Chapter 12

ASL: A Small Language

We present in this chapter a simple language: ASL (A Small Language). This language is basically the λ-calculus (the purely functional kernel of Caml) enriched with a conditional construct. The conditional must be a special construct, because our language will be submitted to call-by-value: thus, the conditional cannot be a function.

ASL programs are built up from numbers, variables, functional expressions (λ-abstractions), applications and conditionals. An ASL program consists of a global declaration of an identifier getting bound to the value of an expression. The primitive functions that are available are equality between numbers and arithmetic binary operations. The concrete syntax of ASL expressions can be described (ambiguously) as:

```
Expr ::= INT
       | IDENT
       | "if" Expr "then" Expr "else" Expr "fi"
       | "(" Expr ")"
       | "\" IDENT "." Expr
```

and the syntax of declarations is given as:

```
Decl ::= "let" IDENT "be" Expr ";"
       | Expr ";"
```

Arithmetic binary operations will be written in prefix position and will belong to the class `IDENT`. The `\` symbol will play the role of the Caml keyword `function`.

We start by defining the abstract syntax of ASL expressions and of ASL toplevel phrases. Then we define a parser in order to produce abstract syntax trees from the concrete syntax of ASL programs.

12.1 ASL abstract syntax trees

We encode variable names by numbers. These numbers represent the *binding depth* of variables. For instance, the function of `x` returning `x` (the ASL identity function) will be represented as:

```
Abs("x", Var 1)
```

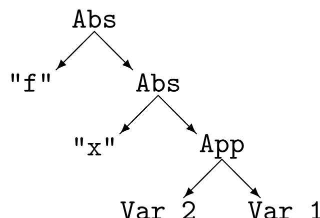
And the ASL application function which would be written in Caml:

```
(function f -> (function x -> f(x)))
```

would be represented as:

```
Abs("f", Abs("x", App(Var 2, Var 1)))
```

and should be viewed as the tree:



`Var n` should be read as "an occurrence of the variable bound by the `n`th abstraction node encountered when going toward the root of the abstract syntax tree". In our example, when going from `Var 2` to the root, the 2nd abstraction node we encounter introduces the `"f"` variable.

The numbers encoding variables in abstract syntax trees of functional expressions are called "De Bruijn[1] numbers". The characters that we attach to abstraction nodes simply serve as documentation: they will not be used by any of the semantic analyses that we will perform on the trees. The type of ASL abstract syntax trees is defined by:

```
#type asl = Const of int
#         | Var of int
#         | Cond of asl * asl * asl
#         | App of asl * asl
#         | Abs of string * asl
#
#and top_asl = Decl of string * asl;;
Type asl defined.
Type top_asl defined.
```

12.2 Parsing ASL programs

Now we come to the problem of defining a concrete syntax for ASL programs and declarations.

The choice of the concrete aspect of the programs is simply a matter of taste. The one we choose here is close to the syntax of λ-calculus (except that we will use the *backslash* character because there is no "λ" on our keyboards). We will use the *curried* versions of equality and arithmetic functions. We will also use a *prefix* notation (à la Lisp) for their application. We will write "`+ (+ 1 2) 3`" instead of "`(1+2)+3`". The "`if` e_1 `then` e_2 `else` e_3" construct will be written "`if` e_1 `then` e_2 `else` e_3 `fi`", and will return the `then` part when e_1 is different from 0 (0 acts thus as falsity in ASL conditionals).

[1]They have been proposed by N.G. De Bruijn in [10] in order to facilitate the mechanical treatment of λ-calculus terms.

12.2.1 Lexical analysis

The concrete aspect of ASL programs will be either declarations of the form:

`let` *identifier* `be` *expression*;

or:

expression;

which will be understood as:

`let it be` *expression*;

The tokens produced by the lexical analyzer will represent the keywords `let`, `be`, `if` and `else`, the `\` binder, the dot, parentheses, integers, identifiers, arithmetic operations and terminating semicolons. We reuse here most of the code that we developed in chapter 10 or in the answers to its exercises.

Skipping blank spaces:

```
#let rec spaces = function
#  [< ’‘ ‘|‘\t‘|‘\n‘; spaces _ >] -> ()
#| [< >] -> ();;
spaces : char stream -> unit = <fun>
```

The type of tokens is given by:

```
#type token = LET | BE | LAMBDA | DOT | LPAR | RPAR
#           | IF | THEN | ELSE | FI | SEMIC
#           | INT of int | IDENT of string;;
Type token defined.
```

Integers:

```
#let int_of_digit = function
#  ‘0‘..‘9‘ as c -> (int_of_char c) - (int_of_char ‘0‘)
#| _ -> raise (Failure "not a digit");;
int_of_digit : char -> int = <fun>

#let rec integer n = function
#  [< ’ ‘0‘..‘9‘ as c; (integer (10*n + int_of_digit c)) r >] -> r
#| [< >] -> INT n;;
integer : int -> char stream -> token = <fun>
```

We restrict ASL identifiers to be composed of lowercase letters, the eight first being significative. An explanation about the `ident` function can be found in the chapter dedicated to the answers to exercises (chapter 17). The function given here is slightly different and tests its result in order to see wether it is a keyword (`let`, `be`, ...) or not:

```
#let ident_buf = make_string 8 ‘ ‘;;
ident_buf : string = "        "

#let rec ident len = function
```

```
#  [< ' ‘a‘..‘z‘ as c;
#     (if len >= 8 then ident len
#      else begin
#            set_nth_char ident_buf len c;
#            ident (succ len)
#           end) s >] -> s
#| [< >] -> (match sub_string ident_buf 0 len
#            with "let" -> LET
#               | "be" -> BE
#               | "if" -> IF
#               | "then" -> THEN
#               | "else" -> ELSE
#               | "fi" -> FI
#               | s -> IDENT s);;
ident : int -> char stream -> token = <fun>
```

A reasonable lexical analyzer would use a hash table to recognize keywords faster.

Primitive operations are recognized by the following function, which also detects illegal operators and ends of input:

```
#let oper = function
#  [< '‘+‘|‘-‘|‘*‘|‘/‘|‘=‘ as c >] -> IDENT(make_string 1 c)
#| [< 'c >] -> prerr_string "Illegal character: ";
#             prerr_endline (char_for_read c);
#             raise (Failure "ASL parsing")
#| [< >] -> prerr_endline "Unexpected end of input";
#           raise (Failure "ASL parsing");;
oper : char stream -> token = <fun>
```

The lexical analyzer has the same structure as the one given in chapter 10 except that leading blanks are skipped.

```
#let rec lexer str = spaces str;
#match str with
#  [< '‘(‘; spaces _ >]  -> [< 'LPAR; lexer str >]
#| [< '‘)‘; spaces _ >]  -> [< 'RPAR; lexer str >]
#| [< '‘\\‘; spaces _ >] -> [< 'LAMBDA; lexer str >]
#| [< '‘.‘; spaces _ >]  -> [< 'DOT; lexer str >]
#| [< '‘;‘; spaces _ >]  -> [< 'SEMIC; lexer str >]
#| [< '‘0‘..‘9‘ as c;
#     (integer (int_of_digit c)) tok;
#      spaces _ >]       -> [< 'tok; lexer str >]
#| [< '‘a‘..‘z‘ as c;
#     (set_nth_char ident_buf 0 c; ident 1) tok;
#     spaces _ >]        -> [< 'tok; lexer str >]
#| [< oper tok; spaces _ >] -> [< 'tok; lexer str >]
#;;
```

```
lexer : char stream -> token stream = <fun>
```

The lexical analyzer returns a stream of tokens that the parser will receive as argument.

12.2.2 Parsing

The final output of our parser will be abstract syntax trees of type `asl` or `top_asl`. This implies that we will detect unbound identifiers at parse-time. In this case, we will raise the `Unbound` exception defined as:

```
#exception Unbound of string;;
Exception Unbound defined.
```

We also need a function which will compute the binding depths of variables. That function simply looks for the position of the first occurrence of a variable name in a list. It will raise `Unbound` if there is no such occurrence.

```
#let binding_depth s rho =
# let rec bind n = function
#     []  -> raise (Unbound s)
#  | t::l -> if s = t then Var n else bind (n+1) l
#  in bind 1 rho
#;;
binding_depth : string -> string list -> asl = <fun>
```

We also need a global environment, containing names of already bound identifiers. The global environment contains predefined names for the equality and arithmetic functions. We represent the global environment as a reference since each ASL declaration will augment it with a new name.

```
#let init_env =  ["+";"-";"*";"/";"="];;
init_env : string list = ["+"; "-"; "*"; "/"; "="]
```

```
#let global_env = ref init_env;;
global_env : string list ref = ref ["+"; "-"; "*"; "/"; "="]
```

We now give a parsing function for ASL programs. Blanks at the beginning of the string are skipped.

```
#let rec top = function
#    [< 'LET; 'IDENT id; 'BE; expression e; 'SEMIC >] -> Decl(id,e)
#  | [< expression e; 'SEMIC >] -> Decl("it",e)
#
#and expression = function
#    [< (expr !global_env) e >] -> e
#
#and expr rho =
#  let rec rest e1 = function
#          [< (atom rho) e2; (rest (App(e1,e2))) e >] -> e
#        | [< >] -> e1
```

```
#  in function
#       [< 'LAMBDA; 'IDENT id; 'DOT; (expr (id::rho)) e >] -> Abs(id,e)
#     | [< (atom rho) e1; (rest e1) e2 >] -> e2
#
#and atom rho = function
#    [< 'IDENT id >] ->
#        (try binding_depth id rho with Unbound s ->
#              print_string "Unbound ASL identifier: ";
#              print_string s; print_newline();
#              raise (Failure "ASL parsing"))
#  | [< 'INT n >] -> Const n
#  | [< 'IF; (expr rho) e1; 'THEN; (expr rho) e2;
#       'ELSE; (expr rho) e3; 'FI >] -> Cond(e1,e2,e3)
#  | [< 'LPAR; (expr rho) e; 'RPAR >] -> e;;
top : token stream -> top_asl = <fun>
expression : token stream -> asl = <fun>
expr : string list -> token stream -> asl = <fun>
atom : string list -> token stream -> asl = <fun>
```

The complete parser that we will use reads a string, converts it into a stream, and produces the token stream that is parsed:

```
#let parse_top s = top(lexer(stream_of_string s));;
parse_top : string -> top_asl = <fun>
```

Let us try our grammar (we do not augment the global environment at each declaration: this will be performed after the semantic treatment of ASL programs). We need to write double \ inside strings, since \ is the string escape character.

```
#parse_top "let f be \\x.x;";;
- : top_asl = Decl ("f", Abs ("x", Var 1))
#parse_top "let x be + 1 ((\\x.x) 2);";;
- : top_asl =
 Decl ("x", App (App (Var 1, Const 1), App (Abs ("x", Var 1), Const 2)))
```

Unbound identifiers and undefined operators are correctly detected:

```
#parse_top "let y be g 3;";;
Unbound ASL identifier: g
Uncaught exception: Failure "ASL parsing"
#parse_top "f (if 0 then + else - fi) 2 3;";;
Unbound ASL identifier: f
Uncaught exception: Failure "ASL parsing"
#parse_top "^ x y;";;
Illegal character: ^
Uncaught exception: Failure "ASL parsing"
```