Programming Languages: An Active Learning Approach ©2008, Springer Publishing

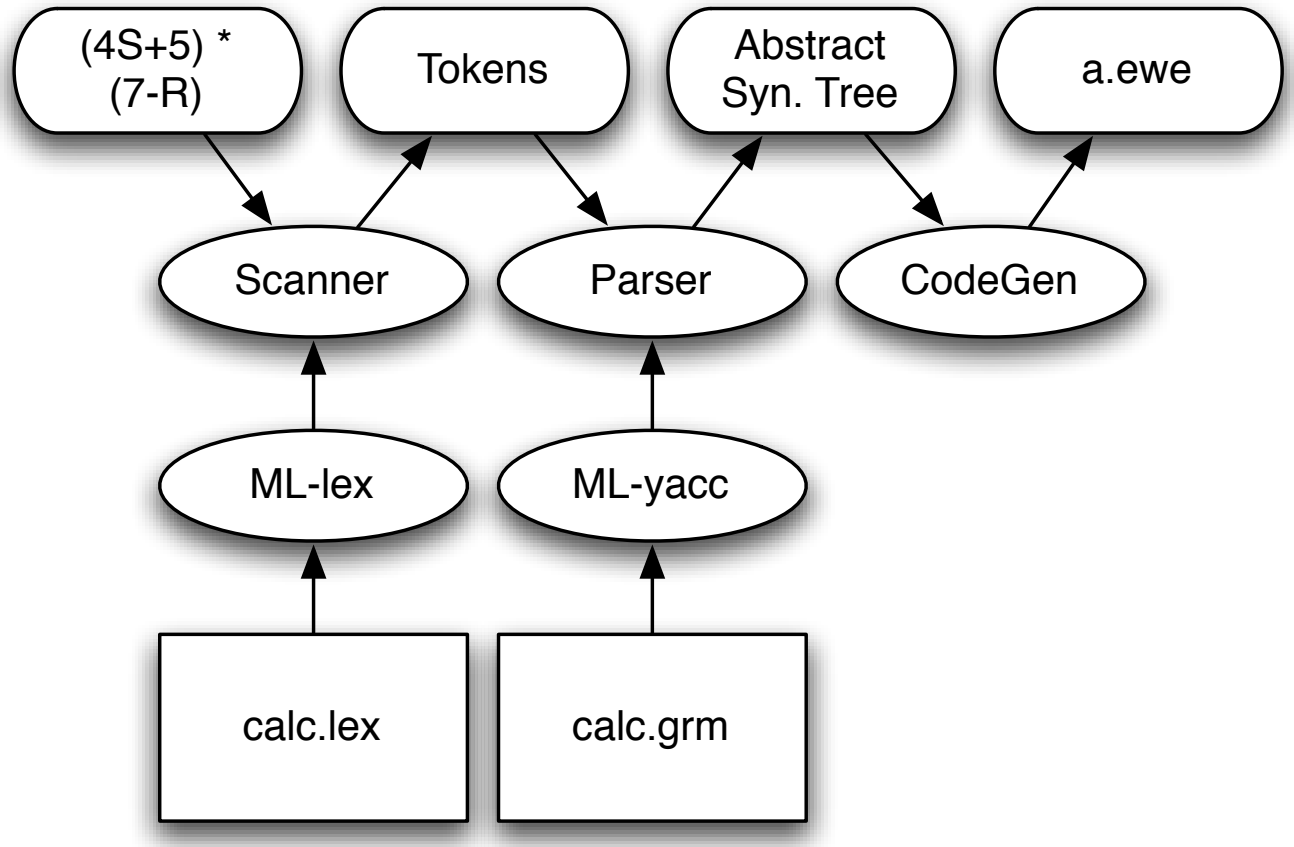
Chapter 6 Language Implementation in Standard ML

Kent D. Lee

Language Implementation in Standard ML

- SML was designed as a language for describing languages when it was used as part of the Logic for Computable Functions (LCF) system.
- Standard ML is an excellent language choice for implementing interpreters and compilers.
- There are two very nice tools for SML that will generate a scanner and a parser given a description of the tokens and grammar of a language.
- This chapter introduces these two tools through a case study involving the development of a simple compiler for the calculator language presented in previous chapters.

Structure of the Calculator Compiler implemented in ML



Language Implementation in Standard ML

1. The tokens of the language are defined in a file called `calc.lex`.
2. The datatype for expression ASTs is defined in a file called `calcast.sml`.
3. The grammar of the language is defined in a file called `calc.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.

Using ML-lex

- The format of an ML-lex input file is:

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

- 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);
```

Using ML-lex

Example 6.1

Here is an ML-lex specification for the calculator language.

```
1  type pos = int
2  type svalue = Tokens.svalue
3  type ('a, 'b) token = ('a, 'b) Tokens.token
4  type lexresult = (svalue, pos) token
5  val pos = ref 1
6  val error = fn x => TextIO.output(TextIO.stdErr, x ^ "\n")
7  val eof = fn () => Tokens.EOF(!pos, !pos)
8  fun sval([], r) = r
9    | sval(a::s, r) = sval (s, r*10+(ord(a) - ord(#"0")));
10 %%
11 %header (functor calcLexFun(structure Tokens : calc_TOKENS));
12 alpha=[A-Za-z];
13 alphanumeric=[A-Za-z0-9_];
14 digit=[0-9];
15 ws=[\ \t];
16 %%
17 \n  => (pos := (!pos) + 1; lex());
18 {ws}+  => (lex());
19 "("  => (Tokens.LParen(!pos,!pos));
```

```
20  ")"  => (Tokens.RParen(!pos,!pos));
21  "+"  => (Tokens.Plus(!pos,!pos));
22  "*"  => (Tokens.Times(!pos,!pos));
23  "/"  => (Tokens.Div(!pos,!pos));
24  "-"  => (Tokens.Minus(!pos,!pos));
25  {digit}+  => (Tokens.Int(sval(explode yytext,0),!pos,!pos));
26  {alpha}{alphanumeric}* =>
27     (let val tok = String.implode (List.map (Char.toLower)
28                 (String.explode yytext))
29      in
30        if      tok="s" then Tokens.Store(!pos,!pos)
31        else if tok="r" then Tokens.Recall(!pos,!pos)
32        else (error ("error: bad token "^yytext); lex())
33      end);
34  .  => (error ("error: bad token "^yytext); lex());
```

Using ML-lex

👉 Practice 6.1

Given the ML-lex specification in example 6.1, 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?

```
1  let val x = 6
2  in
3     x + 6
4  end
```

▸ View Solution

The Calculator Abstract Syntax Definition

Example 6.2

This is the abstract syntax definition for calculator ASTs.

```
1  structure calcAS =
2  struct
3
4  datatype
5      AST = add' of AST * AST
6          | sub' of AST * AST
7          | prod' of AST * AST
8          | div' of AST * AST
9          | negate' of AST
10         | integer' of int
11         | store' of AST
12         | recall';
13 end;
```

The Calculator Abstract Syntax Definition

👉 Practice 6.2

How would you modify the abstract syntax so expressions like the one below could be represented?

```
1  let val x = 6
2  in
3     x + 6
4  end
```

▸ View Solution

Using ML-yacc

- An ML-yacc specification consists of three parts.

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

Using ML-yacc

Example 6.3

This is the ML-yacc specification for the calculator language. The file is called `calc.grm`.

```
1  open calcAS;
2
3  %%
4  %name calc (* calc becomes a prefix in functions *)
5  %verbose
6  %eop EOF
7  %pos int
8  %nodefault
9  %pure (* no side-effects in actions *)
10 %term EOF
11     | LParen
12     | RParen
13     | Plus
14     | Minus
15     | Times
16     | Div
17     | Store
```

```
18        | Recall
19        | Int of int
20    %nonterm Prog of AST
21           | Expr of AST
22           | Term of AST
23           | StoreIt of AST
24           | NegFactor of AST
25           | Factor of AST
26
27    %%
28    Prog : Expr                              (Expr)
29
30    Expr : Expr Plus Term                    (add'(Expr,Term))
31         | Expr Minus Term                   (sub'(Expr,Term))
32         | Term                              (Term)
33
34    Term : Term Times StoreIt                (prod'(Term,StoreIt))
35         | Term Div StoreIt                  (div'(Term,StoreIt))
36         | StoreIt                           (StoreIt)
37
38    StoreIt : NegFactor Store                (store'(NegFactor))
39            | NegFactor                      (NegFactor)
40
41    NegFactor : Minus NegFactor              (negate'(NegFactor))
42              | Factor                       (Factor)
43
```

```
44  Factor : Int                          (integer'(Int))
45         | LParen Expr RParen           (Expr)
46         | Recall                       (recall')
```

Using ML-yacc

- When you see a production like this:

```
Expr : Expr Plus Term                    (add'(Expr,Term))
```

- it means when the production for addition is used an AST with `add'` at the root is returned where the left and right subtrees are the ASTs that were returned from parsing the two subexpressions.

Using ML-yacc

👉 Practice 6.3

What modifications would be required in the `calc.grm` specification to parse expressions like the one below?

```
1  let val x = 6
2  in
3    x + 6
4  end
```

▸ View Solution

Code Generation

Example 6.4

As presented in this chapter, the calculator compiler will compile some simple expressions. For instance, 5 is an expression that will compile. Compiling a program containing 5 yields the following EWE code.

```
1  SP:=100
2  R0:=5
3  writeInt(R0)
4  halt
5
6  equ MEM M[12]
7  equ SP M[13]
8  equ R0 M[0]
```

The goal when this program runs is to print 5 to the screen. This EWE program does that.

Code Generation

Example 6.5

Consider generating code for 5 + 4. If we were to blindly follow the example above the EWE code would look something like this:

```
1   SP:=100
2   R0:=5
3   R0:=4
4   R0:=R0+R0
5   writeInt(R0)
6   halt
7
8   equ MEM M[12]
9   equ SP M[13]
10  equ R0 M[0]
```

Obviously this program would print 8 as a result, not the 9 that we want. The problem is that we can't leave the 5 and the 4 in the same place. That suggests we want something like this to be generated instead.

```
1   SP:=100
2   R0:=5
3   R1:=4
4   R0:=R0+R1
5   writeInt(R0)
6   halt
7
8   equ MEM M[12]
9   equ SP M[13]
10  equ R0 M[0]
11  equ R1 M[1]
```

Code Generation

Example 6.6

Here is the register allocation framework in action. This example, taken from Appendix **??**, shows how code is generated to add two values together.

```
1   fun codegen(add'(t1,t2),outFile,bindings,offset,depth) =
2       let val _ = codegen(t1,outFile,bindings,offset,depth)
3           val _ = codegen(t2,outFile,bindings,offset,depth)
4           val reg2 = popReg()
5           val reg1 = popReg()
6       in
7         TextIO.output(outFile,reg1^":="^reg1^"+"^reg2^"\n");
8         delReg(reg2);
9         pushReg(reg1)
10      end
11
12    | codegen(integer'(i),outFile,bindings,offset,depth) =
13      let val r = getReg()
14      in
15        TextIO.output(outFile, r^":="^Int.toString(i)^"\n");
16        pushReg(r)
```

```
17          end
```

Code Generation

👉 Practice 6.4

How can code be generated to multiply two numbers together? How can code be generated to negate a value as in unary negation? Refer back to the AST definition to see what these nodes in an AST would look like. You can also refer back to the EWE language definition. Follow the pattern in example 6.6 to generate the correct code for expressions containing multiplication and unary negation.

▸ View Solution

Compiling in Standard ML

Example 6.7

This is the `Makefile` for the compiler project. It invokes the `Makefile.gen` script to start the SML compilation process. The CM does the rest. The clean rule erases all files generated during compilation of the project.

```
1  all:
2          Makefile.gen;
3
4  clean:
5          rm calccomp*
6          rm calc.lex.sml
7          rm calc.grm.sml
8          rm calc.grm.desc
9          rm calc.grm.sig
10         rm -Rf CM
11         rm -Rf .cm
```

Compiling in Standard ML

Example 6.8

`Makefile.gen` is a Unix script. The C Shell interpreter called `csh` is used to run this file. The second line of the script says to start SML and read subsequent lines as if they were typed directly into the SML shell until the `EOF` token is encountered.

```
1  #!/bin/csh
2  sml << EOF
3  CM.make "sources.cm";
4  SMLofNJ.exportFn("calccomp",calc.run);
5  EOF
```

Compiling in Standard ML

Example 6.9

The `sources.cm` file is used by the SML compiler manager to know which modules to compile. The `Group is` line is always the first line. The `$/basis.cm` and the `$/ml-yacc-lib.cm` are required to tell the compiler manager that these two built-in libraries are to be included in the compilation. The rest of the lines are the modules of the compiler.

```
1  Group is
2    $/basis.cm
3    $/ml-yacc-lib.cm
4    calc.lex
5    calc.grm
6    calc.sml
7    calcast.sml
8    registers.sml
```

Compiling in Standard ML

Example 6.10

This is the calculator compiler startup script, called `calc`. It reads the name of a text file from the user and then invokes SML by loading the binary image of the program created when the compiler was compiled by the compiler manager.

```
1  #!/bin/bash
2  set -f
3  echo -n "Enter an file name: "
4  read file
5  sml @SMLload=calccomp $file
```

Extending the Language

Example 6.11

To read a value from the keyboard while evaluating a calculator expression the lexical specification in `calc.lex` must be modified to add the `get` keyword to the list of keyword tokens. Keywords are added to the `alpha` followed by zero or more `alphanumeric` characters rule. That rule would be modified as shown here.

```
1  {alpha}{alphanumeric}* =>
2     (let val tok = String.implode (List.map (Char.toLower)
3                 (String.explode yytext))
4      in
5        if       tok="s" then Tokens.Store(!pos,!pos)
6        else if tok="r" then Tokens.Recall(!pos,!pos)
7        else if tok="get" then Tokens.Get(!pos,!pos)
8        else (error ("error: bad token "^yytext); lex())
9      end);
```

The `Get` token must be added to the terminal section of the `calc.grm` file and a production must be added to the rules in `calc.grm` to allow `get` to appear as a `Factor` in an expression.

```
Factor : Get                    (get')
```

The AST definition in `calcast.sml` must be modified so `get'` can appear as an AST node. Finally, the code generator in `calc.grm` must be modified to generate code for a `get'` node.

```
1  | codegen(get',outFile,bindings,offset,depth) =
2      let val r = getReg()
3      in
4        TextIO.output(outFile, "readInt("^r^")\n");
5        pushReg(r)
6      end
```

Let Expressions

Example 6.12

Consider the let expression given here. The scope of the identifier `x` is limited to the let expression's body.

```
1  let val x = 6
2  in
3     x + 5
4  end
```

Let Expressions

- The general format of a let expression is

```
let val id = {Expr}
in
  {Expr}
end
```

Let Expressions

Example 6.13

Currently the scanner specification in `calc.lex` checks for keywords and returns an error if a string of characters is not a keyword.

```
1  {alpha}{alphanumeric}* =>
2     (let val tok = String.implode (List.map (Char.toLower)
3                 (String.explode yytext))
4      in
5        if       tok="s" then Tokens.Store(!pos,!pos)
6        else if tok="r" then Tokens.Recall(!pos,!pos)
7        else (error ("error: bad token "^yytext); lex())
8      end);
```

The else clause needs to be modified so an identifier is returned instead of an error message. An appropriate replacement line for `calc.lex` would be

```
else Tokens.ID(yytext,!pos,!pos)
```

Let Expressions

Example 6.14

The additional AST node types required for let expressions are as follows. The `valref'` node occurs when a bound value is referred to in a program.

```
| letval' of string * AST * AST
| valref' of string
```

Let Expressions

Example 6.15

Consider the let expression given in example 6.12. The code that should be generated to evaluate that let expression is given here.

```
1   SP:=100
2   R0:=6
3   M[SP+0]:=R0
4   R1:=M[SP+0]
5   R2:=5
6   R1:=R1+R2
7   writeInt(R1)
8   halt
9
10  equ MEM M[12]
11  equ SP M[13]
12  equ R0 M[0]
13  equ R1 M[0]
14  equ R2 M[1]
```

Let Expressions

Example 6.16

Here is the code of the `boundTo` function.

```
1  exception unboundId;
2
3  datatype Type = function' of string
4                | constant' of string;
5
6  fun boundTo(name,[]) =
7      let val idname = (case name of
8                            function'(s) => s
9                          | constant'(s) => s)
10     in
11       TextIO.output(TextIO.stdOut, "Unbound identifier "^
12                     idname^" referenced or type error!\n");
13       raise unboundId
14     end
15
16   | boundTo(name,(n,ol,depth)::t) = if name=n
17                 then ol else boundTo(name,t);
```

Let Expressions

👉 Practice 6.5

What is the list of bindings in the body of the let expression presented in example 6.12?

▸ View Solution

Defining Scope in Block Structured Languages

Practice 6.6

Consider the following program.

```
1  let val x = 5
2  in
3    let val y = 10
4    in
5      let val x = 7
6      in
7        x + y
8      end
9     +x
10   end
11 end
```

Label the program by showing all bindings (both visible and invisible) that exist at all appropriate points in the program. What is the result of executing this program?

▸ View Solution

If-Then-Else Expressions

- The general format of an if-then-else expression is

```
if {Expr} {RelOp} {Expr} then {Expr} else {Expr}
```

If-Then-Else Expressions

Example 6.17

Here is a program that prints the maximum of two numbers.

```
1  let val x = get
2  in
3    let val y = get
4    in
5      if x > y then x else y
6    end
7  end
```

If-Then-Else Expressions

Example 6.18

Here is the target EWE code for the program in example 6.17.

```
1   SP:=100
2   readInt(R0)
3   M[SP+0]:=R0
4   readInt(R1)
5   M[SP+1]:=R1
6   R2:=SP
7   R2:=M[R2+0]
8   R3:=SP
9   R3:=M[R3+1]
10  if R2<=R3 then goto L0
11  R4:=SP
12  R4:=M[R4+0]
13  goto L1
14  L0:
15  R5:=SP
16  R5:=M[R5+1]
17  L1:
18  writeInt(R5)
19  halt
```

```
20
21  equ MEM M[12]
22  equ SP M[13]
23  equ R0 M[0]
24  equ R1 M[0]
25  equ R2 M[0]
26  equ R3 M[1]
27  equ R4 M[0]
28  equ R5 M[0]
```

Functions in a Block-Structured Language

Example 6.19

Consider this program that computes the factorial of n. The base case of the function accesses a value bound to the identifier called `base` which is not local to the function.

```
1  let val base = 1
2  in
3    let fun fact(n) =
4      if n = 0 then base else n * fact(n-1)
5    in
6      fact(get)
7    end
8  end
```

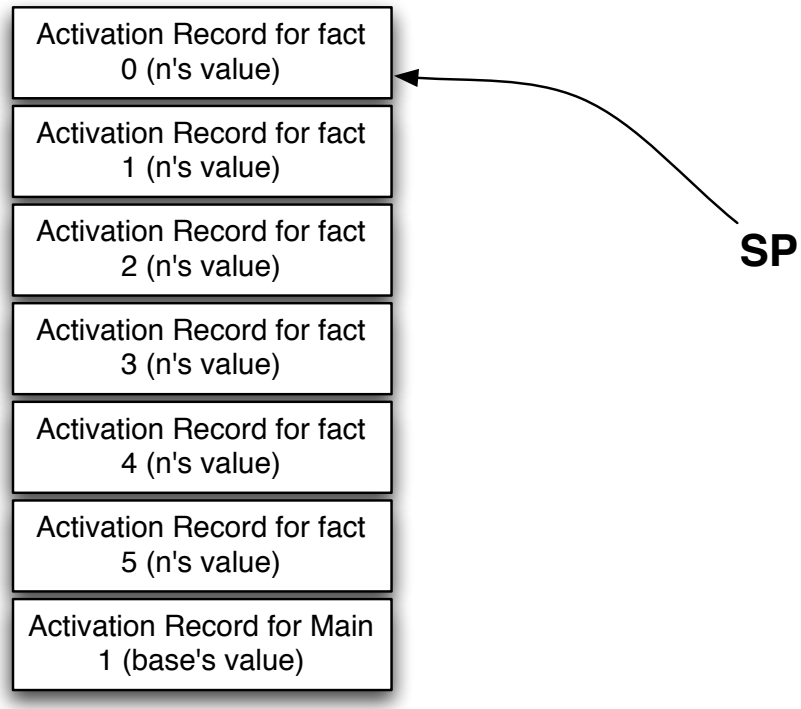
Functions in a Block-Structured Language

- Then, the general format of function definition is

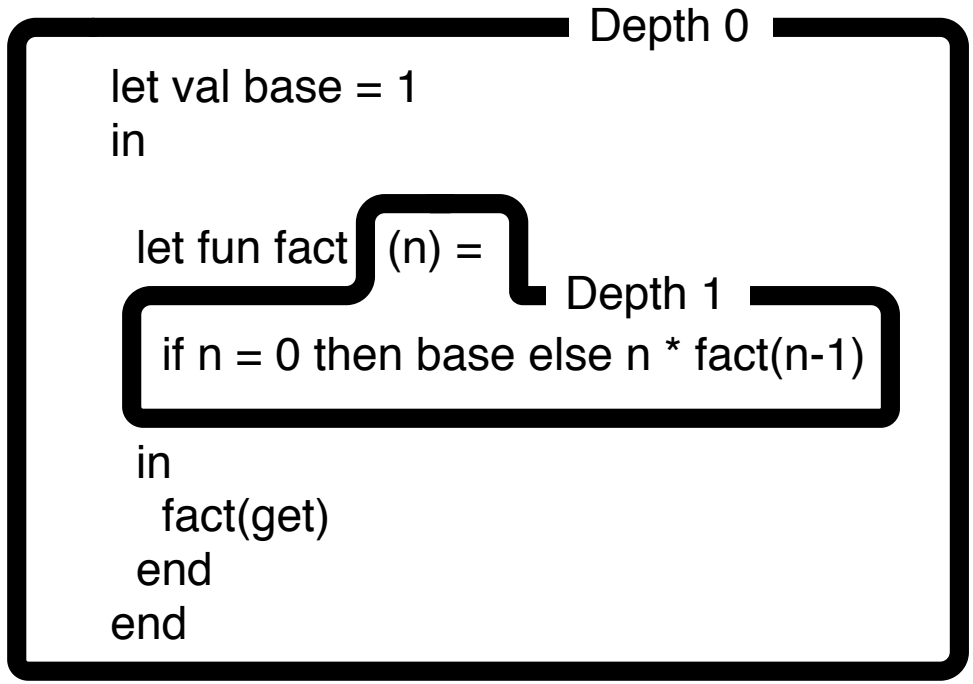
```
let fun id(id) = {Expr} in {Expr} end
```

- A function call is denoted by `id(Expr)`.
- Adding function definitions and function calls to the language greatly increases its power.

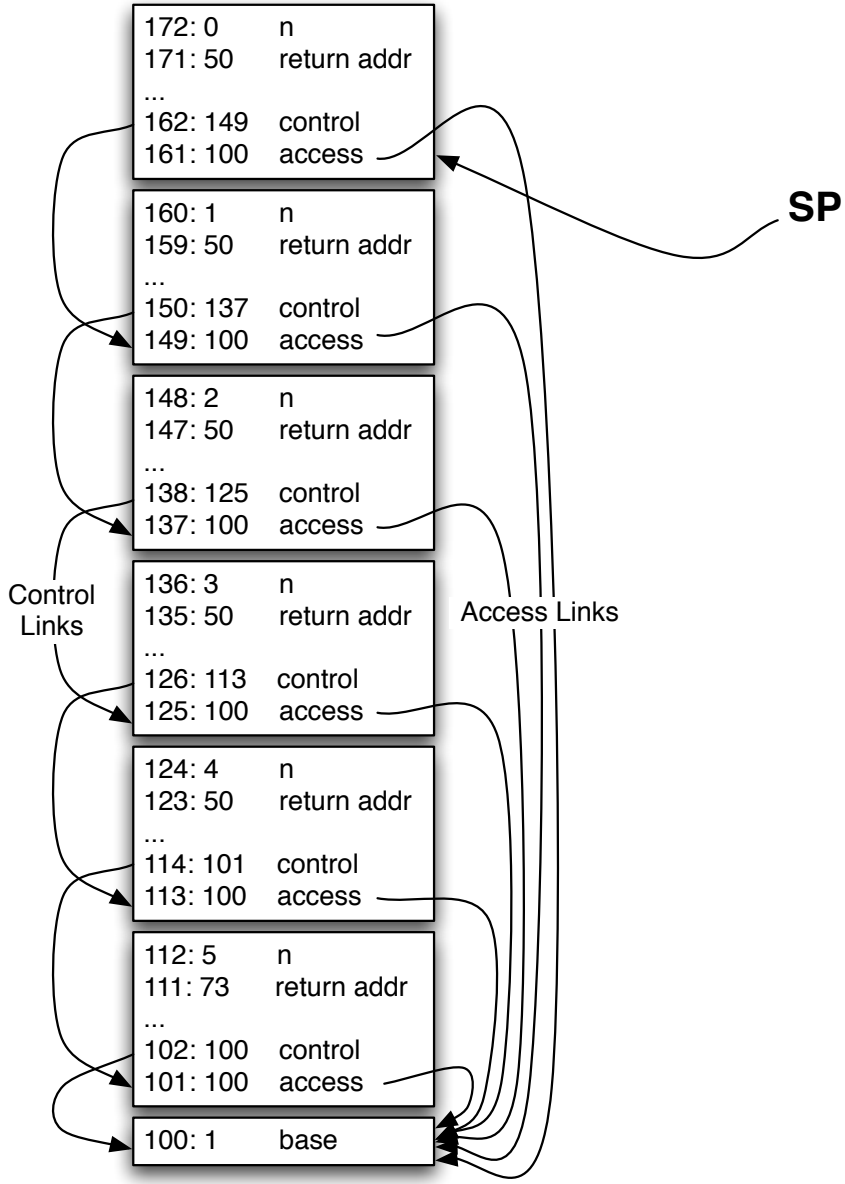
The Run-time Stack



Depth Changes in a Program



The Run-time Stack with Access and Control Links



Activation Record Format

Offset	Purpose
0	Access Link
1	Control Link
2	Saved PR0
3	Saved PR1
4	Saved PR2
5	Saved PR3
6	Saved PR4
7	Saved PR5
8	Saved PR6
9	Saved PR7
10	Saved PR8 (return address)
11	Saved PR9 (function argument)
12+	Local Variables and Constants

Call/Return Conventions

1. The access link is set by the calling code. This means the new activation record is accessed by the calling code. The next activation record starts at the next available offset after the current activation record.
2. The current SP is stored as the control link by the calling code. This value is stored in the called function's activation record. The location to store the SP register is at the next available offset plus one relative to the current activation record.
3. The argument is stored in `PR9` by the calling code during a function call.
4. The Stack pointer is incremented by the calling code to push an activation record on the stack.
5. The return address is stored in `PR8` by the calling code.
6. The function is called by the calling code.

7. The called function immediately stores the contents of the registers `PR0` to `PR9` in its activation record to preserve them across function calls.
8. After generating the code for the body of the function, the result of the body is stored in `PR9` so the calling code can find it after the function has returned.
9. The compiler must generate code to restore the registers `PR0` to `PR8` to their original values before returning from the function.
10. The contents of the stack pointer is restored by the called function to the value stored in the control link field of the activation record.
11. The function returns by restoring the program counter (PC) to the return address (located in `PR8`).
12. The calling code then places the result of the function call, located in `PR9` in the symbolic register designated to hold the result of the function call.

Setting Access Links

- When `fact` is called from the main expression the access link points to the current activation record. So the code

```
PR8:=SP                    # set the access link
M[SP+1]:=PR8               # save the access link
```

- is generated. However, when `fact` is called recursively then the following code is generated to set the access link.

```
PR8:=SP                    # set the access link
PR8:=M[PR8+0]     # follow the access link
M[SP+12]:=PR8              # save the access link
```

Calling a Function

Example 6.20

This code is generated by the compiler when a function is called.

```
1   PR8:=SP                  # set the access link
2   M[SP+1]:=PR8             # save the access link
3   M[SP+2]:=SP              # save the stack pointer
4   PR9:=R9                  # put the parameter in reg 9
5   PR8:=1                   # increment the stack pointer
6   SP:=SP+PR8
7   PR8:=PC+1                # save the return address
8   goto L1                  # make the fact function call
9   R9:=PR9                  # put the function result in the result
10                           # symbolic register
```

Non-local Access

- In the factorial example, when the value bound to `base` is referenced in the body of the function the following code is generated.

```
R3:=SP       # point to the base of the current activation record
R3:=M[R3+0] # follow one access link
R3:=M[R3+0] # load the value into a symbolic register
```

Sequential Execution

- A sequence of one or more statements can be defined as:

```
ExprSeq : Expr                          (Expr)
        | Expr Semicolon ExprSeq        (seq'(Expr,ExprSeq))
```

Exercises

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. What does ML-yacc do? What input does it require? What does it produce?
4. How can an abstract syntax tree be expressed in ML?
5. The code generator presented in this chapter does a postfix traversal of an AST. Write an AST representing the expression 5S+R and write some EWE code that represents the compiled version of this expression. Consider the examples presented in this chapter and how similar code could be used to produce your answer. Write

some justification for why you are convinced your answer is correct.

6. Given the code presented in example 6.18, consider generating code for a while loop. A while loop has the following general form

   ```
   while {Expr} RelOp {Expr} do {Expr}
   ```

   What code would be generated for this construct? Be as specific as possible. You may indicate the code generated for the various expressions as `codegen(Expr1)`, `codegen(Expr2)`, and `codegen(Expr3)`.

7. Complete the basic calculator language compiler. Finish the code generation for multiplication, division, unary negation, store, and recall. Consult `calcast.sml` and write the part of the code generation function to handle the AST nodes that do not yet have code generated for them. Write the code incrementally. Do just one operator at a time and test it.

8. Write the code given in example 6.11 to implement the `get` operator in the calculator. Be sure to test your program. When you test it, the generated code will display a question mark each time it is waiting for input.
9. Implement let expressions following the information presented in section 7 on page 29. When implementing this code be sure to use the `boundTo` function to look up the binding when required. Remember that when code is generated for an expression, the result is left on the top of the register stack. If that result is not needed, the register must still be popped and deleted using `popReg` and `delReg`.
10. Add if-then-else expressions to the calculator language as described in section 9. Make use of the `opposite` function in the code to find the opposite of a relational operator when generating the target code. Recall from that section that the final result of the if-then expression will be in the register pushed on the register stack by the *else*

clause, even if the *else* clause code was not executed at run-time. See the last paragraph in section 9 if you haven't yet read why this is the case.

11. Implement functions as described in section 10 starting on page 41. When writing code to follow access links the `forloop` function in `calc.sml` may come in handy. The function will repeatedly invoke some function a given number of time. To print "hello" 5 times you could write

    ```
    forloop(5,TextIO.output,(TextIO.stdOut,"hello\n");
    ```

    Think carefully about what the bindings and other parameters to codegen should be each time you call it.

12. Add the ability to print values to the screen by examining the code in appendix **??**. Then add sequential execution to the calculator language as described in section 15.

13. Add assignment statements to the language. To do this, add a reference type to the language. Now bindings can have either constant, function, or reference type. A variable is declared by writing

```
let val id = ref {Expr} in {Expr} end
```

The variable given by the identifier is updated by writing an expression like this

```
id := {Expr}
```

This should be declared as a new type of expression in the grammar, not a factor. The variable is dereferenced by writing the identifier preceded by an exclamation point. If `x` were declared as an integer variable then to add one to `x` you would write

```
x := !x + 1
```

14. Assuming that variable assignment in the previous exercise has been implemented it is possible to implement iteration in the language. Implement a while loop. A while loop has the form

```
while {Expr} RelOp {Expr} do {Expr}
```

Solution to Practice Problem 6.1

The keywords `let`, `val`, `in`, `end`, and the symbol = must be added as tokens. Identifiers must also be added as a token. The last section of the specification would look like this.

```
1  %%
2  \n  => (pos := (!pos) + 1; lex());
3  {ws}+  => (lex());
4  "("  => (Tokens.LParen(!pos,!pos));
5  ")"  => (Tokens.RParen(!pos,!pos));
6  "+"  => (Tokens.Plus(!pos,!pos));
7  "*"  => (Tokens.Times(!pos,!pos));
8  "/"  => (Tokens.Div(!pos,!pos));
9  "-"  => (Tokens.Minus(!pos,!pos));
10 "=" => (Tokens.Equals(!pos,!pos));
11 {digit}+  => (Tokens.Int(sval(explode yytext,0),!pos,!pos));
12 {alpha}{alphanumeric}* =>
13    (let val tok = String.implode (List.map (Char.toLower)
14                (String.explode yytext))
15     in
16       if      tok="s" then Tokens.Store(!pos,!pos)
17       else if tok="r" then Tokens.Recall(!pos,!pos)
18       else if tok = "let" then Tokens.Let(!pos,!pos)
19       else if tok = "val" then Tokens.Val(!pos,!pos)
```

```
20          else if tok = "in" then Tokens.In(!pos,!pos)
21          else if tok = "end" then Tokens.End(!pos,!pos)
22          else Tokens.ID(yytext,!pos,!pos)
23        end);
24    .   => (error ("error: bad token "^yytext); lex());
```

▸ Go Back to Practice Problem

Solution to Practice Problem 6.2

You need to add two new AST node types. One node must contain the important `let` information including the identifier (a string) and the two expressions which are AST nodes themselves. The other type of node is for referring to the bound value.

```
| letval' of string * AST * AST
| valref' of string
```

▸ Go Back to Practice Problem

Solution to Practice Problem 6.3

The grammar changes required for let expressions are as follows. The ID is needed when a bound value is referred to in an expression.

```
Factor : ...
       | ID                          (valref'(ID))
       | Let Val ID Equal Expr In Expr End
                                     (letval'(ID, Expr1,Expr2))
```

▸ Go Back to Practice Problem

Solution to Practice Problem 6.4

Code is generated in a postfix fashion in general. The code generation for multiplication and unary negation is similar to addition.

```
1  | codegen(prod'(t1,t2),outFile,bindings,offset,depth) =
2    let val _ = codegen(t1,outFile,bindings,offset,depth)
3        val _ = codegen(t2,outFile,bindings,offset,depth)
4        val reg2 = popReg()
5        val reg1 = popReg()
6    in
7      TextIO.output(outFile,reg1^":="^reg1^"*"^reg2^"\n");
8      delReg(reg2);
9      pushReg(reg1)
10   end
11
12 | codegen(negate'(t1),outFile,bindings,offset,depth) =
13   let val _ = codegen(t1,outFile,bindings,offset,depth)
14       val reg1 = popReg()
15       val reg2 = getReg()
16    in
17      TextIO.output(outFile,reg2 ^ ":= 0\n");
18         (* uses registers, rather than specifying a
19            memory location *)
```

```
20          TextIO.output(outFile,reg1^":="^reg2^" - "^reg1^"\n");
21          delReg(reg2);
22          pushReg(reg1)
23      end
```

▸ Go Back to Practice Problem

Solution to Practice Problem 6.5

The bindings in the body of the let are `[(constant'("x"),0,0)]`. There is only one binding of `"x"` to offset 0 and depth 0.

▸ Go Back to Practice Problem

Solution to Practice Problem 6.6

```
1  let val x = 5
2  in                              bindings = [(constant'("x"),0,0)]
3    let val y = 10
4    in                            bindings = [(constant'("y"),1,0),
5                                               (constant'("x"),0,0)]
6      let val x = 7
7      in                          bindings = [(constant'("x"),2,0),
8                                               (constant'("y"),1,0),
9                                               (constant'("x"),0,0)]
10       x + y
11     end
12    +x
13   end
14 end
```

Recall that bindings are a triple of identifier (with some type information), offset from the SP (i.e. Stack Pointer), and depth which was described in the section on implementing functions. The first `x` is bound to `SP+0` which supposedly holds the value 5. The `y` is bound to `SP+1` which holds 10. Finally, the second `x` is bound to `SP+2` which hold 7.

In the third version of the bindings the first `x` is not visible even though it is in the bindings. The scope of the first `x` extends through the body of the expression but it is not always visible since the innermost scope includes another `x` binding. The result of executing the program is 22.

▸ Go Back to Practice Problem