = y + x$, which are statements about known arithmetic operations. The λ-calculus is not concerned with previously existing mathematical objects. The λ-terms themselves are the objects, and the λ-conversions are symbolic transformations upon them.

Most important is ***β-conversion***, which transforms a function application by substituting the argument into the body:

$$((\lambda x.t)u) \Rightarrow_\beta t[u/x]$$

In this example, the argument is $(g\ a)$:

$$((\lambda x.((f\ x)x))(g\ a)) \Rightarrow_\beta ((f(g\ a))(g\ a))$$

Here is an example of two successive β-conversions:

$$((\lambda z.(z\ a))(\lambda x.x)) \Rightarrow_\beta ((\lambda x.x)a) \Rightarrow_\beta a$$

An ***α-conversion*** renames the bound variable in an abstraction:

$$(\lambda x.t) \Rightarrow_\alpha (\lambda y.t[y/x])$$

The abstraction over x is transformed into an abstraction over y, and x is replaced by y. Examples:

$$(\lambda x.a\ x) \Rightarrow_\alpha (\lambda y.a\ y)$$
$$(\lambda x.(x(\lambda y.(y\ x)))) \Rightarrow_\alpha (\lambda z.(z(\lambda y.(y\ z))))$$

Two λ-terms are ***congruent*** if one can be transformed into the other using α-conversions (possibly applied to subterms). Intuitively, we may regard congruent terms as being the same, renaming bound variables whenever necessary. Free variables are significant, however; thus a is distinct from b while $(\lambda x.x)$ is congruent to $(\lambda y.y)$.

Notation. Nested abstractions and applications can be abbreviated:

$$(\lambda x_1.(\lambda x_2.\ldots(\lambda x_n.t)\ldots)) \quad \text{as} \quad (\lambda x_1 x_2 \ldots x_n.t)$$
$$(\ldots(t_1\,t_2)\ldots t_n) \quad \text{as} \quad (t_1\,t_2\ldots t_n)$$

The outer parentheses are dropped when the term is not enclosed in another term or is the body of an abstraction. For example,

$(\lambda x.(x(\lambda y.(y\ x))))$ can be written as $\lambda x.x(\lambda y.y\ x)$.

Reduction to normal form. A ***reduction step*** $t \Rightarrow u$ transforms t to u by applying a β-conversion to any subterm of t. If a term admits no reductions then it is in ***normal form***. To ***normalize*** a term means to apply reductions until a normal form is reached.

Some terms can be reduced in more than one way. The ***Church-Rosser Theorem*** states that different reduction sequences starting from a term can always be brought together again. In particular, no two sequences of reductions can reach distinct (non-congruent) normal forms. The normal form of a term can be regarded as its value; it is independent of the order in which reductions are performed.

For instance, $(\lambda x.a\ x)((\lambda y.b\ y)c)$ has two different reduction sequences, both leading to the same normal form. The affected subterm is underlined at each step:

$$\underline{(\lambda x.a\ x)((\lambda y.b\ y)c)} \Rightarrow a(\underline{(\lambda y.b\ y)c}) \Rightarrow a(b\ c)$$
$$(\lambda x.a\ x)(\underline{(\lambda y.b\ y)c}) \Rightarrow \underline{(\lambda x.a\ x)(b\ c)} \Rightarrow a(b\ c)$$

Many λ-terms have no normal form. For instance, $(\lambda x.x\ x)(\lambda x.x\ x)$ reduces to itself by β-conversion. Any attempt to normalize this term must fail to terminate:

$$\underline{(\lambda x.x\ x)(\lambda x.x\ x)} \Rightarrow \underline{(\lambda x.x\ x)(\lambda x.x\ x)} \Rightarrow \cdots$$

A term t can have a normal form even though certain reduction sequences never terminate. Typically, t contains a subterm u that has no normal form, but u can be erased by a reduction step. For example, the reduction sequence

$$\underline{(\lambda y.a)((\lambda x.x\ x)(\lambda x.x\ x))} \Rightarrow a$$

reaches normal form directly, erasing the term $(\lambda x.x\ x)(\lambda x.x\ x)$. This corresponds to a ***call-by-name*** treatment of functions: the argument is not evaluated but simply substituted into the body of the function. Attempting to normalize the argument generates a nonterminating reduction sequence:

$$(\lambda y.a)(\underline{(\lambda x.x\ x)(\lambda x.x\ x)}) \Rightarrow (\lambda y.a)(\underline{(\lambda x.x\ x)(\lambda x.x\ x)}) \Rightarrow \cdots$$

Evaluating the argument prior to substitution into the function body corresponds to a ***call-by-value*** treatment of function application. In this example, the call-by-value strategy never reaches the normal form. The reduction strategy corresponding to call-by-name evaluation always reaches a normal form if one exists.

You may well ask, in what sense is $\lambda x.x\ x$ a function? It can be applied to any object and applies that object to itself! In classical mathematics, a function can only be defined over some previously existing set of values. The λ-calculus does not deal with functions as they are classically understood.[1]

9.6 *Preventing variable capture in substitution*

Substitution must be defined carefully: otherwise the conversions could go wrong. For instance, the term $\lambda x\ y.y\ x$ ought to behave like a curried function that, when applied to arguments t and u, returns $u\ t$ as its result. For all λ-terms t and u, we should expect to have the reductions

$$\underline{(\lambda x\ y.y\ x)t}\ u \Rightarrow \underline{(\lambda y.y\ t)u} \Rightarrow u\ t.$$

The following reduction sequence is certainly wrong:

$$\underline{(\lambda x\ y.y\ x)y}\ b \Rightarrow \underline{(\lambda y.y\ y)b} \Rightarrow b\ b \quad ???$$

[1] Dana Scott has constructed models in which every abstraction, including $\lambda x.x\ x$, denotes a function (Barendregt, 1984). However, this chapter regards the λ-calculus from a purely syntactic point of view.

The β-conversion of $(\lambda x\ y.y\ x)y$ to $\lambda y.y\ y$ is incorrect because the free variable y has become bound. The substitution has ***captured*** this free variable. By first renaming the bound variable y to z, the reduction can be performed safely:

$$\underline{(\lambda x\ z.z\ x)y}\ b \Rightarrow \underline{(\lambda z.z\ y)b} \Rightarrow b\ y$$

In general, the substitution $t[u/x]$ will not capture any variables provided no free variable of u is bound in t.

If bound variables are represented literally, then substitution must sometimes rename bound variables of t to avoid capturing free variables. Renaming is complicated and can be inefficient. It is essential that the new names do not appear elsewhere in the term. Preferably they should be similar to the names that they replace; nobody wants to see a variable called `G6620094`.

A name-free representation. Changing the representation of λ-terms can simplify the substitution algorithm. The name x of a bound variable serves only to match each occurrence of x with its binding λx so that reductions can be performed correctly. If these matches can be made by other means, then the names can be abolished.

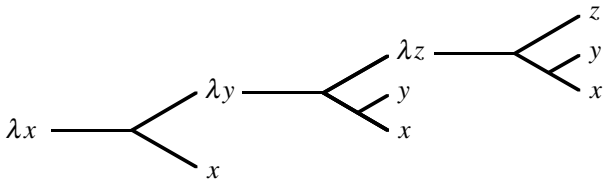
We can achieve this using the nesting depth of abstractions. Each occurrence of a bound variable is represented by an index, giving the number of abstractions lying between it and its binding abstraction. Two λ-terms are congruent — differing only by α-conversions — if and only if their name-free representations are identical.

In the name-free notation, no variable name appears after the λ symbol and bound variable indices appear as numbers. The first occurrence of x in the body of $\lambda x.(\lambda y.x)x$ is represented by 1 because it is enclosed in an abstraction over y. The second occurrence of x is not enclosed in any other abstraction and is represented by 0. Therefore the name-free representation of $\lambda x.(\lambda y.x)x$ is $\lambda.(\lambda.1)0$.

Here is a term where the bound variables occur at several nesting depths:

$$\lambda x.x(\lambda y.x\ y(\lambda z.x\ y\ z))$$

Viewing the term as a tree emphasizes its nesting structure:



In the name-free notation, the three occurrences of x are represented by 0, 1 and 2:

$$\lambda.0(\lambda.1\,0(\lambda.2\,1\,0))$$

Operations such as abstraction and substitution are easily performed in the name-free representation. It is a good data structure for variable binding, but is unreadable as a notation. The original variable names should be retained for later display, so that the user sees the traditional notation.

Abstraction. Suppose that t is a λ-term that we would like to abstract over all free occurrences of the variable x, constructing the abstraction $\lambda x.t$. Take, for instance, $x(\lambda y.a\ x\ y)$, which in the name-free notation is

$$x(\lambda.a\ x\ 0).$$

To bind all occurrences of x, we must replace them by the correct indices, here 0 and 1, and insert a λ symbol:

$$\lambda.0(\lambda.a\ 1\ 0)$$

This can be performed by a recursive function on terms that keeps count of the nesting depth of abstractions. Each occurrence of x is replaced by an index equal to its depth.

Substitution. To perform the β-conversion

$$(\lambda x.t)u \Rightarrow_\beta t[u/x],$$

the term t must be recursively transformed, replacing all the occurrences of x by u. In the name-free notation, x could be represented by several different indices. The index is initially 0 and increases with the depth of abstractions in t. For instance, the conversion

$$(\lambda x.x(\lambda y.a\ x\ y))b \Rightarrow_\beta b(\lambda y.a\ b\ y)$$

becomes

$$(\lambda.0(\lambda.a\ 1\ 0))b \Rightarrow_\beta b(\lambda.a\ b\ 0)$$

in the name-free notation. Observe that x has index 0 in the outer abstraction and index 1 in the inner one.

Performing β-conversion on a subterm $(\lambda x.t)u$ is more complex. The argument u may contain variables bound outside, namely indices with no matching abstraction in u. These indices must be increased by the current nesting depth

before substitution into t; this ensures that they refer to the same abstractions afterwards.

For instance, in

$$\lambda z.(\lambda x.x(\lambda y.x))(a\ z) \Rightarrow_\beta \lambda z.a\ z(\lambda y.a\ z),$$

the argument $a\ z$ is substituted in two places, one of which lies in the scope of λy. In the name-free approach, $a\ z$ receives two different representations:

$$\lambda.(\lambda.0(\lambda.1))(a\ 0) \Rightarrow_\beta \lambda.a\ 0(\lambda.a\ 1)$$

Exercise 9.11 Show all the reduction sequences for normalizing the term

$$(\lambda f.f(f\ a))((\lambda x.x\ x)((\lambda y.y)(\lambda y.y))).$$

Exercise 9.12 For each term, show its normal form or demonstrate that it has none:

$$(\lambda f\ x\ y.f\ x\ y)(\lambda u\ v.u)$$
$$(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$
$$(\lambda x\ y.y\ x)(\lambda x.f(f\ x))(\lambda x.f(f(f(f\ x))))$$
$$(\lambda x.x\ x)(\lambda x.x)$$

Exercise 9.13 Give the name-free representations of the following terms:

$$\lambda x\ y\ z.x\ z(y\ z)$$
$$\lambda x\ y.(\lambda z.x\ y\ z)y\ x$$
$$\lambda f.(\lambda x.f(\lambda y.x\ x\ y))(\lambda x.f(\lambda y.x\ x\ y))$$
$$(\lambda p\ x\ y.p\ x\ y)(\lambda x\ y.y)\ a\ b$$

Exercise 9.14 Consider a representation of λ-terms that designates bound variables internally by unique integers. Give algorithms for constructing λ-terms and for performing substitution.

Representing λ-terms in ML

Implementing the λ-calculus in ML is straightforward under the name-free representation. The following sections present ML programs for abstraction and substitution, and for parsing and pretty printing λ-terms.

We shall need *StringDict*, the dictionary structure declared in Section 7.10. It allows us to associate any information, in this case λ-terms, with strings. We can evaluate λ-terms with respect to an ***environment*** of defined identifiers.

9.7 The fundamental operations

Here is the signature for the name-free representation:

```
signature LAMBDA =
  sig
  datatype t = Free   of string
             | Bound  of int
             | Abs    of string * t
             | Apply  of t * t;
  val abstract  : int -> string -> t -> t
  val absList   : string list * t -> t
  val applyList : t * t list -> t
  val subst     : int -> t -> t -> t
  val inst      : t StringDict.t -> t -> t
  end;
```

Datatype t comprises free variables (as strings), bound variables (as indices), abstractions and applications. Each Abs node stores the bound variable name for use in printing.

Calling $abstract\ i\ b\ t$ converts each occurrence of the free variable b in t to the index i (or a greater index within nested abstractions). Usually $i = 0$ and the result is immediately enclosed in an abstraction to match this index. Recursive calls over abstractions in t have $i > 0$.

Calling $absList([x_1, \ldots, x_n], t)$ creates the abstraction $\lambda x_1 \ldots x_n.t$.

Calling $applyList(t, [u_1, \ldots, u_n])$ creates the application $t\ u_1\ \ldots\ u_n$.

Calling $subst\ i\ u\ t$ substitutes u for the bound variable index i in t. Usually $i = 0$ and t is the body of an abstraction in the β-conversion $(\lambda x.t)u$. The case $i > 0$ occurs during recursive calls over abstractions in t. All indices exceeding i are decreased by one to compensate for the removal of that index.

Calling $inst\ env\ t$ copies t, replacing all occurrences of variables defined in env by their definitions. The dictionary env represents an environment, and $inst$ expands all the definitions in a term. This process is called ***instantiation***. Definitions may refer to other definitions; instantiation continues until defined variables no longer occur in the result.

Signature *LAMBDA* is concrete, revealing all the internal details. Many values of type t are ***improper***: they do not correspond to real λ-terms because they contain unmatched bound variable indices. No term has the representation $Bound\ i$, for any i. Moreover, $abstract$ returns improper terms and $subst$ expects them. An abstract signature for the λ-calculus would provide operations upon λ-terms themselves, hiding their representation.

Structure $Lambda$ (Figure 9.5) implements the signature. Function $shift$ is private to the structure because it is called only by $subst$. Calling $shift\ i\ d\ u$

Figure 9.5 *The name-free representation of λ-terms*

```
structure Lambda : LAMBDA =
  struct
  datatype t = Free   of string
             | Bound  of int
             | Abs    of string * t
             | Apply  of t * t;

  (*Convert occurrences of b to bound index i in a term*)
  fun abstract i b (Free a)      = if a=b then Bound i else Free a
    | abstract i b (Bound j)     = Bound j
    | abstract i b (Abs(a,t))    = Abs(a, abstract (i+1) b t)
    | abstract i b (Apply(t,u)) = Apply(abstract i b t, abstract i b u);

  (*Abstraction over several free variables*)
  fun absList (bs,t) = foldr (fn (b,u) => Abs(b, abstract 0 b u)) t bs;

  (*Application of t to several terms*)
  fun applyList (t0,us) = foldl (fn (u,t) => Apply(t,u)) t0 us;

  (*Shift a term's non-local indices by i*)
  fun shift 0 d u            = u
    | shift i d (Free a)     = Free a
    | shift i d (Bound j)    = if j>=d then Bound(j+i) else Bound j
    | shift i d (Abs(a,t))   = Abs(a, shift i (d+1) t)
    | shift i d (Apply(t,u)) = Apply(shift i d t, shift i d u);

  (*Substitute u for bound variable i in a term t*)
  fun subst i u (Free a)       = Free a
    | subst i u (Bound j)      =
          if j<i then          Bound j          (*locally bound*)
          else if j=i then     shift i 0 u
          else (*j>i*)         Bound(j-1)       (*non-local to t*)
    | subst i u (Abs(a,t))     = Abs(a, subst (i+1) u t)
    | subst i u (Apply(t1,t2)) = Apply(subst i u t1, subst i u t2);

  (*Substitution for free variables*)
  fun inst env (Free a)        = (inst env (StringDict.lookup(env,a))
                                  handle StringDict.E _ => Free a)
    | inst env (Bound i)       = Bound i
    | inst env (Abs(a,t))      = Abs(a, inst env t)
    | inst env (Apply(t1,t2)) = Apply(inst env t1, inst env t2);
  end;
```

adds i to all the unmatched indices j in u such that $j \geq d$. Initially $d = 0$ and d is increased in recursive calls over abstractions in u. Before substituting some term u into another term, any unmatched indices in u must be shifted.

Function $inst$ substitutes for free variables, not bound variables. It expects to be given proper λ-terms having no unmatched indices. It therefore does not keep track of the nesting depth or call $shift$.

Exercise 9.15 Explain the use of the fold functionals in the declarations of *absList* and $applyList$.

Exercise 9.16 Declare a signature for the λ-calculus that hides its internal representation. It should specify predicates to test whether a λ-term is a variable, an abstraction or an application, and specify functions for abstraction and substitution. Sketch the design of two structures, employing two different representations of λ-terms, that would have this signature.

9.8 *Parsing λ-terms*

In order to apply the parser and pretty printer, we require a grammar for λ-terms, including the abbreviations for nested abstractions and applications. The following grammar distinguishes between ordinary terms and atomic terms. A per cent sign (%) serves as the λ symbol:

$$\begin{aligned} Term &= \texttt{\%}\ Id\ Id^*\ Term \\ &\mid Atom\ Atom^* \\ Atom &= Id \\ &\mid \texttt{(}\ Term\ \texttt{)} \end{aligned}$$

Note that $phrase^*$ stands for zero or more repetitions of $phrase$. A term consisting of several Atoms in a row, such as $a\ b\ c\ d$, abbreviates the nested application $(((a\ b)c)d)$. A more natural grammar would define the phrase class $Applic$:

$$\begin{aligned} Applic &= Atom \\ &\mid Applic\ Atom \end{aligned}$$

Then we could replace the $Atom\ Atom^*$ in $Term$ by $Applic$. But the second grammar rule for $Applic$ is left-recursive and would cause our parser to loop. Eliminating this left recursion in the standard way yields our original grammar.

Structure $ParseTerm$ (Figure 9.6) uses structures $Parse$ and $Lambda$, containing the parser and the λ-term operations, to satisfy signature *PARSE_TERM*:

Figure 9.6 *The λ-calculus parser*

```
structure ParseTerm : PARSE_TERM =
  struct

  fun makeLambda ((b,bs),t) = Lambda.absList (b::bs, t);

  open LamParsing

  fun term toks =
    (   "%" $-- id -- repeat id -- "." $-- term >> makeLambda
     || atom -- repeat atom                     >> Lambda.applyList
    ) toks
  and atom toks =
    (   id                                      >> Lambda.Free
     || "(" $-- term -- $")"                    >> #1
    ) toks;
  val read = reader term;

  end;
```

```
signature PARSE_TERM =
  sig  val read: string -> Lambda.t  end;
```

The structure's only purpose is to parse λ-terms. Its signature specifies just one component: the function *read* converts a string to a λ-term. Its implementation is straightforward, using components *absList* and *applyList* of structure *Lambda*.

Exercise 9.17 In function *makeLambda*, why does the argument pattern have the form it does?

Exercise 9.18 What is the result of parsing `"%x x.x(%x x.x)"`?

9.9 *Displaying λ-terms*

Structure *DisplayTerm* (Figure 9.7 on page 390) implements pretty printing for λ-terms. Using structures *Pretty* and *Lambda* — the pretty printer and the term operations — it satisfies signature *DISPLAY_TERM*:

```
signature DISPLAY_TERM =
  sig
  val rename   : string list * string -> string
  val stripAbs : Lambda.t -> string list * Lambda.t
  val pr       : Lambda.t -> unit
  end;
```

The signature specifies several components:

- $rename([a_1, \ldots, a_n],\ a)$ appends prime (`'`) characters to a to make it differ from each of $a_1, \ldots, a_n$.
- *stripAbs* analyses an abstraction into its bound variables and body, as described below.
- Calling $pr\ t$ prints the term t at the terminal.

Even with the name-free representation, bound variables may have to be renamed when a term is displayed. The normal form of $(\lambda xy.x)y$ is shown as `%y'. y`, not as `%y. y`. Function *stripAbs* and its auxiliary function *strip* handle abstractions. Given $\lambda x_1 \ldots x_m.t$, the bound variables are renamed to differ from all free variables in t. The new names are substituted into t as free variables. Thus, all indices are eliminated from a term as it is displayed.

The mutually recursive functions $term$, $applic$ and $atom$ prepare λ-terms for pretty printing. A *Free* variable is displayed literally. A $Bound$ variable index should never be encountered unless it has no matching Abs node (indicating that the term is improper). For an Abs node, the bound variables are renamed; then *foldleft* joins them into a string, separated by spaces. An $Apply$ node is displayed using $applic$, which corresponds to the grammatical phrase $Applic$ mentioned in the previous section. Finally, $atom$ encloses a term in parentheses unless it is simply an identifier.

Exercise 9.19 How will the normal form of $(\lambda x\ y.x)(\lambda y.y)$ be displayed? Modify $DisplayTerm$ to ensure that, when a term is displayed, no variable name is bound twice in overlapping scopes.

Exercise 9.20 Terms can be displayed without substituting free variables for bound variables. Modify $DisplayTerm$ to keep a list of the variables bound in the abstractions enclosing the current subterm. To display the term $Bound\ i$, locate the ith name in the list.

Figure 9.7 *The λ-calculus pretty printer*

```
structure DisplayTerm : DISPLAY_TERM =
  struct

  (*Free variable in a term*)
  fun vars (Lambda.Free a)       = [a]
    | vars (Lambda.Bound i)      = []
    | vars (Lambda.Abs(a,t))     = vars t
    | vars (Lambda.Apply(t1,t2)) = vars t1 @ vars t2;

  (*Rename variable "a" to avoid clashes*)
  fun rename (bs,a) =
      if List.exists (fn x => x=a) bs then rename (bs, a ^ "'") else a;

  (*Remove leading lambdas; return bound variable names*)
  fun strip (bs, Lambda.Abs(a,t)) =
        let val b = rename (vars t, a)
        in  strip (b::bs, Lambda.subst 0 (Lambda.Free b) t)
        end
    | strip (bs, u)                = (rev bs, u);

  fun stripAbs t = strip ([],t);

  fun spaceJoin (b,z) = " " ^ b ^ z;

  fun term (Lambda.Free a)       = Pretty.str a
    | term (Lambda.Bound i)      = Pretty.str "??UNMATCHED INDEX??"
    | term (t as Lambda.Abs _)   =
        let val (b::bs,u) = stripAbs t
            val binder    = "%" ^ b ^ (foldr spaceJoin ". " bs)
        in  Pretty.blo(0, [Pretty.str binder, term u])
        end
    | term t                     = Pretty.blo(0, applic t)
  and applic (Lambda.Apply(t,u)) = applic t @ [Pretty.brk 1, atom u]
    | applic t                   = [atom t]
  and atom (Lambda.Free a)       = Pretty.str a
    | atom t                     = Pretty.blo(1, [Pretty.str"(",
                                                  term t,
                                                  Pretty.str")"]);

  fun pr t = Pretty.pr (TextIO.stdOut, term t, 50);
  end;
```

The λ-calculus as a programming language

Despite its simplicity, the λ-calculus is rich enough to model the full range of functional programming. Data structures such as pairs and lists can be processed under either call-by-value or call-by-name evaluation strategies. After a brief discussion of these topics we shall demonstrate them using ML. First, we must make some definitions.

Write $t \Rightarrow^* u$ whenever t can be transformed into u by zero or more reduction steps. If u is in normal form then $t \Rightarrow^* u$ can be viewed as evaluating t to obtain the result u. Not every evaluation strategy will succeed in finding this normal form.

Write $t_1 = t_2$ whenever there is some term u (not necessarily in normal form!) such that $t_1 \Rightarrow^* u$ and $t_2 \Rightarrow^* u$. If $t_1 = t_2$ then both terms have the same normal form, if any. Viewing normal forms as values, $t_1 = t_2$ means that t_1 and t_2 have the same value.

Write $a \equiv t$, where a is a free variable, to mean 'a abbreviates t by definition.'

9.10 *Data structures in the λ-calculus*

We now consider how to encode boolean values, ordered pairs, natural numbers and lists. The codings given below are arbitrary; all that matters is that the data structures and their operations satisfy certain standard properties. An encoding of the booleans must define the truth values $true$ and $false$ and the conditional operator if as λ-terms, satisfying (for all t and u)

$$if\ true\ t\ u = t$$
$$if\ false\ t\ u = u$$

Once we have two distinct truth values and the conditional operator, we can define negation, conjunction and disjunction. Analogously, the ML compiler may represent $true$ and $false$ by any bit patterns provided the operations behave properly.

The booleans. The booleans can be coded by defining

$$true \equiv \lambda x\ y.x$$
$$false \equiv \lambda x\ y.y$$
$$if \equiv \lambda p\ x\ y.p\ x\ y$$

The necessary properties are easily verified. For instance:

$$\begin{aligned} \mathit{if}\ \mathit{true}\ t\ u &\equiv (\lambda p\ x\ y.p\ x\ y)\mathit{true}\ t\ u \\ &\Rightarrow (\lambda x\ y.\mathit{true}\ x\ y)t\ u \\ &\Rightarrow (\lambda y.\mathit{true}\ t\ y)u \\ &\Rightarrow \mathit{true}\ t\ u \\ &\Rightarrow (\lambda y.t)u \\ &\Rightarrow t \end{aligned}$$

This establishes $\mathit{if}\ \mathit{true}\ t\ u \Rightarrow^* t$ and therefore $\mathit{if}\ \mathit{true}\ t\ u = t$.

Ordered pairs. An encoding must specify a function pair (to construct pairs) and projection functions fst and snd (to select the components of a pair). The usual encoding is

$$\begin{aligned} \mathit{pair} &\equiv \lambda x\ y\ f.f\ x\ y \\ \mathit{fst} &\equiv \lambda p.p\ \mathit{true} \\ \mathit{snd} &\equiv \lambda p.p\ \mathit{false} \end{aligned}$$

where true and false are defined as above. These reductions, and the corresponding equations, are easily verified for all t and u:

$$\begin{aligned} \mathit{fst}(\mathit{pair}\ t\ u) &\Rightarrow^* t \\ \mathit{snd}(\mathit{pair}\ t\ u) &\Rightarrow^* u \end{aligned}$$

The natural numbers. Of the several known encodings of the natural numbers, Church's is the most elegant. Underlined numbers $\underline{0}$, $\underline{1}$, ..., denote the ***Church numerals***:

$$\begin{aligned} \underline{0} &\equiv \lambda f\ x.x \\ \underline{1} &\equiv \lambda f\ x.f\ x \\ \underline{2} &\equiv \lambda f\ x.f(f\ x) \\ &\vdots \\ \underline{n} &\equiv \lambda f\ x.f^n(x) \end{aligned}$$

Here $f^n(x)$ abbreviates $\underbrace{f(\cdots(f}_{n \text{ times}}\ x)\cdots)$.

The function suc computes the successor of a number, and $iszero$ tests whether a number equals zero:

$$\begin{aligned} suc &\equiv \lambda n\ f\ x.n\ f\ (f\ x) \\ iszero &\equiv \lambda n.n(\lambda x.false)true \end{aligned}$$

It is not difficult to verify the following reductions, where $\underline{n}$ is an arbitrary Church numeral:

$$\begin{aligned} suc\ \underline{n} &\Rightarrow^* \underline{n+1} \\ iszero\ \underline{0} &\Rightarrow^* true \\ iszero(suc\ \underline{n}) &\Rightarrow^* false \end{aligned}$$

Church numerals allow wonderfully succinct definitions of addition, multiplication and exponentiation:

$$\begin{aligned} add &\equiv \lambda m\ n\ f\ x.m\ f(n\ f\ x) \\ mult &\equiv \lambda m\ n\ f.m(n\ f) \\ expt &\equiv \lambda m\ n\ f\ x.n\ m\ f\ x \end{aligned}$$

These can be formally verified by induction, and their underlying intuitions are simple. Each Church numeral $\underline{n}$ is an operator to apply a function n times. Note that

$$add\ \underline{m}\ \underline{n}\ f\ x = f^m(f^n(x)) = f^{m+n}(x);$$

the others are understood similarly.

An encoding of the natural numbers must also specify a predecessor function pre such that

$$pre(suc\ \underline{n}) = \underline{n}$$

for all numbers $\underline{n}$. With Church numerals, computing $\underline{n}$ from $\underline{n+1}$ is complex (and slow!); given f and x, we must compute $f^n(x)$ from $g^{n+1}(y)$ for some g and y. A suitable g is a function on pairs such that $g(z, z') = (f(z), z)$ for all (z, z'); then

$$g^{n+1}(x, x) = (f^{n+1}(x), f^n(x))$$

and we take the second component. To formalize this, define $prefn$ to construct g. Then define the predecessor function pre and the subtraction func-

tion sub:

$$\begin{aligned} prefn &\equiv \lambda f\ p.pair(f(fst\ p))\ (fst\ p) \\ pre &\equiv \lambda n\ f\ x.snd(n(prefn\ f)(pair\ x\ x)) \\ sub &\equiv \lambda m\ n.n\ pre\ m \end{aligned}$$

For subtraction, $sub\ \underline{m}\ \underline{n} = pre^n(\underline{m})$; this computes the nth predecessor of $\underline{m}$.

Lists. Lists are encoded using pairing and the booleans. A non-empty list with head x and tail y is coded as $(false, (x, y))$. The empty list nil could be coded as $(true, true)$, but a simpler definition happens to work:

$$\begin{aligned} nil &\equiv \lambda z.z \\ cons &\equiv \lambda x\ y.pair\ false\ (pair\ x\ y) \\ null &\equiv fst \\ hd &\equiv \lambda z.fst(snd\ z) \\ tl &\equiv \lambda z.snd(snd\ z) \end{aligned}$$

The essential properties are easy to check for all t and u:

$$\begin{aligned} null\ nil &\Rightarrow^* true \\ null(cons\ t\ u) &\Rightarrow^* false \\ hd(cons\ t\ u) &\Rightarrow^* t \\ tl(cons\ t\ u) &\Rightarrow^* u \end{aligned}$$

A call-by-name evaluation reduces $hd(cons\ t\ u)$ to t without evaluating u, and can process infinite lists.

Exercise 9.21 Define an encoding of ordered pairs in terms of an arbitrary encoding of the booleans. Demonstrate it by encoding the booleans with $true = \lambda x\ y.y$ and $false = \lambda x\ y.x$.

Exercise 9.22 Verify, for all Church numerals $\underline{m}$ and $\underline{n}$:

$$\begin{aligned} iszero(suc\ \underline{n}) &= false \\ add\ \underline{m}\ \underline{n} &= \underline{m+n} \\ mult\ \underline{m}\ \underline{n} &= \underline{m \times n} \\ expt\ \underline{m}\ \underline{n} &= \underline{m^n} \end{aligned}$$

Exercise 9.23 Define an encoding of the natural numbers that has a simple predecessor function.

Exercise 9.24 Define an encoding of labelled binary trees.

Exercise 9.25 Write an ML function $numeral$ of type $int \to Lambda.t$ such that $numeral\ n$ constructs the Church numeral $\underline{n}$, for all $n \geq 0$.

9.11 *Recursive definitions in the λ-calculus*

There is a λ-term $fact$ that computes factorials of Church numerals by the recursion

$$fact\ n = if\ (iszero\ n)\ \underline{1}\ (mult\ n\ (fact(pre\ n))).$$

There is a λ-term $append$ that joins two lists by the recursion

$$append\ z\ w = if\ (null\ z)\ w\ (cons(hd\ z)(append(tl\ z)w)).$$

There even is a λ-term $inflist$ satisfying the recursion

$$inflist = cons\ MORE\ inflist,$$

encoding the infinite list $[MORE, MORE, \ldots]$.

Recursive definitions are encoded with the help of the λ-term Y:

$$Y \equiv \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Although the intuition behind Y is obscure, a simple calculation verifies that Y satisfies the ***fixed point property***

$$Y\ f = f(Y\ f)$$

for all λ-terms f. We can exploit this property to expand the body of a recursive object repeatedly. Define

$$\begin{aligned} fact &\equiv Y(\lambda g\ n.if\ (iszero\ n)\ \underline{1}\ (mult\ n\ (g(pre\ n)))) \\ append &\equiv Y(\lambda g\ z\ w.if\ (null\ z)\ w\ (cons(hd\ z)(g(tl\ z)w))) \\ inflist &\equiv Y(\lambda g.cons\ MORE\ g) \end{aligned}$$

In each definition, the recursive occurrence is replaced by the bound variable g in $Y(\lambda g.\ldots)$. Let us verify the recursion equation for $inflist$; the others are similar. The first and third lines hold by definition, while the second line uses the fixed point property:

$$\begin{aligned} inflist &\equiv Y(\lambda g.cons\ MORE\ g) \\ &= (\lambda g.cons\ MORE\ g)(Y(\lambda g.cons\ MORE\ g))) \\ &\equiv (\lambda g.cons\ MORE\ g)inflist \\ &\Rightarrow cons\ MORE\ inflist \end{aligned}$$

Recursive functions coded using Y execute correctly under call-by-name reduction. In order to use call-by-value reduction, recursive functions must be coded using a different fixed point operator (discussed below); otherwise execution will not terminate.

9.12 *The evaluation of λ-terms*

The structure *Reduce* (Figure 9.8) implements the call-by-value and call-by-name reduction strategies. Its signature is *REDUCE*:

```
signature REDUCE =
  sig
  val eval    : Lambda.t -> Lambda.t
  val byValue : Lambda.t -> Lambda.t
  val headNF  : Lambda.t -> Lambda.t
  val byName  : Lambda.t -> Lambda.t
  end;
```

The signature specifies four evaluation functions:

- *eval* evaluates a term using a call-by-value strategy resembling ML's. Its result need not be in normal form.
- *byValue* normalizes a term using call-by-value.
- *headNF* reduces a term to ***head normal form***, which is discussed below.
- *byName* normalizes a term using call-by-name.

Call-by-value. In ML, evaluating the abstraction `fn` x `=>` E does not evaluate E, for there is no general way to evaluate E without having a value for x. We have often exploited ML's treatment of abstractions, writing `fn () =>` E to delay the evaluation of E. This allows a kind of lazy evaluation.

The situation in the λ-calculus is different. The abstraction $\lambda x.(\lambda y.a\ y)x$ reduces to the normal form $\lambda x.a\ x$ with no question of whether x has a value. Even so, it is advantageous not to reduce the bodies of abstractions. This permits the delaying of evaluation, like in ML. It is essential for handling recursion.

The function *eval*, given the application $t_1\ t_2$, evaluates t_1 to u_1 and t_2 to u_2. (Assume these evaluations terminate.) If u_1 is the abstraction $\lambda x.u$ then *eval* calls itself on $u[u_2/x]$, substituting the value of the argument into the body; if u_1 is anything else then *eval* returns $u_1\,u_2$. Given an abstraction or variable, *eval* returns its argument unchanged. Although *eval* performs most of the work of reduction, its result may contain abstractions not in normal form.

The function *byValue* uses *eval* to reduce a term to normal form. It calls *eval* on its argument, then recursively scans the result to normalize the abstractions in it.

Figure 9.8 *Reduction of λ-terms*

```
structure Reduce : REDUCE =
  struct

  fun eval (Lambda.Apply(t1,t2)) =
                (case eval t1 of
                      Lambda.Abs(a,u) => eval(Lambda.subst 0 (eval t2) u)
                    | u1              => Lambda.Apply(u1, eval t2))
    | eval t                        = t;

  fun byValue t                     = bodies (eval t)
  and bodies (Lambda.Abs(a,t))      = Lambda.Abs(a, byValue t)
    | bodies (Lambda.Apply(t1,t2))  = Lambda.Apply(bodies t1, bodies t2)
    | bodies t                      = t;

  fun headNF (Lambda.Abs(a,t))      = Lambda.Abs(a, headNF t)
    | headNF (Lambda.Apply(t1,t2)) =
                (case headNF t1 of
                      Lambda.Abs(a,t) => headNF(Lambda.subst 0 t2 t)
                    | u1              => Lambda.Apply(u1, t2))
    | headNF t                      = t;

  fun byName t                      = args (headNF t)
  and args (Lambda.Abs(a,t))        = Lambda.Abs(a, args t)
    | args (Lambda.Apply(t1,t2))    = Lambda.Apply(args t1, byName t2)
    | args t                        = t;
  end;
```

Suppose that t equals $true$. When $eval$ is given $if\ \ t\ \ u_1\ \ u_2$, it evaluates both u_1 and u_2 although only u_1 is required. If this is the body of a recursive function then it will run forever, as discussed in Section 2.12. We should insert abstractions to delay evaluation. Choose any variable x and code the conditional expression as

$$(if\ \ t\ \ (\lambda x.u_1)\ \ (\lambda x.u_2))\ \ x.$$

Given this term, $eval$ will return $\lambda x.u_1$ as the result of the if and then apply it to x. Thus it will evaluate u_1 but not u_2. If t equals $false$ then only u_2 will be evaluated. Conditional expressions must be coded this way under call-by-value.

Recursive definitions encoded using Y fail under call-by-value because the evaluation of $Y\ \ f$ never terminates. Abstractions may be inserted into Y to delay evaluation. The operator

$$YV \equiv \lambda f.(\lambda x.f(\lambda y.x\ \ x\ \ y))(\lambda x.f(\lambda y.x\ \ x\ \ y))$$

enjoys the fixed point property and can express recursive functions for evaluation using $byValue$.

Call-by-name. A λ-term is in ***head normal form*** if, for $m \geq 0$ and $n \geq 0$, it can be viewed as follows:

$$\lambda x_1 \ldots x_m.x\ t_1 \ldots t_n.$$

The variable x may either be free or bound (one of $x_1, \ldots, x_m$).

Observe that the term's normal form (if it exists) must be

$$\lambda x_1 \ldots x_m.x\ u_1 \ldots u_n,$$

where u_i is the normal form of t_i for $i = 1, \ldots, n$. Head normal form describes a term's outer structure, which cannot be affected by reductions. We can normalize a term by computing its head normal form, then recursively normalizing the subterms $t_1, \ldots, t_n$. This procedure will reach the normal form if one exists, because every term that has a normal form also has a head normal form.

For example, the term $\lambda x.a((\lambda z.z)x)$ is in head normal form and its normal form is $\lambda x.a\ \ x$. A term not in head normal form can be viewed as

$$\lambda x_1 \ldots x_m.(\lambda x.t)\ t_1 \ldots t_n,$$

where $n > 0$. It admits a reduction in the leftmost part of the body. For instance, $(\lambda x\ \ y.y\ \ x)t$ reduces to the head normal form $\lambda y.y\ \ t$, for any term t. Many terms without a normal form have a head normal form; consider

$$Y = \lambda f.f(Y\ \ f).$$

A few terms, such as $(\lambda x.x\ x)(\lambda x.x\ x)$, lack even a head normal form. Such terms can be regarded as undefined.

The function $headNF$ computes the head normal form of $t_1\ t_2$ by recursively computing $headNF\ t_1$ and then, if an abstraction results, doing a β-conversion. The argument t_2 is not reduced before substitution; this is call-by-name.[2]

Function $byName$ normalizes a term by computing its $headNF$ and then normalizing the arguments of the outermost application. This achieves call-by-name reduction with reasonable efficiency.

Exercise 9.26 Show that $YV\ f = f(\lambda y.\, YV\ f\ y)$.

Exercise 9.27 Derive a head normal form of $Y\ \ Y$, or demonstrate that none exists.

Exercise 9.28 Derive a head normal form of $inflist$, or demonstrate that none exists.

Exercise 9.29 Describe how $byValue$ and $byName$ would compute the normal form of $fst(pair\ t\ u)$, for arbitrary λ-terms t and u.

9.13 *Demonstrating the evaluators*

To demonstrate our implementation of the λ-calculus, we create an environment $stdEnv$. It defines the λ-calculus encodings of the booleans, ordered pairs and so forth (Figure 9.9 on the next page). Function $insert$ of $StringDict$ adds a definition to a dictionary, provided that string is not already defined there.

Function $stdRead$ reads a term and instantiates it using $stdEnv$, expanding the definitions. Note that `"2"` expands to something large, derived from $suc(suc\ 0)$:

```
fun stdRead a = inst stdEnv (ParseLam.read a);
> val stdRead = fn : string -> term
DisplayTerm.pr (stdRead "2");
> (%n f x. n f (f x))
> ((%n f x. n f (f x)) (%f x. x))
```

This term could do with normalization. We define a function try such that $try\ evfn$ reads a term, applies $evfn$ to it, and displays the result. Using call-by-value, we reduce `"2"` to a Church numeral:

[2] $headNF$ exploits Proposition 8.3.13 of Barendregt (1984): if $t\ u$ has a head normal form then so does t.

Figure 9.9 *Constructing the standard environment*

```
fun insertEnv ((a,b),env) =
    StringDict.insert (env, a, ParseTerm.read b);

val stdEnv = foldl insertEnv StringDict.empty
[    (*booleans*)
 ("true", "%x y.x"),           ("false",  "%x y.y"),
 ("if", "%p x y. p x y"),
     (*ordered pairs*)
 ("pair", "%x y f.f x y"),
 ("fst", "%p.p true"),         ("snd", "%p.p false"),
     (*natural numbers*)
 ("suc", "%n f x. n f (f x)"),
 ("iszero", "%n. n (%x.false) true"),
 ("0", "%f x. x"),             ("1", "suc 0"),
 ("2", "suc 1"),               ("3", "suc 2"),
 ("4", "suc 3"),               ("5", "suc 4"),
 ("6", "suc 5"),               ("7", "suc 6"),
 ("8", "suc 7"),               ("9", "suc 8"),
 ("add",  "%m n f x. m f (n f x)"),
 ("mult", "%m n f. m (n f)"),
 ("expt", "%m n f x. n m f x"),
 ("prefn", "%f p. pair (f (fst p)) (fst p)"),
 ("pre",  "%n f x. snd (n (prefn f) (pair x x))"),
 ("sub",  "%m n. n pre m"),
     (*lists*)
 ("nil",  "%z.z"),
 ("cons", "%x y. pair false (pair x y)"),
 ("null", "fst"),
 ("hd", "%z. fst(snd z)"),     ("tl", "%z. snd(snd z)"),
    (*recursion for call-by-name*)
 ("Y", "%f. (%x.f(x x))(%x.f(x x))"),
 ("fact", "Y(%g n. if (iszero n) 1 (mult n (g (pre n))))"),
 ("append", "Y(%g z w.if (null z) w (cons (hd z) (g(tl z)w)))"),
 ("inflist", "Y(%z. cons MORE z)"),
     (*recursion for call-by-value*)
 ("YV", "%f. (%x.f(%y.x x y)) (%x.f(%y.x x y))"),
 ("factV",
  "YV (%g n.(if (iszero n) (%y.1) (%y.mult n (g (pre n))))y)")
];
```

```
fun try evfn = DisplayTerm.pr o evfn o stdRead;
> val try = fn : (term -> term) -> string -> unit
try Reduce.byValue "2";
> %f x. f (f x)
```

Call-by-value can perform simple arithmetic on Church numerals: $2 + 3 = 5$, $2 \times 3 = 6$, $2^3 = 8$:

```
try Reduce.byValue "add 2 3";
> %f x. f (f (f (f (f x))))
try Reduce.byValue "mult 2 3";
> %f x. f (f (f (f (f (f x)))))
try Reduce.byValue "expt 2 3";
> %f x. f (f (f (f (f (f (f (f x)))))))
```

The environment defines $factV$, which encodes a recursive factorial function using YV and with abstractions to delay evaluation of the arguments of the if. It works under call-by-value reduction, computing $3! = 6$:

```
try Reduce.byValue "factV 3";
> %f x. f (f (f (f (f (f x)))))
```

Call-by-name reduction can do the same computations as call-by-value can, and more. It handles recursive definitions involving Y and if, without needing any tricks to delay evaluation. Here, we *append* the lists $[FARE, THEE]$ and $[WELL]$:

```
try Reduce.byName
  "append (cons FARE (cons THEE nil)) (cons WELL nil)";
> %f. f (%x y. y)
>      (%f. f FARE
>           (%f. f (%x y. y)
>                (%f. f THEE
>                     (%f. f (%x y. y)
>                          (%f. f WELL (%z. z))))))
```

Let us take the head of the infinite list $[MORE, MORE, \ldots]$:

```
try Reduce.byName "hd inflist";
> MORE
```

Execution is extremely slow, especially with call-by-name. Computing $fact\ \underline{3}$ takes 330 msec, compared with 60 msec for $factV\ \underline{3}$. Computing $fact\ \underline{4}$ takes forty seconds! This should hardly be surprising when arithmetic employs unary notation and recursion works by copying. Even so, we have all the elements of functional programming.

With a little more effort, we can obtain a real functional language. Rather than encoding data structures in the pure λ-calculus, we can take numbers, arithmetic

operations and ordered pairs as primitive. Rather than interpreting the λ-terms, we can compile them for execution on an abstract machine. For call-by-value reduction, the SECD machine is suitable. For call-by-name reduction we can compile λ-terms into combinators and execute them by graph reduction. The design and implementation of a simple functional language makes a challenging project.

Further reading. M. J. C. Gordon (1988) describes the λ-calculus from the perspective of a computer scientist; he discusses ways of representing data and presents Lisp code for reducing and translating λ-expressions. Barendregt (1984) is the comprehensive reference on the λ-calculus. Boolos and Jeffrey (1980) introduce the theory of computability, including Turing machines, register machines and the general recursive functions.

N. G. de Bruijn (1972) developed the name-free notation for the λ-calculus, and used it in his AUTOMATH system (Nederpelt *et al.*, 1994). It is also used in Isabelle (Paulson, 1994) and in Hal, the theorem prover of Chapter 10.

Field and Harrison (1988) describe basic combinator reduction. Modern implementations of lazy evaluation use more sophisticated techniques (Peyton Jones, 1992).

Exercise 9.30 What is the result when $try\ \ Reduce\,.\,byName$ is applied to these strings?

```
"hd (tl (Y (%z. append (cons MORE (cons AND nil)) z)))"
"hd (tl (tl (Y (%g n. cons n (g (suc n))) 0)))"
```

Summary of main points

- Top-down parsers can be expressed in a natural way using higher-order functions.
- The λ-calculus is a theoretical model of computation with close similarities to functional programming.
- The name-free representation of variable binding is easily implemented on the computer.
- Data structures such as numbers and lists, with their operations, can be encoded as λ-terms.
- The λ-term Y encodes recursion by repeated copying.
- There exist call-by-value and call-by-name evaluation strategies for the λ-calculus.