L98 Compiler

Paulo Pinto and Pablo Tavares

First release, 24 October 1999

Second release, 14 April 2013

Contents

Chapter 1

Foreword

This compiler was developed as part of a workgroup project in 1999, while both of the authors were at the university.

Although the compiler was developed in Java, one of the mainstream languages in use nowadays, it shows its Java 1.1 roots in several places.

So a decision was made to bring the build infrasture, code generation and overall code structure to Java 7, while dropping a few uses that were only relevant in Java 1.1 days.

Additionally the compiler is now GPL licensed.

Chapter 2

The L98 Language

This chapter is intended to present L98 to the reader. After reading this chapter you should be able to write L98 programs without much dificulty.

L98 has a syntax that resembles ML but it is not a funcional language. L98 allows the use of procedures and has I/O like imperative languages. So you should consider L98 an imperative language.

The language is based in the one described in the book known as the *Tiger Book* [1].

2.1 Types

As most languages, L98 has data types. Because the language is very simple, it only posses two data types. The data type `bool` represents the boolean values true and false. The data type `int` represents signed integers.

2.2 Statements

L98 has all the common statements that can be found in current languages. It has statement for performing loops, like `for` and `while`. It has two statements for performing control flow, `if...then...else` and `return`.

2.2.1 Simple Statements

In L98 as in every other language, the assignment is a fundamental operation, so it is the first statement to be presented. You can assign values to variables like this:

```
someVar    := 2;
anotherVar := 4 + someVar * 2;
aboveTen   := anotherVar > 10
```

Since we must use variables to store the values of expressions, we need a way to declare them. In L98 the statement used to declare variable is `let` as shown:

```
let
  var someVar: int := 0
  var anotherVar: int := 0
  var aboveTen: bool := false
in
 someVar    := 2;
 anotherVar := 4 + someVar * 2;
 aboveTen   := anotherVar > 10
end
```

Note that every variable must be initialized in the declaration. The variables can only be used in the `in...end` part of the `let` statement.

The L98 uses lexical scope for the variables, as demostrated by:

```
let
  var someVar: int := 5
in
 printint (someVar); /* Prints 5 */
 let
  var someVar: int := someVar + 1
 in
  printint (someVar) /* Prints 6 */
 end;
 printint (someVar); /* Prints 5 again */
end
```

The routine *printint ()* and several others are described in section 2.5 and appendix A.3.

2.3   Control Flow

2.3.1   Decisions

Like every available language, L98 must have a way to make decisions. In L98 the `if` is available for that purpose.

The `else` part of the `if` statement is optional.

As an example of the `if` statement we could make the following code:

```
let
  var someVar: int := 0
  var positive: bool := false;
in
 someVar := readint (); /* Reads an int value from the input */
 if someVar > 0 then
```

```
  positive := true;

 /* or */
 if someVar > 0 then
  positive := true
 else
  positive := false
end
```

2.3.2 Performing Loops

The loops are another form of control flow, that is why they are also in this section. In L98 there are two forms of loops. The `for` loop and the `while` loop.

The `for` loop can be used in two different ways. It could be a loop in which the control variable gets incremented, or decremented. The variables used in the `for` loop don't need to be declared. To print all the numbers from 0 to 10 we could write something like this:

```
for i := 0 upto 10 do
  printint (i)
```

If we want to print in the reverse order, we could change the loop to this form:

```
for i := 10 downto 0 do
  printint (i)
```

The `for` loop is only adequate when we know how many times to perform the loop. When the loop must be executed while some condition holds, the solution is a `while` loop. To calculate a factorial using a `while` loop, we could write something like this:

```
let
  var value: int := readint ()
  var  temp: int := 0
in
  while value > 1 do
   (
     temp := temp * value;
     value := value - 1
   );
 printint (temp)
end
```

The (...) represents a list of statements and is used like the {...} in C or `begin...end` in Pascal.

2.4 Sub-Routines

L98 supports the definition of sub-routines. They are defined by the use of the `let` statement, and can be functions or procedures.

2.4.1 Functions

As an example let's declare a function to calculate the square of its argument:

```
let
  var i : int := 0

  fun square (val x: int): int = return x * x
in
  i := square (2);      /* i contains 4 */
  i := square (3)       /* i contains 9 */
end
```

As you can see, the functions are called as in C or Pascal by using the name of the function followed by a list of arguments enclosed in (...). In L98 when a function doesn't have any arguments it must be called using an empty list. let's write a function that acts as a constant and doesn't have arguments:

```
let
  var i : int := 0

  fun const (): int = return 5
in
  i := const ()       /* i contains 5 */
end
```

Another thing that you should note, is that in the `let` statement the declarations and variables must appear before the declarations of sub-routines.

Having the possibility of executing a statement is good, but we want more. How do we declare a function with several statements? There are several solutions to this problem. One way is to use a statement list :

```
let
  var i : int := 0

  fun const (): int = (i := 2 * 3; return 5)
in
  i := const ()       /* i contains 5 */
end
```

7

Or we can use a statement that allows multiple statements, as the `let`. So we could arrange the previous code to:

```
let
  var i : int := 0

  fun const (): int = let
                      in
                        i := 2 * 3;
                        return 5
                      end
in
  i := const ()      /* i contains 5 */
end
```

2.4.2 Procedures

Using procedures is as easy as using functions, so you could declare a procedure like this:

```
let
  var i : int := 0

  proc test () = i := 5
in
  test ()       /* after the call of test (), i contains 5 */
end
```

Of course this routine isn't very helpful, so let's create one that outputs the value of an integer followed by a new line:

```
let
  proc printintln (val i: int) = (printint (i); println ())
in
  printint (2);
  printint (3);   /* Prints 23 */
  printintln (2);
  printintln (3)  /* Prints 2
                     3 */
end
```

2.4.3 Multiple Arguments

Both functions and procedures allow the use of more than one argument, for that you just have to declare them using a **;** as separator. As an example, the sample code for the min function is provided :

```
let
  fun min (val a: int; val b:int) = if a < b then
                                        return a
                                     else
                                        return b
in
  printint (min (2,3)) /* Prints 2 */
end
```

2.4.4 Variable Arguments

Sometimes we need to pass an argument that is modified by the subrotine and pre-serves its value after the call. One example of such procedure is one that exchanges the contents of its arguments, like the following one:

```
let
  var x: int := 0
  var y: int := 6
  proc swap (var a: int; var b:int) = let
                                         val temp: int = a
                                       in
                                        a := b;
                                        b := temp
                                       end
in
  printint (x); /* Prints 0 */
  printint (y); /* Prints 6 */
  swap (x, y);
  printint (x); /* Prints 6 */
  printint (y)  /* Prints 0 */
end
```

The difference is that the arguments are declared with `var`, as in Pascal.

2.4.5 Recursion

It is possible to use recursion in L98. As an example let's code the famous factorial using recursion:

```
let
  func fact (var x: int): int = if x = 0 then
                                   return 1
                                else
                                   return x * fact (x - 1)
                                        end
```

9

```
in
  printint (fact (2)) /* Prints 2! => 4 */
end
```

2.5 Builtin Routines

There are some routines that are available to all programs in L98. They are used to read and print the values of the builtin types, and to emit a new line to the output.

The available routines are:

printint Takes an integer as argument and outputs its value;

printbool Takes a bool as argument and outputs its value;

readint Reads an integer from the input and returns its value;

readbool Reads a bool from the input and returns its value;

println Outputs a new line

These routines are fully described in appendix A.3. The output and input of the programs in L98 are the standard output and standard input of the process.

Appendix A

L98 Language Definition

A.1 Keywords

The table A.1 summarizes the available keywords in L98.

and	end	in	proc	while
bool	false	int	return	val
do	for	let	then	var
downto	fun	not	true	
else	if	or	upto	

Table A.1: L98 Keywords

A.2 Grammar

The L98 grammar is described using EBNF.

For those that don't know EBNF, a brief description is given in this paragraph. The productions are described by *NameOfProduction* ::= *Production Rules*. The terminals appear between quotes, for example, "let". The non-terminals have the first letter of each work in upper case, for example, StatList. When a production like *A|B* appears, it means that A or B will be chosen. If a group of terminals and non-terminals are enclosed in [. . .], it means 0 or 1 times. If a group of terminals and non-terminals are enclosed in (. . .)*, it means 0 or more times. Finally if they are enclosed in (. . .)+, it means 1 or more times.

```
Start      ::= StatList.

StatList   ::= Statement (";" Statement)*.

Statement  ::= "let" DeclList "in" StatList "end"
             | "(" StatList ")"
```

```
              | Id (":=" Exp | "(" ElementList ")" )
              | "if" Exp "then" Statement
                ["else" Statement]
              | "while" Exp "do" Statement
              | "return" Exp
              | "for" Id ":=" Exp ("upto"|"downto")  Exp
                 "do" Statement.


DeclList  ::=   ( ( DeclVarVal )+ ( DeclFunProc )* |
                  ( DeclVarVal )* ( DeclFunProc )+ ).


DeclVarVal  ::= "val" Id ":" Type "="  Exp
              | "var" Id ":" Type ":=" Exp.


DeclFunProc ::= "fun" Id "(" ArgList ")" ":" Type
                 "=" Statement
               | "proc" Id "(" ArgList ")"
                 "=" Statement.


Type         ::= "int"
               | "bool".


Exp          ::= ArithExp [("<"|">"|"<="|
                       ">="|"="|"!=")  ArithExp].


ArithExp    ::= Term (("+"|"-"|"or") Term)*.


Term         ::= Unary (("*"|"/"|"and") Unary)*.


Unary        ::= [("-"|"not")] Element.


Element      ::= "(" Exp  ")"
               | "true"
               | "false"
               | Id  [ "(" elementList ")"]
               | Digit.


ElementList  ::= [ Exp ("," Exp)* ].


Arg          ::= "val" Id ":" Type
               | "var" Id ":" Type.


ArgList       ::= [Arg ("," Arg )* ].
```

```
Id ::= Letter (Letter | Digit)*

Letter ::= "a".."z" | "A".."Z"

Digit  ::= "0".."9"
```

A.3 Predefined Routines

The following rotines are available to the L98 programs.

A.3.1 printint

```
proc printint (val i: int)
```

Displays the value of the argument *i* to the standard output.

  Example

```
let
  a := 3
in
 printint (3);  /* Prints 3 */
 printint (a)   /* Prints 3 too */
end
```

A.3.2 printbool

```
proc printbool (val b: bool)
```

Displays the value of the argument *b* to the standard output. If the value *b* is false then the output is *false*, if the value *b* is true then the output is *true*.

  Example

```
let
  a := false
in
 printint (true);  /* Prints true */
 printint (a)      /* Prints false */
end
```

A.3.3 println

```
proc println
```

Prints a new line to the standard output.

Example

```
let
in
 println      /* Prints a new line */
end
```

A.3.4 readint

```
fun readint (): int
```

Reads an integer from the input and returns it's value.

Example

```
let
  a := readint ()
in
 printint (a)     /*  If the user typed 2, */
end                /* the output will be 2 */
```

A.3.5 readbool

```
fun readbool (): bool
```

Reads an boolean from the standard input and returns it's value.

Example

```
let
  a := readbool ()
in
 printbool (a)     /* If the user typed true, */
end                /* the output will be true */
```

Appendix B

The LVM Description

The L98 compiler generates code for an abstract stack machine. The machine was named LVM (L98 Virtual Machine).

The stack is made of 32 bit words and grows from bottom to top.

B.1   Registers

The LVM has the following 32 bit registers:

PC The Program Counter, it points to the next instruction to be executed;

SP The Stack Pointer, it points to the top of the stack;

FP The Frame Pointer, it points to the most recent activation record.

B.2   Instructions

In this section we are going to describe the instructions available in LVM. During the description we use some variables as described below.

n An integer constant;

L An address label, for example L0, L1...;

i A index inside the activation record;

l The lexical level of the variables;

M Memory address;

V An integer value

A Activation record

When the registers are used inside [...], it means the contents of the memory position pointed by the register. The → means flow of information. The symbol & is used to mean *the address of*.

B.2.1 Memory allocation

OpCode	Description
GLOBALS n	Reserves n memory slots in global memory

Table B.1: Instructions for allocating memory

ps

In the table B.1 we describe the available LVM instructions for performing memory allocation related operations.

B.2.2 Memory instructions

OpCode	Description
LOAD	n → [SP], SP + 1 → SP
LOADVAR l, i	A[l][i] → [SP], SP + 1 → SP
STOREVAR l, i	SP - 1 → SP, [SP] → A[l][i]
LOADGLOBAL i	M[i] → [SP], SP + 1 → SP
STOREGLOBAL i	SP - 1 → SP, [SP] → M[i]
LOADVARA i,l	&A[l][i] → [SP], SP + 1 → SP
LOADGLOBALA i	&M[i] → [SP], SP + 1 → SP
LOADIND	[SP - 1] → M, [M] → [SP - 1]
STOREIND	[SP - 1] → M, SP - 1 → SP, [SP - 1] → V, SP- 1 → SP, V → [M]
ALLOC n	SP + n → SP

Table B.2: Instructions for acessing memory

ps

In the table B.2 we describe the available LVM instructions for performing memory related operations. Please see the next section for information about the lexical level argument.

B.2.3 Control Flow

The instructions described in B.3 are used for control flow. Most of the instructions are already explained in the table, but we need to take a good look into CALL.

When a subrotine is called it's arguments are placed in the stack in the left to right order. This means that the first argument is the deeper one in the stack. After placing the arguments, the static link is placed in the stack, followed by the

OpCode	Description
JPC c, L	[SP - 1] → V, SP - 1 → SP, if V = c then L → PC
JMP L	L → PC
CALL l, L	
RET	
CSP n	Calls predefined rotine n. See table B.5 for details about n
HALT	Terminates execution

Table B.3: Instructions for changing the control flow

Figure B.1: Stack layout for procedure calls

dynamic link. The *static link* contains the address (FP) of the stack frame of the subrotine declared in the previous lexical level. The *dynamic link* is the previous address of the stack frame (FP). After these actions, space is allocated to the local variables. The figure B.1 shows the stack frame for a procedure call. In the figure the stack growns down the page, the bottom of the stack is the first slice.

The frame pointer points to the cell where the static link is placed. When a positive offset is issued it means a local variable, when it is negative it means an argument.

When a call is made to a routine at the same lexical level that the current one, the static link is the same that the dynamic link, so the first argument to CALL is 0. If the routine is at a lower level than the current one then the dynamic link is the same that the current one and the value of the first argument to CALL is -1. On the other hand if it is a routine at a higher lexical level than the first argument to CALL is the difference between the levels.

Figure B.2: Stack layout for function calls

It is the caller responsability to place the stack in the same stack as before the call.

In the case of function calls there isn't a explicit support in LVM, but there is a convention. When calling a function space is allocated in the stack for the result, so the called routine treats it as an argument passed by reference. This is displayed in the figure B.2.

Lexical level	Description
n > 0	The called function is n levels above the current one
0	The called function is at the same level that the current one
-1	The called function is at a lower level

Table B.4: Lexical levels for computing the CALL instruction

B.2.4   Arithmetic Operators

The table B.6 describes the available arithemetic operators in LVM. All the operators perform 32 bit integer arithmetic. Checking for bad arguments, like *division by zero* is not required, but advisable. The numbers can be negative, although there isn't a standard representation, we think that two's complement is a good choice.

Argument	Routine
0	printint
1	readint
2	printbool
3	readbool
4	println

Table B.5: Codes for standard routines

OpCode	Description
ADD	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 + V2 → [SP - 1]
SUB	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 - V2 → [SP - 1]
MUL	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 * V2 → [SP - 1]
DIV	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 / V2 → [SP - 1]
NEG	[SP - 1] → V1, - V1 → [SP - 1]
NOT	[SP - 1] → V1, not V1 → [SP - 1]
AND	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 and V2 → [SP - 1]
OR	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 or V2 → [SP - 1]
EQ	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 = V2 → [SP - 1]
NEQ	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 != V2 → [SP - 1]
LT	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 < V2 → [SP - 1]
GT	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 > V2 → [SP - 1]
GEQ	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 >= V2 → [SP - 1]
LEQ	[SP - 1] → V1, [SP - 2] → V2, SP - 1 → SP, V1 <= V2 → [SP - 1]

Table B.6: Instructions for performing math operations

Appendix C

Code generation

The compiler currently offers two backendss

The original backend that was part of the first release, which generates L98 bytecodes, and a new one that generates 32 bit x86 code.

C.1   Generating bytecodes

By default, the compiler will produce a text file with the *.s* extension, containing a textual version of the LVM bytecode.

C.1.1   Bytecode file format

The format is the same used by most assemblers :

```
[Label:] instruction [; comment]
```

The entry point of the program is specified by the label P_START, and the `GLOBAL` instruction needs to be the first in the sequence of instructions.

C.2   Generating native code

To generate an executable, instead of plain bytecode, the compiler needs to be invoked with the *-e* switch.

In the first release the compilation to native code was performed by making use of NASM macros that translated the bytecodes text file into x86 Assembly code.

In this release, the backend was refactored and now it takes the responsability to generate proper Assembly code as well.

Currently an Assembly file is generated in the AT&T format ending with a *.s* extension, similar to the bytecode case.

The runtime is still written in C and is stored as a resource inside the *jar* file.

During compilation to native code, a temporary file with the runtime code is generated and GCC is invoked with it and the generated Assembly file together.

C.3   The builtin routines

The builtin routines are defined in C. This makes the compiler dependent on a C compiler in addition to an Assembler and Linker.

On the other hand, it allows for more portability across systems, due to the ways it is possible to invoke kernel APIs across operating systems, regardless of the process architecture.

Currently there is the plan to eventually improve the L98 language to be able to do syscalls, thus removing the need of a C compiler in the process.

Appendix D

Installing and using the compiler

D.1 Installing

To install the compiler you need to have installed the following software:

- a JDK compatible with Java version 7 or higher;

- Maven version 3 or higher;

- GNU binutils and GCC (required for native code generation).

To compile the compiler, execute the following steps:

1. *cd* into the L98 directory;

2. Make sure Maven and Java are defined in the *PATH*

3. invoke *mvn*

Assuming everything went without errors you can start using the L98 compiler.

D.2 Compiling

To compile your programs, you just have to use the *l98c* shell script. The *l98c* command is a sh shell script. It invokes the java interpreter with the name of the class that implements the compiler and the program arguments.

The command line sintax for invoking the *l98c* script is :

```
l98c aSourceFile.l98
```

This way an *aSourceFile.s* file is created with the LVM code. If the *-e* switch additionally given, the contains x86 code.

```
l98c -e aSourceFile.l98
```

An executable file named *aSourceFile* is generated.

If you want the intermediate Assembly file to be left on the system, you should use the *-s* switch as well.

Appendix E

License

This release is under the GNU GPL v2.

Bibliography

[1] Addrew W. Appel. *Modern Compiler Implementation in Java.* 1998 Cambridge University Press. ISBN 0521583888