CSA2201 – Compiling Techniques
Course Assignment 2011 – 2012

Department of Computer Science. University of MALTA.

Sandro Spina / Gordon Mangion

This is the description for the assignment of unit CSA2201, Compiling Techniques. This assignment is worth 15% of the total mark for this unit. The assignment has to be carried out on an individual basis. Under no circumstances should code be shared among students. Please remember that plagiarism will not be tolerated; the final submission must be entirely your work.

Deliverables

You will submit your project source code, executable files and a DOC/DOCX/PDF file containing the project documentation/report on optical medium. Note that assignment binaries must run straight from CD/DVD-ROM without the need for any installation unless specified in the accompanying document, in which case a detailed installation guide must also be provided.

Description

In this assignment you are to develop a compiler which will translate source files of a simple language called SL (Simple Language) into S-Machine (a simple stack-based virtual machine) programs. The S-Machine and its instruction-set are described in a later section. Below is the definition in Extended Backus-Naur Form (EBNF) of the SL language. The starting point of the grammar is the “*program*” non-terminal at the bottom of the definition.

`Letter`	`::=`	`[“a“-“z“ “A“-“z“]`
`Digit`	`::=`	`[“0“-“9“]`
`Type`	`::=`	`“int“`
`IntegerLiteral`	`::=`	`[“0“-“9“] { [“0“-“9“] }`
`Identifier`	`::=`	`( “_” | Letter ) { “_” | Letter | Digit }`
`RelationOp`	`::=`	`"<" | ">" | "==" | "!=" | "<=" | ">="`
`AdditiveOp`	`::=`	`"+" | "-"`
`MultiplicativeOp`	`::=`	`"*" | "/"`
`AssignmentOp`	`::=`	`"="`

`Factor`	`::=`	`"(" Expression ")"`
		`| Literal`
		`| Identifier`
		`| FunctionCall`

```
ActualParameters        ::=   Expression { “,” Expression }

FunctionCall            ::=   Identifier "(" [ActualParameters] ")"

Term                    ::=   Factor [ MultiplicativeOp Factor ]

SimpleExpression        ::=   Term { AdditiveOp Term }

Expression              ::=   SimpleExpression [ RelationOp
                              SimpleExpression ]

DeclarationStatement    ::=   "var" Type Identifier [ "="
                              IntegerLiteral ] ";"

AssignmentStatement     ::=   Identifier AssignmentOp Expression ";"

IfStatement             ::=   "if" "(" Expression ")" StatementBlock [
                              "else" StatementBlock ]

WhileStatement          ::=   "while""("Expression")" StatementBlock";"

ReadStatement           ::=   "read" Identifier ";"

WriteStatement          ::=   "write" (IntegerLiteral | Identifier) ";"

ReturnStatement         ::=   "return" [Expression] ";"

HaltStatement           ::=   "halt" ";"

FormalParameter         ::=   Type Identifier

FormalParameters        ::=   FormalParameter { “,” FormalParameter }

FunctionDeclaration     ::=   Type Identifier “(“ [FormalParameters]
                              “)” StatementBlock

Statement                     AssignmentStatement
                              | DeclarationStatement
                              | IfStatement
                              | WhileStatement
                              | ReadStatement
                              | WriteStatement
                              | ReturnStatement
                              | HaltStatement
                              | StatementBlock

StatementBlock          ::=   "{" { Statement } "}"

Program                 ::=   { FunctionDeclaration } { Statement }
```

Task Breakdown

The assignment is broken down into four tasks. Below is a description of each task accompanied with the assigned mark.

Task 1 - Create Javacc grammar file

In this first task you are to create the Javacc grammar file for the SL definition given above. You are free to modify the production rules as long as the changes are documented in the report and that the source language (SL) remains unaltered. Should you prefer to use an alternative to JJTree pre-processor in order to build the parse tree please go ahead.

[Marks: 20%]

Task 2 - Parse Tree Generation

You should enhance the parser developed in Task 1 to output a textual (or graphical if you prefer) representation of the generated parse tree.

[Marks: 5%]

Task 3 - Semantic Analysis and Code Generation

In this task you are to use the visitor design pattern (or any method you deem suitable) to traverse the parse tree to perform type checking and code generation.

[Marks: 30%]

Task 4 Sample programs

Together with the above, you are to design and implement <u>short</u> sample source programs to test the outcome of your compiler. In your report, state what you are testing for, insert the programs' parse tree, the resulting code generated and the outcome of your test.

[Marks: 20%]

The Report

In addition to the source and class files, you are to deliver a report. In your report include any deviations from the original EBNF, the salient points on how you developed the compiler (and reasons behind any decisions you took) including semantic rules and code generation, and any sample SL programs you developed for testing.

[Marks: 25%]

S-Machine

The S-Machine (SM) is a simple Stack-based virtual machine. The main components of this virtual machine are its CodeArea (where the programs are loaded), the Stack (where the data resides and is manipulated) and the VCPU (Virtual CPU).

SM's VCPU has just 3 registers (see list below) because its operations rely heavily on the machine's stack.

- PC – Program Counter; Points to the next instruction in memory
- SP – Stack Pointer; Points to the top of Stack
- FP – Frame Pointer; Points to the current Stack Frame

Each S-Machine instruction takes exactly four bytes (32 bits). This makes it easier to generate code for the instruction set since each instruction is aligned at a 32 bit boundary. This means that you can calculate exactly the offset of the n^{th} instruction (= n*4 bytes).

The instruction-set of S-Machine is illustrated in the table below:

Opcode (Hex)	**Mnemonic**	**Description**	**Semantics**
00 00 00 00	NOP	No operation	PC = PC + 1;
01 00 nn nn	LDC n	Load integer constant n onto top of stack	SP = SP + 1; Stack(SP) = n; PC = PC + 1;
02 ll nn nn	LD l, n	Load value onto top of stack from variable level l, offset n	SP = SP + 1; Stack(SP)= Stack(F(FP,l)+n); PC = PC + 1;
03 ll nn nn	STORE l, n	Store value from top of stack into variable level l, offset n	Stack(F(FP,l)+n)=Stack(SP); SP = SP - 1; PC = PC + 1;
04 00 00 00	DUP	Duplicate top of stack item	SP = SP + 1; Stack(SP) = Stack(SP-1); PC = PC + 1;
05 00 00 00	POP	Pop item from stack	SP = SP - 1; PC = PC + 1;
06 00 nn nn	JMP n	Jump to location n	PC = n;

07 00 nn nn	JZ n	Jump if zero	If(Stack(SP) == 0)
			  PC = n;
			else
			  PC = PC + 1;
			SP = SP – 1;

08 00 nn nn	JNZ n	Jump if not-zero	If(Stack(SP) != 0)
			  PC = n;
			else
			  PC = PC + 1;
			SP = SP – 1;

09 00 00 00	HALT	Stop program execution	
0A 00 nn nn	ENTER n	Enter a stack frame	SP = SP + 1; Stack(SP) = FP; FP = SP; SP = SP + n; PC = PC + 1;
0B 00 00 00	LEAVE	Leave stack frame	SP = SP – FP; FP = Stack(SP); PC = PC + 1;

0C 00 nn nn	CALL n	Call function at location n	Stack(SP) = PC + 1; PC = n;
0D 00 00 00	RET	Return from Call	PC = Stack(SP); SP = SP – 1;
0E 00 nn nn	RETN n	Pop n items from stack and return from Call	PC = Stack(SP); SP = SP – (n+1);
0F 00 00 00	ADD	Addition	Stack(SP-1) = Stack(SP-1) + Stack(SP); SP = SP - 1; PC = PC + 1;
10 00 00 00	SUB	Subtraction	Stack(SP-1) = Stack(SP-1) - Stack(SP); SP = SP - 1; PC = PC + 1;
11 00 00 00	MUL	Multiplication	Stack(SP-1) = Stack(SP-1) * Stack(SP); SP = SP - 1; PC = PC + 1;
12 00 00 00	DIV	Division	Stack(SP-1) = Stack(SP-1) / Stack(SP); SP = SP - 1; PC = PC + 1;
13 00 00 00	MOD	Modulus	Stack(SP-1) = Stack(SP-1) % Stack(SP); SP = SP - 1; PC = PC + 1;
14 00 00 00	EQ	Equal	Stack(SP-1) = Stack(SP-1) == Stack(SP); SP = SP - 1; PC = PC + 1;
15 00 00 00	NE	Not Equal	Stack(SP-1) = Stack(SP-1) != Stack(SP); SP = SP - 1; PC = PC + 1;
16 00 00 00	LT	Less Than	Stack(SP-1) = Stack(SP-1) < Stack(SP); SP = SP - 1; PC = PC + 1;
17 00 00 00	GT	Greater Than	Stack(SP-1) = Stack(SP-1) > Stack(SP); SP = SP - 1; PC = PC + 1;
18 00 00 00	LE	Less Than or Equal To	Stack(SP-1) = Stack(SP-1) <= Stack(SP); SP = SP - 1; PC = PC + 1;
19 00 00 00	GE	Greater Than or Equal To	Stack(SP-1) = Stack(SP-1) >= Stack(SP); SP = SP - 1; PC = PC + 1;
1A ll nn nn	READ l, n	Read an integer input from console	Stack(F(FP,l)+n) = Input; PC = PC + 1;
1B 00 00 00	WRITE	Write an integer to console output	Output = Stack(SP); SP = SP - 1; PC = PC + 1;

Where: F(FP, l) = if (l == 0) then FP else F(Stack(FP), l-1)

For further details on S-Machine please refer to the S-Machine tutorial/lecture slides.

Final Notes

As an example, the SL source program below:

```
// Factorial Function
int fact(int n)
{
    if(n==1)
    {
        return 1;
    }
    else
    {
        return n * fact(n-1);
    }
}

var int n = 4;
n = fact(5);
write n;
halt;
```

Should generate an S-Machine program along the lines of the code below:

```
00: ENTER 0, 1      // Prepare 1 data variable
01: JMP   0, 22     // Jump to main code
02: ENTER 0, 0      // Start of fact function, prepare 0 local variables
03: LD    0, -2     // Load value of n param on stack
04: LDC   0, 1      // Load/Push constant 1 on stack
05: EQ    0, 0      // n == 1
06: JZ    0, 12     // Jump if false(0) to instruction at location 12
07: LDC   0, 1      // Load/Push constant 1 on stack
08: STORE 0, -3     // Store it in the return value location
09: LEAVE 0, 0      // Leave the scope
10: RETN  0, 1      // Return and pop 1 argument
11: JMP   0, 22     // Jump to instruction at location 22
12: LD    0, -2     // Load value of param n (at scope 1)
13: LDC   0, 0      // Load/Push constant 0 on stack
14: LD    0, -2     // Load value of param n (at scope 1)
15: LDC   0, 1      // Load constant 1 on stack
16: SUB   0, 0      // Subtract top two stack items and push the result
17: CALL  0, 2      // Call subroutine at location 2
18: MUL   0, 0      // Multiply top two stack items and push the result
19: STORE 0, -3     // Store result in the return value location
20: LEAVE 0, 0      // Leave the scope
21: RETN  0, 1      // Return and pop 1 argument
22: LDC   0, 4      // Load/Push constant 4 on stack
23: STORE 0, 1      // Store it in the variable n (scope 0) location
24: LDC   0, 0      // Load/Push constant 0 on stack
25: LDC   0, 5      // Load/Push constant 5 on stack
26: CALL  0, 2      // Call subroutine at location 2
27: STORE 0, 1      // Store it in the variable n (scope 0) location
28: LD    0, 1      // Load value of variable n (at scope 0)
29: WRITE 0, 0      // Output value on top of stack (result)
30: HALT  0, 0      // Halt machine execution
31: LEAVE 0, 0      // Leave the scope
```

The above is a textual representation of the code generated. The actual file written to disk should look like the following (the numbers below show the file dumb in hexadecimal base, this means that the file is actually a sequence of bytes 0a, 00, 00, 01, 06, 00, …);

```
0a 00 00 01
06 00 00 16
0a 00 00 00
02 00 ff fe
01 00 00 01
14 00 00 00
07 00 00 0c
01 00 00 01
03 00 ff fd
0b 00 00 00
0e 00 00 01
06 00 00 16
02 00 ff fe
01 00 00 00
02 00 ff fe
01 00 00 01
10 00 00 00
0c 00 00 02
11 00 00 00
03 00 ff fd
0b 00 00 00
0e 00 00 01
01 00 00 04
03 00 00 01
01 00 00 00
01 00 00 05
0c 00 00 02
03 00 00 01
02 00 00 01
1b 00 00 00
09 00 00 00
0b 00 00 00
```

The numbers above are formatted as 4 bytes per line for clarity only and for a one-to-one correspondence with the generated code dump above. The bytes are in binary format and consecutive.

The initial state of the S-Machine is as follows:

```
PC = 0x0000
SP = 0xffff (-1)
FP = 0x0000
```

The CodeArea and Stack are both empty.

Hints

In general, “if” statements have the following structure when compiled:

SL source	Compiled Code

`if( Condition )`
`{`
`    Then_Block`
`}`
	`    Code for Condition`
	`    JZ` **`Out_of_Then`**
	`    Code for Then_Block`
	`Out_of_Then:`

`if( Condition )`
`{`
`    Then_Block`
`}`
`else`
`{`
`    Else_Block`
`}`
	`    Code for Condition`
	`    JZ` **`Location_of_Else`**
	`    Code for Then_Block`
	`    JMP` **`Out_Of_If`**
	`Location_of_Else:`
	`    Code for Else_Block`
	`Out_Of_If:`

Code generated for “while” blocks should look like the snippet below:

SL source	Compiled Code

`while( Condition )`
`{`
`    While_Block`
`}`
	`Location_of_Condition:`
	`    Code for Condition`
	`    JZ` **`Out_Of_While`**
	`    Code for While_Block`
	`    JMP` **`Location_of_Condition`**
	`Out_Of_While:`

In the code snippets above, the entities in bold are labels and represent memory locations.