6 Compiling Standard ML

The ML in the name *Standard ML* stands for meta-language. SML was designed as a language for describing languages when it was used as part of the Logic for Computable Functions (LCF) system [9]. Two tools were designed to work with Standard ML for language implementation, *ML-lex* and *ML-yacc*. The pattern matching, ease of defining recursive datatypes, the functional nature of the language along with these two tools make Standard ML an excellent language choice for implementing interpreters and compilers. This chapter introduces these two tools through a case study involving the development of a compiler for a subset of the Standard ML language called the *Small* language. Over the years the Small language has grown into a pretty robust subset of Standard ML.

Depicted in Fig. 6.1 are all the relevant pieces in constructing and using the *mlcomp* compiler which can be downloaded from http://github.com/kentdlee/mlcomp. Compiling an SML program begins by scanning the source file for tokens. The scanner is called by the parser to get each of the tokens from the SML source file. The parser, a bottom-up parser, performs a reverse right-most derivation of the source program forming an abstract syntax tree along the way. When the AST is returned by the parser, the compiler calls the code generator to evaluate the tree and produce the target code, in this case JCoCo assembly language. In this chapter the scanner and the parser won't have to be written by hand. *ML-lex* and *ML-yacc* are used to generate these parts of the compiler from specifications that are provided to these tools.

Two commonly used terms in compiler construction are the *front end* and the *back end*. The *front end*, referring to the scanner and the parser, reads the tokens and builds an AST of a program. The *back end* generates the code given the AST representation of the program. ML-lex and ML-yacc are used to generate the front end from two specifications, provided by the compiler writer. The back end is written by the compiler writer to generate the code given an AST of the program. In Fig. 6.1 the light green objects are the parts of the compiler provided by the compiler writer. The dark green box represents the SML program provided by the user of the compiler, the

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_6

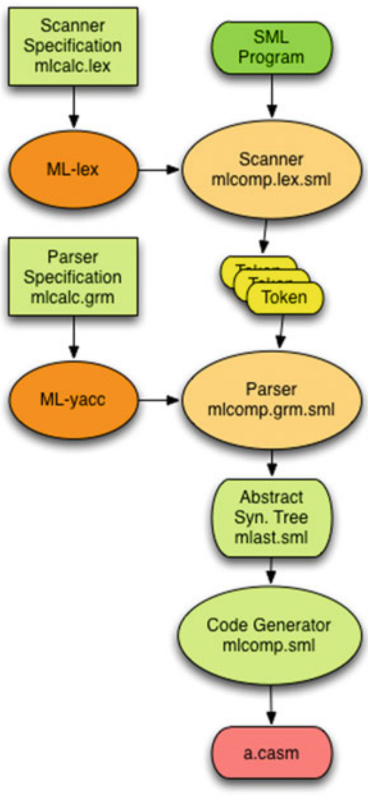

Scanner Specification mlcalc.lex
SML Program
ML-lex
Scanner mlcomp.lex.sml
Parser Specification mlcalc.grm
Token
Token
Token
ML-yacc
Parser mlcomp.grm.sml
Abstract Syn. Tree mlast.sml
Code Generator mlcomp.sml
a.casm

Fig. 6.1 Structure of MLComp

SML programmer who is compiling his or her code. Summarizing, the files written by the compiler writer include the following.

- The tokens of the language are defined in a file called *mlcomp.lex*.
- The datatype for the AST is defined in a file called *mlast.sml*.
- The grammar of the language is defined in a file called *mlcomp.grm*. This file also contains a mapping from productions in the grammar to nodes in an AST. The parser reads tokens and builds an AST of the expression being compiled.
- The code generator is defined in a file called *mlcomp.sml*.

The next sections introduce ML-lex, ML-yacc, and code generation. The rest of this chapter explores parts of the compiler that are already completed and other possible enhancements to the language. Building and using this compiler requires installation of Standard ML and the ML-yacc and ML-lex tools.

Don't be intimidated! The suggested enhancements to the language are accompanied by test programs that use these enhancements. By attempting to compile one of these tests you will be pointed at the location in the compiler where new code is required. Adding that code will lead you to another location within the compiler, and so on. The compiler is designed so that it will tell you where enhancements are needed when you attempt to compile a test that is not currently supported. By repeatedly attempting to build the compiler and compile a new test, you will be given a hands-on tour of the compiler. That, along with the descriptions in this chapter of how the compiler currently works will teach you about compiler construction for a non-trivial language! Good luck. With a little work you will learn a lot about compiler construction and implementing a functional programming language!

6.1 ML-lex

ML-lex is a scanner generator. ML-lex generates a function that can be used to get tokens from the input. It is based on a similar tool called *lex* that generates scanners for C programs. The input to the two tools is similar but not exactly the same. The input to ML-lex is a file consisting of three sections, where each section is separated by %%. The format of an ML-lex input file is:

```
User declarations
%%
ML-lex definitions
%%
Token Rules
```

The user declarations include any ML code that will assist you in defining the tokens. Typically, a variable is used to keep track of the line of input being read. There might also be some functions for converting strings to other values like integers. An error function that handles bad tokens is a common function for this section to get the scanner and the parser to work together.

The ML-lex definitions follow the user declarations. Sets of characters are declared in this section. In addition a functor must be declared. A functor is a module that takes a structure as a parameter and returns a new structure as a result. A functor is used by ML-lex and ML-yacc to create the scanner.

The last section of an ML-lex definition is composed of a set of rules that define the tokens of the language. Each rule has the form:

```
reg_exp => (return_value);
```

The *reg_exp* is a regular expression. The language of regular expressions can be used to define tokens. Every regular expression can be expressed as a finite state machine. Finite state machines can be used to recognize tokens. The set of *reg_exp* is eventually translated into a finite state machine that can be used to recognize tokens in the language. When a string of characters is recognized as a token, its matching return value is constructed from the rules and that value is returned by the scanner to the parser. Figures 6.2, 6.3, and 6.4 contain the three parts of the lexer specification given to ML-lex for the mlcomp compiler. The file is called mlcomp.lex.

```
1  (* mlcomp.lex -- lexer spec *)
2  type pos = int
3  type svalue = Tokens.svalue
4  type ('a, 'b) token = ('a, 'b) Tokens.token
5  type lexresult = (svalue, pos) token
6  val pos = ref 1
7  val error = fn x => TextIO.output(TextIO.stdErr, x ^ "\n")
8  val eof = fn () => Tokens.EOF(!pos, !pos)
9  fun countnewlines s =
10     let val lst = explode s
11         fun count (c:char) nil = 0
12           | count c (h::t) =
13             let val tcount = count c t
14             in
15               if c = h then 1+tcount else tcount
16             end
17     in
18       pos:= (!pos) + (count #"\n" lst)
19     end
```

Fig. 6.2 mlcomp.lex part one

```
20  %%
21  %header (functor mlcompLexFun(structure Tokens : mlcomp_TOKENS));
22  alpha=[A-Za-z];
23  alphanumeric=[A-Za-z0-9_\.];
24  digit=[0-9];
25  ws=[\ \t];
26  dquote=[\"];
27  squote=[\'];
28  anycharbutquote=[^"];
29  anychar=[.];
30  pound=[\#];
31  tilde=[\~];
32  period=[\.];
```

Fig. 6.3 mlcomp.lex part two

In Fig. 6.2 lines 1–19 make up the first part of the ML-lex specification, the user declarations. The *pos* type must be defined and is used to define the position within the source program where a token is found. In this case, the position is the line on which it is found. Later a variable called *pos* is also initialized to 1 for the first line of the source program.

A structure called Tokens is used to contain information about the tokens returned by ML-lex. The *Tokens.svalue* is the actual string representing the characters of each token. Line 3 just equates a type called *svalue* to the *Tokens.svalue*. Line 4 does the same for the type *token*. Those two type names are used in line 6 where *lexresult* is declared. This *lexresult* is required to be defined in the user declarations section of *ML-lex*.

The *error* function is used later in the lexer specification. The *eof* function is used to return the *EOF* token and is called automatically by the lexer when it reaches the

```
%%
\(\*([^*]|[\r\n]|(\*+([^*\)]|[\r\n])))*\*+\) => (countnewlines yytext; lex());
\n  => (pos := (!pos) + 1; lex());
{ws}+  => (lex());
"+"  => (Tokens.Plus(!pos,!pos));
"*"  => (Tokens.Times(!pos,!pos));
"-"  => (Tokens.Minus(!pos,!pos));
"@"  => (Tokens.Append(!pos,!pos));
"=" => (Tokens.Equals(!pos,!pos));
"("  => (Tokens.LParen(!pos,!pos));
")"  => (Tokens.RParen(!pos,!pos));
"[" => (Tokens.LBracket(!pos,!pos));
"]" => (Tokens.RBracket(!pos,!pos));
"::" => (Tokens.ListCons(!pos,!pos));
"," => (Tokens.Comma(!pos,!pos));
";" => (Tokens.Semicolon(!pos,!pos));
"_" => (Tokens.Underscore(!pos,!pos));
"=>" => (Tokens.Arrow(!pos,!pos));
"|" => (Tokens.VerticalBar(!pos,!pos));
">" => (Tokens.Greater(!pos,!pos));
(* a few token are omitted here *)
{tilde}?{digit}+  => (Tokens.Int(yytext,!pos,!pos));
{pound}{dquote}{anychar}{dquote} => (Tokens.Char(yytext,!pos,!pos));
{dquote}{anycharbutquote}*{dquote} => (Tokens.String(yytext,!pos,!pos));
{alpha}{alphanumeric}*=>
   (let val tok = String.implode (List.map (Char.toLower)
              (String.explode yytext))
    in
      if      tok="let" then Tokens.Let(!pos,!pos)
      else if tok="val" then Tokens.Val(!pos,!pos)
      else if tok="in" then Tokens.In(!pos,!pos)
      else if tok="end" then Tokens.End(!pos,!pos)
      else if tok="if" then Tokens.If(!pos,!pos)
      else if tok="then" then Tokens.Then(!pos,!pos)
      else if tok="else" then Tokens.Else(!pos,!pos)
      else if tok="div" then Tokens.Div(!pos,!pos)
      else if tok="mod" then Tokens.Mod(!pos,!pos)
      else if tok="fn" then Tokens.Fn(!pos,!pos)
      else if tok="while" then Tokens.While(!pos,!pos)
      else if tok="do" then Tokens.Do(!pos,!pos)
      else if tok="and" then Tokens.And(!pos,!pos)
      else if tok="rec" then Tokens.Rec(!pos,!pos)
      else if tok="fun" then Tokens.Fun(!pos,!pos)
      else if tok="as" then Tokens.As(!pos,!pos)
      else if tok="handle" then Tokens.Handle(!pos,!pos)
      else if tok="raise" then Tokens.Raise(!pos,!pos)
      else if tok="true" then Tokens.True(!pos,!pos)
      else if tok="false" then Tokens.False(!pos,!pos)
      else Tokens.Id(yytext,!pos,!pos)
    end);
.  => (error ("error: bad token "^yytext); lex())
```

Fig. 6.4 mlcomp.lex part three

end of file. The *countnewlines* function is also used later in the lexer specification when skipping over whitespace likes spaces, tabs, and newline characters.

The ML-lex declarations begins with declaring a functor in Fig. 6.3. The functor is required by the parser. A functor is a parameterized type in Standard ML and this functor is expected by the parser and is parameterized by the Tokens structure. Declaring the functor in this way is required to get the parser generated by ML-yacc to talk to the scanner generated by ML-lex.

The alpha declaration declares a class of characters called *alpha* that consists of letters *a* to *z* in lower and upper case. The *alphanumeric* characters include letters, digits, underscores, and the period character. The *digit* declaration defines the class of digits as being *0* to *9*. The *ws* stands for whitespace. It defines blanks and tabs as whitespace. The *dquote* class is for double quote and *squote* for single quote. The ^ means *not* so *anycharbutquote* is exactly as it reads. The period represents any character whatsoever. The actual period character must be escaped by preceding it with a backslash.

Finally, the rules define all the tokens in the third part of the lexer definition in Fig. 6.4. The first rule discards comments in the source file. It says that comments look like *(* any text *)*. Unfortunately, this is a complex regular to start the rule definitions with. It is first because the rules will be matched in order of their definition. It begins by saying look for a left paren followed by an asterisk. Then the next part of the regular expression is one of two possibilities.

- A character which is not an asterisk or it is a carriage return (i.e. the \r) or a newline (i.e. the \n).
- A string of characters which is some number of asterisks followed by a either not an asterisk or a right paren or a carriage return or a newline.

Those two preceeding bullets may repeat zero or more times (the Kleene star that appears near the end of the regular expression says this). Finally, the whole regular expression ends by saying that the comment ends with one or more asterisks followed by a right paren. The action to take in this case is to call the *countnewlines* function to count any newline characters in the comment to update the *pos* variable accordingly. Finally, calling *lex* at the end of the action causes the lexer to get the next token, effectively ignoring the comment so the parser never sees it. The next regular expression skips newlines that are not in a comment. The third regular expression skips blanks and tabs that might appear in the program.

The next several rules define short simple tokens like infix operators. The token is defined within the *Tokens* structure and each rule returns a particular token value, defined in the parser. Every token value carries with it two integers. In this case, the line number is provided for both values. Tokens that consist of more than one or two characters should not be defined in this way since the length of each token string makes the number of states grow exponentially. The remaining rules define tokens that can't be explicitly given along with the keywords that are defined like identifiers in the language.

Line 54 defines positive and negative numbers. The *?* indicates 0 or 1 occurrence of the *negation* symbol. This is followed by 1 or more digits, followed by a possible period and other digits. Line 55 defines character constants in Standard ML like #“a” for instance. Escape characters like the newline character, “n”, are not currently supported by could be. Line 56 defines string tokens which start with a double quote followed by zero or more of any character but double quote followed by a double quote. Lines 57–82 recognize identifiers and keywords. Defining one rule to handle all these different tokens with an *if-else-if* expression reduces the final number of states in the scanner. If each keyword were handled by a separate rule the number of states in the scanner would explode. Finally, line 83 handles any other character that might be found in the source file by writing an error message to the screen and skipping over it.

The scanner generated by ML-lex returns each token described in Fig. 6.4 with the line number in the source program where it was found. In some cases, the *lexeme*, the actual string of characters making up the token, is also returned. The *lexeme* is returned for tokens where the token type is not enough information. For instance, *Int*, *String*, *Char*, and *Id* tokens all need to carry along the lexeme, the *yytext*, because that information is needed by the parser. From a definition like the *mlcomp.lex* file shown in Figs. 6.2, 6.3, and 6.4 the ML-lex tool has enough information to generate a scanner for the tokens of the language.

Practice 6.1 Given the ML-lex specification in Figs. 6.2, 6.3, and 6.4, what more would have to be added to allow expressions like this to be correctly tokenized by the scanner? What new tokens would have to be recognized? How would you modify the specification to accept these tokens?

```
case x of
    1 => "hello"
  | 2 => "how"
  | 3 => "are"
  | 4 => "you"
```

You can check your answer(s) in Section 6.15.1.

6.2 The Small AST Definition

The parser reads tokens and builds an abstract syntax tree of a source program. Figure 6.5 contains the abstract syntax definition for the Small language. In SML, the abstract syntax definition is given by an SML datatype. Each type of node in the tree is tagged with its type. Some nodes in the tree include the subtrees such as the infixexp node. The datatype can consist of multiple types which may all be mutually recursive. For the multiple types to be mutually recursive, the keyword *and* is used to separate the datatype definitions.

The Small subset of Standard ML is primarily composed of expressions. The *exp* datatype describes trees representing expressions in the language. An expression is either an integer, character, boolean value, identifier, list constant, tuple constant, function application, infix expression, a sequence of expressions, a let declaration, a raised exception, an exception handler, an *if then* expression, a *while do* expression, or a function definition.

A function definition and an exception handler contain a list of matches. A match is composed of a pattern and an expression as in *4 =>"you"* for instance. The allowed

```
structure MLAS =
struct
datatype
  exp = int of string
      | ch of string
      | str of string
      | boolval of string
      | id of string
      | listcon of exp list
      | tuplecon of exp list
      | apply of exp * exp
      | infixexp of string * exp * exp
      | expsequence of exp list
      | letdec of dec * (exp list)
      | raisexp of exp
      | handlexp of exp * match list
      | ifthen of exp * exp * exp
      | whiledo of exp * exp
      | func of int * match list
and
  match = match of pat * exp
and
  pat = intpat of string
      | chpat of string
      | strpat of string
      | boolpat of string
      | idpat of string
      | wildcardpat
      | infixpat of string * pat * pat
      | tuplepat of pat list
      | listpat of pat list
      | aspat of string * pat
and
  dec = bindval of pat * exp
      | bindvalrec of pat * exp
      | funmatch of string * match list
      | funmatches of
end
```

Fig. 6.5 mlast.sml

patterns are described by the *pat* datatype and include integers, characters, strings, boolean values, identifiers, the underscore pattern (called *wildcardpat* in the AST definition), tuples, lists, and a special *as* pattern which lets the programmer specify an identifer as a pattern as in *z as (x,y)*. This would match a pattern where *x* and *y* match the elements of a tuple and *z* matches the entire tuple.

A *let* expression binds identifiers to values and the *dec* datatype defines binding declarations. In Standard ML it is possible to bind the identifiers in a pattern to an expression. The *bindvalrec* represents a recursive binding which is needed in the case of recursive function definitions. The *funmatch* is used in a function which is defined with a series of pattern matching cases. The *funmatches* comes into play when a series of mutually recursive function definitions are being defined, somewhat like the mutually recursive AST datatype definition given in Fig. 6.5.

Practice 6.2 How would you modify the abstract syntax so expressions like this could be represented?

```
case x of
    1 => "hello"
  | 2 => "how"
  | 3 => "are"
  | 4 => "you"
```

You can check your answer(s) in Section 6.15.2.

6.3 Using ML-yacc

ML-yacc is a parser generator. The name stands for *Yet Another Compiler Compiler* (i.e. yacc). *Yacc* is a tool that generates parsers for compilers written in C or C++. ML-yacc is the SML version of this tool. ML-yacc is a little different than yacc but provides mostly the same functionality. ML-yacc's input format is similar to ML-lex's input format. An ML-yacc specification consists of three parts.

```
User declarations
%%
ML-yacc definitions
%%
Rules
```

The user declarations include providing the AST definition and any functions, variables, or exceptions that might be useful while parsing the input. Figures 6.6, 6.7, 6.8, and 6.9 contains the parser specification for the Small language.

The user declarations of the parser are on lines 1–42 of Fig. 6.6. This part of the parser contains useful utility functions much like the user declaration section of ML-lex. The abstract syntax definition is opened in the parser. This is similar to the *using*

```
1  open MLAS;
2  val idnum = ref 0
3  fun nextIdNum() =
4    let val x = !idnum
5    in
6      idnum := !idnum + 1;
7      x
8    end
9  exception emptyDecList;
10 exception argumentMismatch;
11 fun uncurryIt nil = raise emptyDecList
12   | uncurryIt (L as ((name,patList,exp)::t)) =
13     let fun len nil = raise argumentMismatch
14           | len [(n,p,e)] = length(p)
15           | len ((n,p,e)::t) =
16             let val size = length(p)
17             in
18               if size = len t then size else
19                 (TextIO.output(TextIO.stdOut,
20                 "Syntax Error: Number of arguments does not match in function "
21                 ^name^"\n");
22                  raise argumentMismatch)
23             end
24         val tupleList = List.map (fn x => "v"^Int.toString(nextIdNum())) patList
25      in
26        len(L); (* just check the parameter list sizes of all patterns *)
27        (name,[match(idpat(hd(tupleList)),
28                   List.foldr (fn (x,y) => func(nextIdNum(),[match(idpat(x), y)]))
29                      (apply (func(nextIdNum(),
30                              List.map (fn (n,p,e) => match(tuplepat(p),e)) L),
31                              tuplecon(List.map (fn x => id(x)) tupleList)))
32                          (tl tupleList))])
33      end
34 fun makeMatchList (nil) = raise emptyDecList
35   | makeMatchList (L as (name,pat,exp)::t) =
36     (name, List.map (fn (n,p,e) =>
37                (if name <> n then (
38                    TextIO.output(TextIO.stdOut,
39                    "Syntax Error: Function definition with different names "
40                    ^name^" and "^n^" not allowed.\n");
41                    raise argumentMismatch)
42                 else match(p,e))) L)
```

Fig. 6.6 mlcomp.grm part one

namespace std in C++. Lines 2–8 define a function that can return a unique integer which is needed in some code in the parser. Line 10–33 define a function and two exceptions that are used in defining curried functions. This is covered in detail later in the chapter. Lines 34–42 convert a list of (name, pattern, expression) tuples to a tuple of (name,list) where the list is a list of (pattern,expression) pairs. It also checks that all names in the original tuples were for the same function.

The ML-yacc definitions start on line 43 if Fig. 6.7. They include a name to prefix functions in the scanner with, in this case *mlcomp*. The *verbose* helps in debugging. The *eop*, or *end of parse*, says that *EOF* is the last token returned. This helps in terminating the parser. The *pos* type is redeclared in Fig. 6.7 for use with the scanner.

The *nodefault* tells the parser not to insert tokens it thinks might have been left out. This helps in finding syntax errors earlier than they would be otherwise. If this were omitted the parser would insert a token when it is reasonably sure the program

```
%%
%name mlcomp (* mlcomp becomes a prefix in functions *)
%verbose
%eop EOF
%pos int
%nodefault
%pure (* no side-effects in actions *)
%term EOF | LParen | RParen | Plus | Minus | Times | Div | Mod | Greater | Less
    | GreaterEqual | LessEqual | NotEqual | Append | ListCons | Negate | Comma
    | Semicolon | Underscore | Arrow | Equals | VerticalBar | LBracket
    | RBracket | Fun | As | Let | Val | In | End | If | Then | Else | Fn
    | While | Do | Handle | Raise | And | Rec | String of string
    | Char of string | Int of string | True | False | Id of string
    | SetEqual | Exclaim
%nonterm Prog of exp | Exp of exp | Expressions of exp list
       | ExpSequence of exp list | MatchExp of match list
       | Pat of pat | Patterns of pat list
       | PatternSeq of pat list | Dec of dec | ValBind of dec
       | FunBind of (string * match list) list
       | FunMatch of (string * pat * exp) list
       | Con of exp | FuncExp of exp | DecSeq of dec list
       | CurriedFun of (string * pat list * exp) list
%right SetEqual
%left Plus Minus Append Equals NotEqual
%left Times Div Mod Greater Less GreaterEqual LessEqual
%right ListCons
%right Exclaim
```

Fig. 6.7 mlcomp.grm part two

```
%%
Prog : Exp EOF                         (Exp)
Exp : Con                              (Con)
    | Id                               (id(Id))
    | FuncExp Exp                      (apply(FuncExp,Exp))
    | Exclaim Exp                      (apply(id("!"),Exp))
    | Id SetEqual FuncExp              (infixexp(":=",id(Id),FuncExp))
    | Exp Plus Exp                     (infixexp("+",Exp1,Exp2))
    | Exp Minus Exp                    (infixexp("-",Exp1,Exp2))
    | Exp Times Exp                    (infixexp("*",Exp1,Exp2))
    | Exp Div Exp                      (infixexp("div",Exp1,Exp2))
    | Exp Mod Exp                      (infixexp("mod",Exp1,Exp2))
    | Exp Greater Exp                  (infixexp(">",Exp1,Exp2))
    | Exp GreaterEqual Exp             (infixexp(">=",Exp1,Exp2))
    | Exp Less Exp                     (infixexp("<",Exp1,Exp2))
    | Exp LessEqual Exp                (infixexp("<=",Exp1,Exp2))
    | Exp Equals Exp                   (infixexp("=",Exp1,Exp2))
    | Exp NotEqual Exp                 (infixexp("<>",Exp1,Exp2))
    | Exp Append Exp                   (infixexp("@",Exp1,Exp2))
    | Exp ListCons Exp                 (infixexp("::",Exp1,Exp2))
    | LParen Exp RParen                (Exp)
    | LParen Expressions RParen        (tuplecon(Expressions))
    | LParen ExpSequence RParen        (expsequence(ExpSequence))
    | LBracket Expressions RBracket    (listcon(Expressions))
    | LBracket RBracket                (id("nil"))
    | Let DecSeq In ExpSequence End
               (List.hd (List.foldr (fn (x,y) =>
                   [letdec(x,y)]) ExpSequence DecSeq))
    | Raise Exp                        (raisexp(Exp))
    | Exp Handle MatchExp              (handlexp(Exp,MatchExp))
    | If Exp Then Exp Else Exp         (ifthen(Exp1,Exp2,Exp3))
    | While Exp Do Exp                 (whiledo(Exp1,Exp2))
    | Fn MatchExp                      (func(nextIdNum(),MatchExp))
```

Fig. 6.8 mlcomp.grm part three

```
FuncExp : Exp                         (Exp)
Expressions : Exp                     ([Exp])
            | Exp Comma Expressions   (Exp::Expressions)
ExpSequence : Exp                     ([Exp])
            | Exp Semicolon ExpSequence
                                      (Exp::ExpSequence)
MatchExp : Pat Arrow Exp              ([match(Pat,Exp)])
         | Pat Arrow Exp VerticalBar MatchExp
                                      (match(Pat,Exp)::MatchExp)
Pat : Int                             (intpat(Int))
    | Char                            (chpat(Char))
    | String                          (strpat(String))
    | True                            (boolpat("true"))
    | False                           (boolpat("false"))
    | Underscore                      (wildcardpat)
    | Id                              (idpat(Id))
    | Pat ListCons Pat                (infixpat("::",Pat1,Pat2))
    | LParen Pat RParen               (Pat)
    | LParen Patterns RParen          (tuplepat(Patterns))
    | LBracket Patterns RBracket      (listpat(Patterns))
    | LBracket RBracket               (idpat("nil"))
    | Id As Pat                       (aspat(Id,Pat))
Patterns : Pat                        ([Pat])
         | Pat Comma Patterns         (Pat::Patterns)
PatternSeq : Pat                      ([Pat])
           | Pat PatternSeq           (Pat::PatternSeq)
Dec : Val ValBind                     (ValBind)
    | Fun FunBind                     (funmatches(FunBind))
DecSeq : Dec                          ([Dec])
       | Dec DecSeq                   (Dec::DecSeq)
ValBind : Pat Equals Exp              (bindval(Pat,Exp))
        | Rec Id Equals Exp           (bindvalrec(idpat(Id),Exp))
FunBind : FunMatch                    ([makeMatchList FunMatch])
        | CurriedFun                  ([uncurryIt CurriedFun])
        | FunBind And FunBind         (FunBind1@FunBind2)
FunMatch : Id Pat Equals Exp          ([(Id,Pat,Exp)])
         | Id Pat Equals Exp VerticalBar FunMatch
                                      ((Id,Pat,Exp)::FunMatch)
CurriedFun :
           Id PatternSeq Equals Exp   ([(Id,PatternSeq,Exp)])
         | Id PatternSeq Equals Exp VerticalBar CurriedFun
                                      ((Id,PatternSeq,Exp)::CurriedFun)
Con : Int                             (int(Int))
    | Char                            (ch(Char))
    | String                          (str(String))
    | True                            (boolval("true"))
    | False                           (boolval("false"))
    | LParen RParen                   (tuplecon([]))
```

Fig. 6.9 mlcomp.grm part four

being parsed is missing a token. The *pure* declarations says that the parser has no side-effects. It simply builds a tree and returns it. This means that ML-yacc can undo certain parsing operations if it needs to without fear of a side-effect not being undone.

Most importantly the terminals and nonterminals of the language are declared in the ML-yacc declarations. Those tokens that carry along their lexeme are declared as a token *of* something. For instance, *Int of string* where the string is a string containing the token's number. The nonterminals include all the *nonterm* defined identifiers and represent the syntactic categories of the grammar.

There are just a few more declarations in the ML-yacc definitions section. The grammar rules, given in the next section, have some ambiguity in them. Specifically, some of the operators have ambiguous precedence. The associativity and precedence rules are defined on lines 65–69 with those operators with lowest precedence coming first and higher precedence operators later. So *SetEqual* has the lowest precedence and is right associative. The *Plus*, *Minus*, *Append*, *Equals*, and *NotEqual* operator tokens have the next lowest precedence and are all left associative. These precedence rules simplify the writing of the grammar while disambiguating it.

Lines 70–150 of Figs. 6.8 and 6.9 make up the *Rules* section and define the grammar for Small. Each production of the grammar is given on the left of the AST it returns when matched. Consider a Small program like this:

```
4 * x + 5
```

When matching the rule on line 77 of the grammar specification the *4*x* will match the expression on the left side of the *Plus* token. The AST that results from parsing *4*x* is named *Exp1* by ML-yacc. Remember, the parser is a bottom-up parser so the *4*x* has already been parsed. The *5* on the right side of the *Plus* has also been parsed when this rule is matched. The *5* is referred to as *Exp2* by ML-yacc. The rule on line 74 says when this rule is matched to return an AST of *infixexp("+",Exp1,Exp2)*. The full AST for this expression, and the value returned for this example, would be

infixexp("+", infixexp("*", int("4"), id("x")), int("5"))

To the right of each production is a value that is returned when that production is matched during parsing. In most cases, this is a straight-forward construction of an AST. In a few cases a list is returned instead as in the *MatchExp* nonterminal or the *PatternSeq* nonterminal. In a couple of cases, the *uncurryIt* or *makeMatchList* functions are called which in turn generate an AST node to be returned. In the end, the parser returns a description of the source program as an abstract syntax tree.

Practice 6.3 What modifications would be required in the *mlcomp.grm* specification to parse expressions like this?

```
case x of
   1 => "hello"
 | 2 => "how"
 | 3 => "are"
 | 4 => "you"
```

You can check your answer(s) in Section 6.15.3.

```
5 + 4
```

Fig. 6.10 SML addition

```
Function: main/0
Constants: None, 5, 4
BEGIN
    LOAD_CONST 1
    LOAD_CONST 2
    BINARY_ADD
    POP_TOP
    LOAD_CONST 0
    RETURN_VALUE
END
```

Fig. 6.11 JCoCo addition

```
infixexp("+", int("5"),
              int("4"))
```

Fig. 6.12 Addition AST

6.4 Compiling and Running the Compiler

Code generation is essential to any compiler. The code generator translates the abstract syntax tree into a language that may either be executed directly or interpreted by some low-level interpreter like the JCoCo Virtual Machine. For this text, the *mlcomp* compiler generates JCoCo assembly language. The code generator for *mlcomp* is in the file named *mlcomp.sml*. The entire file is too big to include here. The remainder of this chapter will examine code generation in parts. First, consider code generation for the addition of two integers.

Adding 5 and 4 in the Small language is written as shown in Fig. 6.10. Adding 5 and 4 in JCoCo can be written as shown in the code of Fig. 6.11. The compiler for the Small language is given a source file as shown in Fig. 6.10 and parses it to produce an abstract syntax tree as shown in Fig. 6.12. The abstract syntax tree is passed to the code generator. It is the job of the code generator, given the abstract sytnax tree shown in Fig. 6.12, to generate code similar to that of Fig. 6.11.

```
 1  fun codegen(int(i),outFile,indent,consts,...) =
 2      let val index = lookupIndex(i,consts)
 3      in
 4        TextIO.output(outFile,indent^"LOAD_CONST "^index^"\n")
 5      end
 6    | codegen(infixexp("+",t1,t2),outFile,indent,consts,...) =
 7      let val _ = codegen(t1,outFile,indent,consts,...)
 8          val _ = codegen(t2,outFile,indent,consts,...)
 9      in
10        TextIO.output(outFile,indent^"BINARY_ADD\n")
11  end
```

Fig. 6.13 Addition code generation

The *codegen* function of *mlcomp.sml* is responsible for generating code. To generate code for the AST shown in Fig. 6.12 the two patterns shown in Fig. 6.13 are needed. When the *infixexp* code generation is called, it recursively calls code generation on the two subtrees. The subtrees in this example are the two *int* nodes in the AST, resulting in calling the code generator on *int(i)*. When code is generated for *int("5")*, line 2 looks up the index of the "5" in the constants which is a list of the function's constants much like it appears on line 2 of Fig. 6.11. Line 4 of the code generator then writes the *LOAD_CONST* instruction to the file. The recursive call of codegen on line 8 of Fig. 6.13 similarly calls the *int* codegen to generate the other *LOAD_CONST* instruction. Finally, line 10 of the code in Fig. 6.13 generates the *BINARY_ADD* instruction.

Every JCoCo program must contain code like lines 1–3 of Fig. 6.11. Likewise, lines 7–10 are needed to finish up the *main* function of every JCoCo program. Lines 1–3 are often referred to as the *prolog* of a compiled program and lines 7–10 are commonly referred to as the *epilog* of the program. The *prolog* and *epilog* code is generated by the code that calls the code generator.

The compiler starts when the *run* function is called in the code of Fig. 6.15. The run function is written with two arguments so it can be exported. Exporting a function in Standard ML causes the SML interpreter to export it and all dependent functions into an executable program that can be started from the command-line. The *run* function is like the *main* function in C or C++ program. The arguments to run include the list of command-line arguments to the program. The first item in that list is the first command-line argument. In this case that is the *filename* of the source program. The argument *a* to the *run* function is the name of the SML interpreter used to run the program.

The *run* function then calls the *compile* function passing it the *filename*. Line 2 calls the parser to parse it which returns the AST. Two output files are opened and the *termFile* is written by the *writeTerm* function. This is covered in more detail in Chap. 8.

Lines 7–16 create various bindings of identifiers to locations or functions within the JCoCo virtual machine. Lines 17–23 check for any unbound identifiers in the

```
1  #!/bin/bash
2  set -f
3  export file="$1"
4  if [ -z $file ]; then
5    echo -n "Enter a file name: "
6    read file
7  fi
8  if [ -e $file ]; then
9    rm a.casm >& /dev/null
10   rm a.term >& /dev/null
11   echo ******* Source File ********
12   cat $file
13   sml @SMLload=mlcompimage $file
14   echo * Target Program Execution *
15   coco a.casm
16 else
17   echo FILE DOES NOT EXIST
18 fi
```

Fig. 6.14 The mlcomp script

program which are not allowed. Lines 24–35 are responsible for writing the *prolog* and generating any code for functions that are defined within the program. The *codegen* function is called on line 36. The *epilog* is written by lines 37–41.

The *run* function is invoked via the bash script *mlcomp* in Fig. 6.14. The script is invoked as *mlcomp test0.sml* for instance. The *test0.sml* command-line argument is *$1* in the code. Line 4 checkes to see if a non-empty filename was provided. If not, then lines 5–6 prompt for and get a filename from the user. Line 13 invokes the exported run function by loading the compiled image file *mlcompimage*. The compiler writes a file called *a.casm* which was opened for output on line 3 of Fig. 6.15 on page 201. Then the JCoCo virtual machine is invoked on the target program on line 15 of the script in Fig. 6.14. The *mlcomp* script both compiles and runs the intended SML program.

The Bash script of Fig. 6.16, found in *Makefile.gen*, runs Standard ML's compiler manager. It does this by starting *sml* and then executing the function *CM.make* on the file *sources.cm*. The *exportFn* function creates the binary image executable that is started on line 13 of Fig. 6.14. The target image is created by Standard ML's compiler manager. This is a tool provided with Standard ML much like the *make* utility for Unix except better because Standard ML's compiler manager figures out all dependencies by itself, without the need for a *Makefile*.

The *sources.cm* file is needed to indicate which files to include in the project. Lines 2–3 include the ML-yacc tool (which in turn include ML-lex), the basis library, and some utility code to help with debugging. The last four lines are the compiler source

```
fun compile filename  =
    let val (ast, _) = parse filename
        val outFile = TextIO.openOut("a.casm")
        val termFile = TextIO.openOut("a.term")
        val _ = writeTerm(termFile,ast)
        val _ = TextIO.closeOut(termFile)
        val consts =
            removeDups ("None"::"'Match Not Found'"::"0"::(constants ast))
        val globalBindings = [("println","print"),...]
        val (newbindings,freeVars,cells) =
            localBindings(ast,[],globalBindings,0)
        val bindingVars = removeDups (List.map (fn x => #2(x)) newbindings)
        val cellVars =
            List.map (fn x => boundTo(x,newbindings@globalBindings)) cells
        val locals = listdiff bindingVars cellVars
        val globals = removeDups (List.map (fn (x,y) => y) globalBindings)
    in
      if length(freeVars) <> 0 then
         (TextIO.output(TextIO.stdOut,
            "Error: Unbound variable(s) found in main expression => " ^
            (commaSepList freeVars) ^ ".\n");
          raise notFound)
      else ();
      TextIO.output(outFile,"Function: main/0\n");
      nestedfuns(ast,outFile,"    ",globals,[],globalBindings,0);
      TextIO.output(outFile,"Constants: "^(commaSepList consts) ^ "\n");
      if not (List.null(locals)) then
        TextIO.output(outFile,"Locals: "^(commaSepList locals) ^ "\n")
      else ();
      if not (List.null(cellVars)) then
        TextIO.output(outFile,"CellVars: "^(commaSepList cellVars) ^ "\n")
      else ();
      TextIO.output(outFile,"Globals: "^(commaSepList globals) ^ "\n");
      TextIO.output(outFile,"BEGIN\n");
      makeFunctions(ast,outFile,"    ",consts,...);
      codegen(ast,outFile,"    ",consts,...);
      TextIO.output(outFile,"    POP_TOP\n");
      TextIO.output(outFile,"    LOAD_CONST 0\n");
      TextIO.output(outFile,"    RETURN_VALUE\n");
      TextIO.output(outFile,"END\n");
      TextIO.closeOut(outFile)
    end
    handle _ => (TextIO.output(TextIO.stdOut,
                   "An error occurred while compiling!\n\n"));
 fun run(a,b::c) = (compile b; OS.Process.success)
   | run(a,b) = (TextIO.print("usage: sml @SMLload=mlcomp\n");
                 OS.Process.success)
```

Fig. 6.15 MLComp run function

```
#!/bin/bash
sml << EOF
CM.make "sources.cm";
SMLofNJ.exportFn("mlcompimage",
    mlcomp.run);
EOF
```

Fig. 6.16 Makefile.gen

```
Group is
  $/ml-yacc-lib.cm
  $/basis.cm
  $smlnj-tdp/back-trace.cm
  mlcomp.lex
  mlcomp.grm
  mlcomp.sml
  mlast.sml
```

Fig. 6.17 sources.cm

code for the *mlcomp* compiler. From this simple specification, the Standard ML Compiler Manager will run ML-lex and ML-yacc if needed and recompile only the parts of the project that have changed, just as *make* does for Unix. To make compiling the compiler even easier a *Makefile* is part of the project which simply invokes the *Makefile.gen* script.

```
make
mlcomp test0.sml
```

To compile and run the *mlcomp* compiler simply type in the mlcomp directory to compile the compiler and run the first test, test0.sml. If all succeeds there will be no errors printed and the program will print nothing to the screen, although other output will be printed like the AST and the compiled and assembled source program.

The remainder of this chapter will cover parts of the code generator that are already implemented and worth taking a look at. It will also cover parts of the compiler that are not yet implemented and suggest how they can be implemented. After working through this chapter you will have a working compiler for the Small language.

6.5 Function Calls

Running *test0.sml* is not very satisfying because no output is printed. Calling a function like *println* will print the output to the screen. The Small language includes a number of functions that can be called for input and output operations. Small differs some from Standard ML in this regard and *println* is one of those differences. Adding a *println* to *test0.sml* results in *println* $5 + 4$, the contents of a file called *test1.sml* in the *mlcomp* distribution file found on Github. Parsing the program results in the AST

```
apply(id("println"),infixexp("+",int("5"),int("4")))
```

Code generation for this program yields the program in Fig. 6.18. The code contains two additional instructions, the *LOAD_GLOBAL* and the *CALL_FUNCTION*

```
1  Function: main/0
2  Constants: None, 'Match Not Found', 0, 5, 4
3  Globals: print, fprint, input, int, len,
4           type, Exception, funlist, concat
5  BEGIN
6      LOAD_GLOBAL 0
7      LOAD_CONST 3
8      LOAD_CONST 4
9      BINARY_ADD
10     CALL_FUNCTION 1
11     POP_TOP
12     LOAD_CONST 0
13     RETURN_VALUE
14 END
```

Fig. 6.18 test1.sml JCoCo code

instructions. The code generator is called for two additional AST nodes as well, the *id* node and the *apply* node. Each of these two calls to *codegen* are provided in Fig. 6.19. The elipses (i.e. …) indicate abbreviated code. The *mlcomp.sml* file can be consulted for the full details.

When *codegen* is called a list of *globals* and *globalBindings* are provided to each call to *codegen*. The *globals* list can be seen in Fig. 6.18 on lines 3 and 4. The *env* in Fig. 6.19 is a list of bindings including a binding of the Small *println* function to a built-in function in JCoCo called *print* which does the same thing in the target language. The *env* list contains the tuple *("println", "print")*. Initially *env* and *globalBindings* are the same list (see line initial call to *codegen* in the *mlcomp.sml* file).

When *codegen* is called for the *apply* in the AST, it immediately calls *codegen* on the *id("println")*.. This results in calling the *load* function which searches all the different bindings to find the binding for *println*. It finds this in the *env* list, finds the corresponding *print* JCoCo function, and looks up *print* in the list of *globals* generating the *LOAD_GLOBAL 0* since it finds *print* in the first position in the *globals* list. The *load* function was written because the type of load necessary depends on where the identifier is found.

The call to *codegen* for *apply* first calls *codegen* to load the print function onto the stack. Then addition code is generated by the call to *codegen* on line 5 of Fig. 6.19. Finally, the *apply* codegen call generates the *CALL_FUNCTION* instruction. There is only one argument passed to any function in the Small language so the *1* is hard-coded.

```
| codegen(id(name),outFile,indent,...,globals,env,globalBindings,...) =
    load(name,outFile,indent,...,freeVars,cellVars,globals,globalBindings,env)
| codegen(apply(t1,t2),outFile,indent,...,globals,env,globalBindings,...) =
    let val _ = codegen(t1,outFile,indent,...,globals,env,globalBindings,...)
        val _ = codegen(t2,outFile,indent,...,globals,env,globalBindings,...)
    in
       TextIO.output(outFile,indent^"CALL_FUNCTION 1\n")
    end
```

Fig. 6.19 Code generation for function calls

```
let val x = 5
in
    println x
end
```

Fig. 6.20 test2.sml

Calling a function is relatively easy as shown in this example. Maintaining and understanding all the bindings is the trickier part, but further examples will serve to make this clearer as well. The next example takes a look at user-defined bindings.

6.6 Let Expressions

Let expressions provide a means for a value or function to be bound to a value. Consider the code in Fig. 6.20 that binds *x* to *5*. This SML program is compiled into the JCoCo program of Fig. 6.21. From the source program, this AST is built.

```
letdec(bindval(idpat("x"),int("5")),
       [apply(id("println"),id("x"))])
```

The AST has the new binding first, followed by the sequence of expressions between the *in* and *end* keywords. In this case there is one expression in the body of the *let* expression. Examining the code in Fig. 6.21 there are two new instructions on lines 7 and 8. These two lines take care of storing the 5 in a local variable called *x@0*. The 0 refers to the scope level of the variable is added to the variable name to be sure that variable names in JCoCo are unique. Line 10 has the *LOAD_FAST* instruction, another new instruction in the program. The *LOAD_FAST* loads from the list of *locals*. We have seen the call to the *load* function in Fig. 6.19 that loads this value from the *locals*.

The Small program contains the binding of *x* to *5*. When compiling this code, the *x* is bound to a location in the locals called *x@0* which contains the *5*. The *let expression* must create this binding to make the *x* visible in the body of the *let*

```
Function: main/0
Constants: None, 'Match Not Found',
           0, 5
Locals: x@0
Globals: print, ...
BEGIN
    LOAD_CONST 3
    STORE_FAST 0
    LOAD_GLOBAL 0
    LOAD_FAST 0
    CALL_FUNCTION 1
    POP_TOP
    LOAD_CONST 0
    RETURN_VALUE
END
```

Fig. 6.21 test2.sml JCoCo code

```
| codegen(letdec(d,L2),...,globals,env,globalBindings,scope) =
  let val newbindings = decgen(d,...,globals,env,globalBindings,scope)
  in
    codegenseq(L2,...,globals,newbindings@env,globalBindings,scope+1)
  end
```

Fig. 6.22 Let expression code generation

expression. It does this in the code in Fig. 6.22 by calling the function *decgen* which generates the code for storing the value in the local location and also creates a new binding *("x","x@0")*. This new binding is added to the *env* environment bindings. When the *load* function is called in the body of the *let* expression, the *x@0* will be found in the list of locals.

Building the list of *locals* for the *main* function is handled by lines 12 in Fig. 6.15. The *locals* is computed in part from the bindings computed by the *localBindings* function. The *localBindings* function traverses the body of a function looking at all identifiers found in the code. If an identifier is free in the body of a function, it is added to the *freeVars* returned by the *localBindings* function. If the identifier is bound to a value or inner function, the bindings is returned in the *newbindings*. If the code passed to the *localBindings* function contains nested functions, then the *freeVars* of those nested functions must be *cellVars* in the current function because a closure will be necessary when the inner function is called. The *localBindings* function finds those identifiers that must be bound to *cellVars* and returns them as well.

```
let val x = 5
    val y = 6
in
  println (x + y)
end
```

Fig. 6.23 test10.sml

```
| Let DecSeq In ExpSequence End
    (List.hd (List.foldr (fn (x,y) => [letdec(x,y)]) ExpSequence DecSeq))
```

Fig. 6.24 The folded let

```
let val x = 5
in
    let val y = 6
    in
        println (x+y)
    end

```

Fig. 6.25 Unsweetened

For the code in Fig. 6.20, the *newbindings* of Fig. 6.22 consist of *[("x","x@0")]* and these bindings are added to the environment *env* when the code for the body of the *let* declaration is generated. The list of the *locals* already is set to *["x@0"]* so when the *load* function is called to load the value of *x*, the combination of the environment *env* and the *locals* results in the correct index being found to generate the *STORE_FAST* and *LOAD_FAST* instructions.

With Standard ML it is possible to define more than one value in a *let* expression. Consider the program in Fig. 6.23. This program has two bindings created in one *let* expression. However, the program is not the program compiled by the *mlcomp* compiler. The parser transforms this program into a program like the one given in Fig. 6.25. The ability to write a program like Fig. 6.23 is called *syntactic sugar*. It is certainly nicer to write programs like that in Fig. 6.23 rather than being limited to one binding per let expression all the time. However, the Small abstract syntax does not include support for multiple bindings. That's what is meant by *syntactic sugar*. When a programming language feature like *let* expressions of multiple bindings is implemented in terms of some other simpler but less desirable form it is called *syntactic sugar*. The *mlcomp* compiler handles multiple bindings by using a *foldr* call to fold those multiple bindings into multiple nested *let* expressions. Figure 6.24 contains the code in the parser that forms this folded *let*.

6.7 Unary Negation

It turns out that unary negation is not implemented correctly in the *mlcomp* compiler. Presently, it is possible to print a negative 5. However, the program in Fig. 6.26 should compile and run, but instead the scanner deletes the ~ as a bad token and a 5 is printed to the screen instead. This is not the behavior of Standard ML. The tilde serves as a unary negation operator in Standard ML. To fix this, several changes are necessary. Starting with the scanner, the tilde must be recognized as its own token. To do this, the tilde is removed from the *Int* token and added as its own token in the *mlcomp.lex* file.

```
{tilde} => (Tokens.Negate(!pos,!pos));
{digit}+   => (Tokens.Int(yytext,!pos,!pos));
```

Adding the token in the parser specification is next. So the tokens are now defined as follows in *mlcomp.grm*

```
%term EOF
    | Negate
    | ...
```

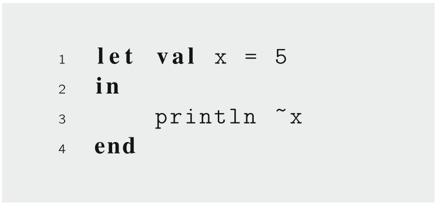
Then we define the precedence of the *Negate* token in *mlcomp.grm*. Unary negation has very high precedence and is right-associative.

```
%right ListCons Negate
```

The last bit in the *mlcomp.grm* is to write a production that uses the *Negate* token. To negate an expression we just write an expression as possibly being negated as in this bit of code.

```
| Negate Exp          (negate(Exp))
```

Writing this production requires a new node definition for the AST in *mlast.sml*. A *negate* node in an AST is another kind of expression. Unary negation can be represented by defining another expression for *negate* as follows.



```
let val x = 5
in
    println ~x
end
```

Fig. 6.26 test3.sml

```
| negate of exp
```

Finally, to finish the correct implementation of unary negation, the code generator module must be modified. The *mlcomp.sml* file must be edited in a few spots to add support for unary negation. The *infixexp* expression is an AST node like the *negate* node. Searching for *infixexp* in the *mlcomp.sml* file helps determine where the changes must be made in *mlcomp.sml*. The first change is in the *nameOf* function.

```
| nameOf(infixexp(operator,e1,e2)) = operator
| nameOf(negate(e)) = "~"
```

The next match is found inside the *constants* function where this code must be added.

```
| con(infixexp(operator,t1,t2)) = (con t1) @ (con t2)
| con(negate(e)) = "0" :: (con e)
```

This code adds a zero to the list of constants. This is because to implement unary negation the generated code will subtract the value from zero. The *bindingsOf* function is the next location where *infixexp* appears in the *mlcomp.sml* file. The code to write here looks like this.

```
| bindingsOf(infixexp(operator,exp1,exp2),bindings,scope) =
        (bindingsOf(exp1,bindings,scope); bindingsOf(exp2,bindings,scope))
| bindingsOf(negate(exp),bindings,scope) = bindingsOf(exp,bindings,scope)
```

The *bindingsOf* function is looking for any new bindings introduced by the new unary negation expression. There are no new bindings created by Unary negation so it just calls the *bindingsOf* function on its sub-expression. The *codegen* function is the next place where *infixexp* is found and the following code is added to generate code for unary negation.

```
| codegen(negate(t),outFile,indent,consts,...) =
  let val _ = codegen(int("0"),outFile,indent,consts,...)
      val _ = codegen(t,outFile,indent,consts,...)
  in
    TextIO.output(outFile,indent^"BINARY_SUBTRACT\n")
  end
```

In the *codegen* function a "fake" *int("0")* node is created to get a zero loaded onto the stack. Then the value for the sub-expression is loaded onto the stack and the *BINARY_SUBTRACT* instruction causes the unary negation to be computed. Both the *nestedfuns* and the *makeFunctions* function need a line for unary negation added as well. In both cases the code is identical and looks like this:

```
| functions(infixexp(operator,exp1,exp2)) = (functions exp1;functions exp2)
| functions(negate(exp)) = functions exp
```

The *nestedfuns* code is looking for any nested functions within the expression. Unary negation is not a nested function so the code just calls the check by calling the *functions* function on the sub-expression. The *makeFunctions* function generates some code for any nested functions to have JCoCo create the closure or function objects for any nested functions. Finally, the *writeTerm* function must be modified. While not needed by the compiler, the *writeTerm* function is useful when reading Chap. 8. Here is the code for writing a unary negation term.

```
1  Function: main/0
2  Constants: None, 'Match Not Found', 5, 0
3  Locals: x@0
4  Globals: print, ...
5      LOAD_CONST 2
6      STORE_FAST 0
7      LOAD_GLOBAL 0
8      LOAD_CONST 3
9      LOAD_FAST 0
10     BINARY_SUBTRACT
11     CALL_FUNCTION 1
12     POP_TOP
13     LOAD_CONST 0
14     RETURN_VALUE
15 END
```

Fig. 6.27 test3.sml JCoCo code

```
| writeExp(indent,negate(exp)) =
           (print("negate(");
            writeExp(indent,exp);
            print(")"))
```

The final result of these changes is code as it appears in Fig. 6.27. The value of ~*x* is computed by subtracting from 0. The new code consists of lines 8 and 10 in the JCoCo code in Fig. 6.27.

6.8 If-Then-Else Expressions

Comparing two values in SML is as simple as writing $x < y$. In JCoCo it involves pushing two values on the operand stack and calling the *COMPARE_OP* instruction. When comparing values in an *if-then-else* expression the result of the comparison will be used to jump to one label or another. Consider the Small program in Fig. 6.28. Again, this code differs a bit from Standard ML. The *input* function is unique to Small as are the *print* and *println* functions. The *input* function returns a string of input from the user. The *print* function prints without a newline character. The *println* prints with a newline at the end of the line.

Compiling the code in Fig. 6.28 should result in the JCoCo code in Fig. 6.29. However, code generation for *if-then-else* expressions is not currently implemented. The abstract syntax tree for the program in Fig. 6.28 includes a node for the *if-then-else* expression like this.

```
ifthen(infixexp(">",id("x"),id("y")),id("x"),id("y"))
```

The AST definition for this program is already in the *mlast.sml* file and the scanner and parser are already able to parse *if-then-else* expressions. Generating code for this AST involves some of the same changes that were needed to add unary negation to the code generator. Those steps can be followed to add all the necessary code to handle *if-then-else* expressions in the code generator. By attempting to compile the code in Fig. 6.28 you will discover places in the compiler where code is missing. The compiler is written to report where code is missing. Attempt to compile *test4.sml*, see where the problem is, fix it, and repeat as many times as is necessary.

Implementing the *codegen* code is the hardest part of adding support for *if-then-else* expressions, but it's not too hard. The AST expression above has three sub-expressions: the greater than comparison, the *id("x")*, and the *id("y")*. Code generation is already done for identifiers so the *id* nodes for *x* and *y* are already handled. Generating code for the *if-then-else* expression involves generating the code for the comparison and then jumping to one place or another depending on the result of the comparison.

The *if-then-else* generated code begins on line 26 of Fig. 6.29 with the comparison code. Calling *codegen* on the infix expression generates the code on lines 26–28. Line 29 begins some of the code for the *if-then-else* expression. Line 29 begins by jumping to *L0* if the condition is false. The label *L0* labels the *else* clause of the expression. Line 30 is the code generated for the *id("x")* which is the *then* expression. Line 31 is generated by the *if-then-else* again to jump past the code in the *else* expression. Line 34 is the last bit of code generated by the *if-then-else* expression.

There are two labels needed by the code generator. The *nextLabel* function in *mlcomp.sml* is designed just for that purpose. Calling it will return a unique label that can be used in the code. Code generation for *if-then-else* expressions calls this function twice. In summary, there are several actions that must occur to generate code for *if-then-else* expressions.

```
let val x = Int.fromString(
              input("Please enter an integer: "))
    val y = Int.fromString(
              input("Please enter an integer: "))
in
  print "The maximum is ";
  println (if x > y then x else y)
end
```

Fig. 6.28 test4.sml

```
Function: main/0
Constants: None, 'Match Not Found',
  0, "Please enter an integer: ",
  "The maximum is "
Locals: y@1, x@0
Globals: print, fprint, input, int, len,
  type, Exception, funlist, concat
BEGIN
    LOAD_GLOBAL 3
    LOAD_GLOBAL 2
    LOAD_CONST 3
    CALL_FUNCTION 1
    CALL_FUNCTION 1
    STORE_FAST 1
    LOAD_GLOBAL 3
    LOAD_GLOBAL 2
    LOAD_CONST 3
    CALL_FUNCTION 1
    CALL_FUNCTION 1
    STORE_FAST 0
    LOAD_GLOBAL 1
    LOAD_CONST 4
    CALL_FUNCTION 1
    POP_TOP
    LOAD_GLOBAL 0
    LOAD_FAST 1
    LOAD_FAST 0
    COMPARE_OP 4
    POP_JUMP_IF_FALSE L0
    LOAD_FAST 1
    JUMP_FORWARD L1
L0:
    LOAD_FAST 0
L1:
    CALL_FUNCTION 1
    POP_TOP
    LOAD_CONST 0
    RETURN_VALUE
END
```

Fig. 6.29 test4.sml JCoCo code

- Two labels need to be created.
- The comparison code is generated.
- The *POP_JUMP_IF_FALSE* instruction is written along with the *else* clause label.
- The *then* clause code is generated.
- A jump to jump past the *else* clause code is written.
- The *else* clause label is written.
- The *else* clause code is generated.
- The final label is written to the file.

Successfully completing this code will get *if-then-else* expressions compiling correctly and *test4.sml* will run printing the maximum of two numbers entered at the keyboard.

6.9 Short-Circuit Logic

Short-circuit logic is a common feature of programming languages. If you have two boolean expressions, *E1* and *E2*, and you want to know if both are true or false there are situations where it is not necessary to test both the conditions. For instance, when testing *E1 and E2* if *E1* is false, there is no reason to evaluate *E2*. Likewise, if evaluating *E1 or E2* if *E1* is true there is no reason to evaluate *E2*. This logic is called *short-circuit logic* and is commonly used by *and* and *or* operators in programming languages. C++ and Java use this logic in their && and || operators. In Standard ML the operators are called *andalso* and *orelse* to indicate their short-circuit nature.

Neither the *andalso* or *orelse* operators are implemented in the *mlcomp* compiler. Support can be added pretty easily by following many of the steps in adding unary negation to the language. These steps include:

- Add two tokens for *andalso* and *orelse* to the scanner. Both are keywords and should be added to the keywords section of the scanner specification in *mlcomp.lex*.
- Add the tokens to the grammar specification in *mlcomp.grm* and define their precedence. Both operators have the same precedence which is at the same level as addition. They are also both left-associative.
- Add two productions to the grammar so the expressions can be parsed. The productions should return AST nodes as described next.
- Implement the code generation for these operators.

A correctly generated AST for this code will both include *infixexp* nodes like this.

```
infixexp("orelse",id("x"),
    infixexp("div",id("y"),int("0")))
infixexp("andalso",id("y"),
    infixexp("*",id("x"),int("5")))
```

The code for line 4 of Fig. 6.30 starts on line 14 of Fig. 6.31. The *print* function is loaded first. This is already implemented of course. Line 15 begins the code

```
let val x = true
    val y = false
in
  println (x orelse y div 0);
  println (y andalso x * 5)
end
```

Fig. 6.30 test5.sml

generation for the *orelse* operator. For the expression *E1 orelse E2* the code for *E1* is generated first, followed by *DUP_TOP*, *POP_JUMP_IF_TRUE*, and the *POP_TOP* instructions. The idea is if the first value is true, leave it on the stack and skip evaluating *E2*. However, if the value of *E1* is false, pop its value, and leave the value of *E2* on the stack after executing the code for *E1 orelse E2*.

A label is needed as the target for the jump instruction. The *nextLabel* function returns a unique label as was described in Sect. 6.8 on compiling *if-then-else* expressions.

The code for *andalso* appearing on lines 26–33 of Fig. 6.31 is analogous to the *orelse* code jumping if the first value is false and evaluating *E2* if *E1* is true.

The program in Fig. 6.30 is of some interest because it is not a valid Small program, yet the *mlcomp* compiler will generate code and it is possible to run the program on the JCoCo virtual machine. Since the short-circuit logic prevents the badly typed expressions from being evaluated, the error is never encountered. Chapter 8 will explore how the program in Fig. 6.30 fails to pass typechecking by looking at how the Standard ML type inference algorithm is implemented.

The difference between Python and Standard ML is that Python will allow a program like this to run as long as no run-time error occurs and Standard ML will complain that it doesn't pass type checking and will abort. Is the type inference of Standard ML better than the dynamic type checking of Python? Type inference catches many errors in logic. Debugging most Standard ML programs is trivial compared to debugging Python programs. However, passing the type checker is often more difficult and often requires tedious type conversion code. Standard ML is a bit better in that regard given its polymorphic type inference algorithm. In general, research like the Fox project at Carnegie Mellon has shown that large software systems benefit enormously from strong type checking by reducing the time it takes to test code.

The tradeoff is in convenience vs safety while writing code and the amount of time spent testing and debugging after the code is written. Standard ML is somewhat less convenient for writing, but debugging costs are negligible. Python is more convenient to write but in a large software system you might pay for it later. Other factors in language selection include appropriateness for the task at hand, whether similar code has already been written in a particular language, the existence of libraries

```
Function: main/0
Constants: None,
   'Match Not Found',
   True, False, 0, 5
Locals: y@1, x@0
Globals: print, fprint, input,
   int, len, type, Exception,
   funlist, concat
BEGIN
    LOAD_CONST 2
    STORE_FAST 1
    LOAD_CONST 3
    STORE_FAST 0
    LOAD_GLOBAL 0
    LOAD_FAST 1
    DUP_TOP
    POP_JUMP_IF_TRUE L0
    POP_TOP
    LOAD_FAST 0
    LOAD_CONST 4
    BINARY_FLOOR_DIVIDE
L0:
    CALL_FUNCTION 1
    POP_TOP
    LOAD_GLOBAL 0
    LOAD_FAST 0
    DUP_TOP
    POP_JUMP_IF_FALSE L1
    POP_TOP
    LOAD_FAST 1
    LOAD_CONST 5
    BINARY_MULTIPLY
L1:
    CALL_FUNCTION 1
    POP_TOP
    LOAD_CONST 0
    RETURN_VALUE
END
```

Fig. 6.31 test5.sml JCoCo code

providing APIs, and the availability of tools like compilers, interpreters, and IDEs (i.e. Integrated Development Environments). All these factors must be weighed to decide what language is most appropriate for a project.

6.10 Defining Functions

Function definitions in Standard ML may appear literally anywhere within the program. Functions are first class values and may appear anywhere a declaration may appear. In addition, anonymous functions may appear anywhere an expression may appear in an SML program. Not so in JCoCo. In the JCoCo virtual machine function definitions may be provided at the top level, outside any other functions, or may be nested inside another function but must be written immediately after the *Function* statement of their outer function. In addition, in JCoCo all functions must be named. There are no anonymous functions.

The *nestedfuns* function traverses an AST for an SML expression looking for any function definitions. If it finds one it generates the code for the nested function immediately. Consider the *compile* function of the *mlcomp.sml* module.

```
TextIO.output(outFile,"Function: main/0\n");
nestedfuns(ast,outFile,"   ",globals,[],globalBindings,0);
```

This code prints the *Function* statement for the *main* function. Then it immediately called the *nestedfuns* function to look for any nested functions and generate their code before continuing with the code generation for the *main* function. Again, this is the order required by the JCoCo virtual machine. When a nested function definition is found in the AST, the *nestedfun* function is called to generate the code for it. There is too much code to include here, but the *nestedfun* function gathers information about the constants, locals, cell variables, and bindings of the inner function before calling *codegen* to generate the body of it. Of course, it also looks for any nested functions inside it before continuing.

When an anonymous function is found it must be assigned a name since that is required by the JCoCo virtual machine. Naming anonymous functions occurs in the parser in the production for anonymous functions.

```
Fn MatchExp (func(nextIdNum(),MatchExp))
```

```
let fun factorial 0 = 1
      | factorial n = n * (factorial (n-1))
in
  println (factorial 5)
end
```

Fig. 6.32 test6.sml

```
Function: main/0
    Function: factorial/1
    Constants: None,
        'Match Not Found', 0, 1
    Locals: factorial@Param, n@1
    FreeVars: factorial
    Globals: print, fprint, input,
        int, len, type, Exception,
        funlist, concat
    BEGIN
        LOAD_FAST 0
        LOAD_CONST 2
        COMPARE_OP 2
        POP_JUMP_IF_FALSE L0
        LOAD_CONST 3
        RETURN_VALUE
L0:
        LOAD_FAST 0
        STORE_FAST 1
        LOAD_FAST 1
        LOAD_DEREF 0
        LOAD_FAST 1
        LOAD_CONST 3
        BINARY_SUBTRACT
        CALL_FUNCTION 1
        BINARY_MULTIPLY
        RETURN_VALUE
L1:
        LOAD_GLOBAL 6
        LOAD_CONST 1
        CALL_FUNCTION 1
        RAISE_VARARGS 1
    END
...
```

Fig. 6.33 test6.sml JCoCo code

In this code the *nextIdNum* function returns a unique integer. In the code generator this unique integer is used to form a name for the anonymous function of *anon@i* where *i* is the unique integer assigned by the parser.

Function definitions are always defined for functions of exactly one argument. Pattern matching may be used in matching the argument as it is in Standard ML. The parameter of the function is matched to each pattern in the function definition. Consider the code in Fig. 6.32. There are two patterns in the function definition, a number pattern, and an identifier pattern, which always matches. The *patMatch* function in *mlcomp.sml* takes care of generating code to match the argument to the pattern.

For the number pattern, the code on lines 12–14 of Fig. 6.33 checks to see if the number matches. If not, the code jumps to the end of its case. There is no code to check the identifier pattern matching because it always matches.

Take note of the code on lines 28–32 of Fig. 6.33. Each time the *patmatch* code is called it is passed the label of the next pattern to jump to if the current pattern does not match. In this case, the last pattern always matches, but if it hadn't the code might have jumped to *L1*. In that case, since all the patterns are exhausted at that point, an exception would be raised by the code. In this particular function, lines 28–32 are an example of *dead code*. The code will never be reached and could be removed.

The *patmatch* function matches patterns for *nil*, numbers, true or false, strings, identifiers, the :: cons operator (i.e. a non-empty list pattern), and tuples. The tuple pattern in turn matches each element of the tuple pattern to the elements of the tuple argument by calling *patmatch*.

6.10.1 Curried Functions

It was said earlier that all functions are functions of one argument in Small (and in Standard ML as well) and it's true. Curried functions are another example of *syntactic sugar*. A curried function appears to be a function of more than one argument where the arguments can be provided one at a time. The truth is that a curried function is transformed into a series of anonymous functions, each of one argument. Consider the program in Fig. 6.34. The *append* function is written in curried form. *appendOne* is a function of one argument. When the program is run they both do exactly the same thing appending two lists together. Calling *append* and *appendOne* look identical. That's because the two functions are identical. Function application is left associative so each function is applied to its first, and only, argument which returns a function that is applied to its second argument.

The *mlcomp parser* reduces curried functions like *append* to a function of one argument with one anonymous function for each of the curried arguments. This is done via a rather complex function that gathers each of the different pattern matches of a curried function and rewrites the code so that each pattern match is a pattern match of exactly one argument returning a function that takes the next argument. This function is called *uncurryIt* and is given in Fig. 6.35 on page 213.

```
let
  fun append nil L = L
    | append (h::t) L =
        h :: (append t L)

  fun appendOne x =
    (fn nil => (fn L => L)
     | h::t => (fn L =>
         h :: (appendOne t L))) x
in
  println(append [1,2,3] [4]);
  println(appendOne [1,2,3] [4])
end
```

Fig. 6.34 test7sml

```
fun uncurryIt nil = raise emptyDecList
  | uncurryIt (L as ((name,patList,exp)::t)) =
    let fun len nil = raise argumentMismatch
          | len [(n,p,e)] = length(p)
          | len ((n,p,e)::t) =
            let val size = length(p)
            in
              if size = len t then size else
                (TextIO.output(TextIO.stdOut,
                     "Syntax Error: Number of arguments does not match ...."
                 raise argumentMismatch)
            end

        val tupleList = List.map (fn x => "v"^Int.toString(nextIdNum())) patList
     in
       len(L); (* check that all patterns have same length *)
       (name,[match(idpat(hd(tupleList)),
                 List.foldr (fn (x,y) => func(nextIdNum(),[match(idpat(x), y)]))
                    (apply (func(nextIdNum(),List.map (fn (n,p,e) =>
                        match(tuplepat(p),e)) L),
                           tuplecon(List.map
                           (fn x => id(x)) tupleList))) (tl tupleList))])
     end
```

Fig. 6.35 The uncurryIt function

6.10.2 Mutually Recursive Functions

Functions in Small and SML are often recursive. Sometimes, functions may be mutually recursive as is the case in Fig. 6.36. The function f calls g and vice versa. In C++, to write two functions like this, a forward declaration is required using the function prototype for at least g. In Standard ML, the use of the *and* keyword between the two function definitions indicates that they are mutually recursive functions. The AST for this program is specified like this:

```
let fun f(0,y) = y
      | f(x,y) = g(x,x*y)
    and g(x,y) = f(x-1,y)
in
  println (f(10,5))
end
```

Fig. 6.36 test11.sml

```
| dec(funmatches(L)) =
  let val nameList = List.map (fn (name,matchlist) => name) L
  in
    List.map (fn (name,matchList) =>
    let val adjustedBindings =
        List.map (fn x => (x,x)) (listdiff nameList [name])
    in
      nestedfun(name,matchList,outFile,indent,globals,
          adjustedBindings@env,globalBindings,scope)
    end) L;
    ()
  end
```

Fig. 6.37 Mutually recursive function declarations

```
letdec(funmatches([funmatch("f",f's body),funmatch("g",g's body)]))
```

When a *funmatches* AST node is encountered, the bindings of all the functions in the *funmatches* list are passed to the code generation of each function. This is seen in the *nestedfuns* function when matching a declaration for a *funmatch* as shown in Fig. 6.37.

In this code the list of all function names is gathered in *nameList* and then passed to each recursive call of *nestedfun* after taking out the name of the function on which *nestedfun* is being called. Mutually recursive functions are more common than you might think. Look for uses of *and* in the *mlcomp.sml* file to see when it is needed in the implementation of the compiler.

6.11 Reference Variables

Adding variables to the Small language turns out to be almost trivial. Examining Fig. 6.38 the new code involves the *ref* keyword, the exclamation point used as the dereference operator, and the := operator (pronounced *set equal*). The scanner includes support for the dereference and the set equal operators. The *ref* will be recognized as an identifier, which turns out to be just fine.

The grammar specification in *mlcomp.grm* already has support for both the dereference and set equal operators. The productions for the two are of some interest. In Fig. 6.39 the set equal production demands that an identifier be on the left hand side. A variable cannot be an expression. If the reference variable is to point to a new value, the left hand side must name the reference variable. Yet, the AST is an *infixexp* by creating an expression node from the identifier using *id(Id)*. The dereference production is even more interesting creating a fake function application node with a *!* identifier. No production is needed for the *ref* keyword addition because the grammar already parses this as function application of the *ref* function to the value *0*.

Code generation for variables is handled by a series of special cases. The *decBindingsOf* function must be modified because binding for a variable is different than the binding for a regular identifier. The code in Fig. 6.40 must be placed before the pattern for regular identifiers.

The code in Fig. 6.40 binds the variable name to a unique identifier in the JCoCo program and it adds the variable name to the list of identifiers that will be associated with cell variables. A cell variable is a reference and variables are references in Standard ML.

The dereference operator must be handled as a special case in the *bindingsOf* function. Normally an identifier is looked up to see if it is bound or free in a function. The parser generated AST for the dereference operator makes it look like an identifier in Fig. 6.39. To handle this, the following code is a special case and must appear before the normal look up of identifiers in the *bindingsOf* function.

```
let val x = ref 0
in
  x := !x + 1;
  println (!x)
end
```

Fig. 6.38 test8.sml

```
| Exclaim Exp (apply(id("!"),Exp))
| Id SetEqual FuncExp
             (infixexp(":=",id(Id),FuncExp))
```

Fig. 6.39 Set equal and deref operators

```
and decbindingsOf(bindval(idpat(name),apply(id(1+s+s2{r}ef"),exp)),bindings,scope) =
    let val newbindings = patBindings(idpat(name),scope)
    in
      bindingsOf(exp,newbindings@bindings,scope+1);
      addIt(name,cellVars);
      [addIt((name,name^1+s+s2{@}n{^}Int.toString(scope)),theBindings)]
    end
```

Fig. 6.40 Reference variable bindings

```
1   | codegen(id(name),outFile,...) =
2     load(name,outFile,...)
3   | codegen(apply(id("ref"),t2),outFile,...) =
4     codegen(t2,outFile,...)
5   | codegen(infixexp(":=",id(name),t2),...) =
6   let val _ = codegen(t2,outFile,...)
7         val noneIndex = lookupIndex("None",consts)
8   in
9      store(name,outFile,...);
10     TextIO.output(outFile,
11         indent^"LOAD_CONST "^noneIndex^"\n")
12  end
```

Fig. 6.41 Variable code generation

```
| bindingsOf(id("!"),bindings,scope) = ()
```

Finally, code generation must be done for the *ref* declaration, the dereference operator, and the set equal operator. The *ref* code generation is another special case and must be done before normal function application. What is interesting is that all the work of code generation was actually done by the *decBindingsOf* function when the variable was added to the cell variables list. In lines 1–2 of Fig. 6.41, the code for a *ref* expression is identical to the code for a *non-reference* expression because the *store* function will find the variable in the cell variables and then generate the appropriate store instruction.

Lines 3–4 generate the code for dereferencing a variable. Indirectly, this calls *load* which will automatically generate the appropriate load instruction because the *decBindingsOf* function placed the variable in the list of cell variables. Finally, the code for the set equal operator is pretty straightforward. The *LOAD_CONST* instruction is needed because every expression in Standard ML has a result and at the end of the assignment statement the result is popped from the stack. The result of assignment is *unit* which translates to the *None* value in the JCoCo virtual machine.

When a binding to an identifier is used in an inner function, the identifier must be bound to a cell variable so a closure can be constructed when the inner function is

```
let val x = 0
    fun f y = (x:=!x+1)
in
  f 0;
  println x
end
```

Fig. 6.42 test9.sml

called. Reference variables are also bound to cell variables so they can be updated. Having two different sorts of bindings both map to the same implementation leads to some interesting possibilities in the code. Consider the program in Fig. 6.42. This program is not a legal Small program. The binding of *x* to 0 is a constant binding. It should not be possible to update the contents of the variable. However, the assignment statement on line 2 works because *x* is used in the inner function *f* and therefore is assigned to a cell variable.

The code in Fig. 6.42 is an example of when type checking is needed to prevent an illegal program from executing. The program is incorrect. The programmer made a mistake and would like to know about this mistake. Yet JCoCo doesn't care and neither does the *mlcomp* compiler. A typechecker should flag this as an error and terminate the code generator before any program is generated. This example, and the need for type checking, will be studied in more detail in Chap. 8.

6.12 Chapter Summary

The goal of the chapter was to provide an introduction to language features by studying the implementation of the Small language. Those wishing to learn more about compiler construction may want to consult a full text on the subject. For instance Aho, Sethi, and Ullman's dragon book [2]. There are many other good texts on compiler writing as well.

The case study in this chapter illustrated several features of programming languages. The implementation of functions in block structured languages is perhaps the most difficult of the concepts presented. Important concepts and skills presented in this chapter include the scope of bindings and how bindings are created, mutually recursive functions, reference variables, code generation for several language features, how to extend a language, how to use ML-lex and ML-yacc, syntactic sugar and its uses in the Small language, and short-circuit logic. Exception handling was not covered in this chapter and is a part of the mlcomp compiler.

As the need for embedded systems grows so will the demand for new programming languages targeting those platforms. The demands of a fast-paced work environment have also spurred interest in programming language design and development. This is an exciting time for experts in programming languages and this text only scratches the surface of a vast and exciting area of study.

6.13 Review Questions

1. The language of regular expressions can be used to define the tokens of a language. Give an example for a regular expression from the chapter and indicate what kind of tokens it represents.
2. What does ML-lex do? What input does it require? What does it produce?
3. Why do keywords have to be recognized by an if-else-if statement in the ML-lex definition? Why couldn't each keyword just be recognized like other fixed tokens in a language?
4. How is an abstract syntax tree declared in ML?
5. Using the grammar specification for Small, what is the AST of the following expression?

```
fun abs(x) = if x > 0 then x else ~1*x
```

6. How does the load function of the code generator decide which load instruction to generate?
7. In the code generation for function calls in Fig. 6.19, what is the purpose of the two recursive calls to *codegen*?
8. Which function in the code generator is responsible for returning the new bindings created by a *let* expression?
9. What does it mean for the *Small* language to support short-circuit logic? What happens in the code generation?
10. In Fig. 6.37 what do *nameList* and *adjustedBindings* refer to for the program given in Fig. 6.36? Give the actual contents of the three lists? Why three lists?

6.14 Exercises

1. Modify the compiler to support unary negation as described in this chapter. Upon completion *test3.sml* should compile and run correctly.
2. Add >=, <=, and <> (not equal) operators to the Small language. Provide all the pieces in all the files so programs using these operators can be compiled. Write a Small program that demonstrates that this functionality works.

3. Add support for *if-then-else* expressions to the Small compiler as described in this chapter. Follow the instructions of the chapter and be sure to test your implementation using *test4.sml*.
4. Implement short-circuit logic as described in this chapter for the *andalso* and the *orelse* operators.
5. Follow the step in this chapter to add support for compiling expressions with variables. Then, implement a *while do* loop for the *mlcomp* compiler. A while loop is written *while Exp1 do Exp2*. The *Exp1* expression is evaluated first to see if it yields true. If it does, then *Exp2* is evaluated. This repeats until *Exp2* returns false. Remember your job is to generate code for a while loop, not execute it. Use examples like adding *if-then-else* to help you determine where the changes need to be made to add support for *while do* loops. Successfully writing this code will result in successfully compiling and running test12.sml.
6. Add support for *case* expressions in the *mlcomp* Small compiler. The concrete syntax of a case statement is

```
Expression : ...
  | Case Exp Of MatchExp   (caseof(Exp,MatchExp))
```

while the abstract syntax of a case expression is given here.

```
caseof of exp * match list
```

Follow an example like adding support for unary negation to see what all is required to support the *case* expression in JCoCo. Write a program to test the use of the *case* expression in your code. There is currently no support for case expressions in the mlcomp compiler. This project will require you to add support to all facets of the compiler including the scanner, parser, and code generator. When you have successfully implemented the code to parse and compile case expressions, you will be able to compile this program which is test15.sml in the mlcomp distribution.

```
let val x = 4
in
  println
    (case x of
       1 => "hello"
     | 2 => "how"
     | 3 => "are"
     | 4 => "you")
end
```

The generated code for this program is given below. The program, when run, will print *you* to the screen.

```
Function: main/0
Constants: None, 'Match Not Found', 0, 1, "hello", 2, "how", 3, "are", 4, "you"
Locals: x@0
Globals: print, fprint, input, int, len, type, Exception, funlist, concat
BEGIN
    LOAD_CONST 9      # Here the 6 is stored in x.
    STORE_FAST 0
    LOAD_GLOBAL 0     # This is the println pushed onto stack.
    LOAD_FAST 0       # x is loaded onto stack.
    DUP_TOP           # Case expression code where x's value is duplicated.
    LOAD_CONST 3      # This is a pattern match for the first pattern.
    COMPARE_OP 2
    POP_JUMP_IF_FALSE L1
    POP_TOP           # Case expression code to pop x from stack
    LOAD_CONST 4      # This is the expression for the first match.
    JUMP_FORWARD L0   # Case expression code to jump to end of case.
L1:                   # Case expression code for label for end of first pattern.
    DUP_TOP           # Case expression code where x's value is duplicated.
    LOAD_CONST 5      # This is a pattern match for the second pattern.
    COMPARE_OP 2
    POP_JUMP_IF_FALSE L2
    POP_TOP           # Case expression code to pop x from stack
    LOAD_CONST 6      # This is the expression for the second match.
    JUMP_FORWARD L0   # Case expression code to jump to end of case.
L2:                   # Case expression code for label for end of second pattern.
    DUP_TOP           # Case expression code where x's value is duplicated.
    LOAD_CONST 7      # This is a pattern match for the third pattern.
    COMPARE_OP 2
    POP_JUMP_IF_FALSE L3
    POP_TOP           # Case expression code to pop x from stack
    LOAD_CONST 8      # This is the expression for the third match.
    JUMP_FORWARD L0   # Case expression code to jump to end of case.
L3:                   # Case expression code for label for end of third pattern.
    DUP_TOP           # Case expression code where x's value is duplicated.
    LOAD_CONST 9      # This is a pattern match for the fourth pattern.
    COMPARE_OP 2
    POP_JUMP_IF_FALSE L4
    POP_TOP           # Case expression code to pop x from stack
    LOAD_CONST 10     # This is the expression for the fourth match.
    JUMP_FORWARD L0   # Case expression code to jump to end of case.
L4:                   # Case expression code for label for end of fourth pattern.
L0:                   # This is the end of case expression label.
    CALL_FUNCTION 1   # print the result which was left on the stack
    POP_TOP           # Pop the None left by println
    LOAD_CONST 0      # Push a None to return
    RETURN_VALUE      # Return the None
END
```

7. The following program does not compile correctly using the mlcomp compiler and type inference system. However, it is a valid Standard ML program. Modify the mlcomp compiler to correctly compile this program.

```
let val [(x,y,z)] = [(1+s+s2{h}ellop{,}1,true)] in println x end
```

8. Currently, the abstract syntax and parser of *Small* includes support for the wildcard pattern in pattern matching, but the code generator does not support it. Add support for wildcard patterns, write a test program, and test the compiler and code generation.
9. Currently, the abstract syntax and parser of *Small* includes support for the *as* pattern in pattern matching, but the code generator does not support it. Add support for *as* patterns, write a test program, and test the compiler and code generation. The *as* pattern comes up when you write a pattern like *L as h::t* which assigns *L* as a pattern that represents the same value as the compound pattern of *h::t*.

6.15 Solutions to Practice Problems

These are solutions to the practice problem s. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

6.15.1 Solution to Practice Problem 6.1

The keywords *case* and *of* must be added to the scanner specification in *mlcomp.lex*. All the other tokens are already available in the scanner.

6.15.2 Solution to Practice Problem 6.2

You need to add a new AST node type.

```
| caseof of exp * match list
```

6.15.3 Solution to Practice Problem 6.3

The grammar changes required for case expressions are as follows.

```
Expression : ...
  | Case Exp Of MatchExp  (caseof(Exp,MatchExp))
```