AOR Priv.-Doz. Dr. Thomas Noll

Friedrich Gretz, Christina Jansen

Exercise 1 (Procedure Stack of the AM): (3 Points)

Which of the following procedure stacks could be result of the execution of an EPL-programm? Why?

a) $p_1 = 13:3:9:1:4:3:2:2:4:5:5:15:1:3:2:12:0:0:0:17:3$

b) $p_2 = 13:3:9:1:4:3:2:2:5:4:5:15:1:3:2:12:0:0:0:17:3$

c) $p_3 = 13:3:9:1:4:3:2:2:8:4:5:15:1:3:2:12:0:0:0:17:3$

Exercise 2 (Code Generation): (4+4+7+2 Points)

Note, that you do not need a working parser to solve this exercise!

We now want to complete our compiler implementation with the generation of Jasmin[1] code.
Assume the following grammar for the *WHILE* language. The terminal alphabet is the set of tokens (coming from the lexer). The production rules are given below:

start	→	prog (1)
prog	→	stmt prog | stmt (2, 3)
stmt	→	decl SEM | assign SEM | branch (4, 5, 6)
decl	→	INT ID (7)
assign	→	ID ASSIGN expr (8)
branch	→	IF LBRAC guard RBRAC LCBRAC prog RCBRAC (9)
expr	→	NUM | ID | LBRAC expr PLUS expr RBRAC (10, 11, 12)
guard	→	expr LT expr | expr EQ expr | expr NEQ expr | expr GT expr (13, 14, 15, 16)

Our goal is to write a function `translateWHILE`, that takes a left-most derivation as input and outputs a corresponding Jasmin program.

You may assume that the input *WHILE*-program satisfies the following conditions:

- every variable declared is accessible, independent of its "usual" scope
- every variable is declared before it is used
- no variable is declared more than once

The (simplified) syntax of Jasmin is the following:

- iload <varnum> - pushes the int value held in a variable represented by *varnum* onto the operand stack.
- istore <varnum> - pops an int off the stack and stores it in the variable represented by *varnum*.
- sipush <int> - takes a single integer and pushes it onto the operand stack.
- iadd - pops two integers from the operand stack, adds them, and pushes the integer result back onto the stack.
- if_icmpeq <Label> - if_icmpeq pops the top two ints off the stack and compares them. If the two integers are equal, execution branches to the address (pc + offset), where pc is the address of the if_icmpeq opcode in the bytecode and offset is computed for you from the address of <Label>
- if_icmplt <Label> - analogous to if_icmpeq

[1] http://jasmin.sourceforge.net/

- if_icmpgt <Label> - analogous to if_icmpeq
- goto <Label> - Causes execution to branch to the instruction at the address (pc + offset), where pc is the address of the goto opcode in the bytecode and offset is computed for you using the address associated with <Label>.

where <Label> is a label name. To define the location of the label, use the <Label> name followed by a colon, e.g. <Label>:.

An example Jasmin file is provided on the course webpage. You can use it as template for your generated code. To compile it to JVM bytecode use:

```
java -jar jasmin.jar Count.j
```

You can then execute it on your JVM as usual with:
`java Count`

a) Briefly describe which adaptations/simplifications you carry out to the semantics of AM code (lecture) in order to meet Jasmin as the target abstract machine code.

Provide semantics for the instructions iadd, iload, istore.

b) Define the translation functions for decl, assign and branch as well as for the complete program.

c) Implement a function `translateWHILE(List<Integer> derivation, List<Symbol> lexOutput)`, which takes a left-most derivation *derivation* and the output of a lexical analysis *lexOutput* as input and outputs a Jasmin program according to your specified translation.

d) Provide the output of `translateWHILE` on the results of the lexical analysis of a *WHILE*-program:
(INT,) (ID, x) (SEM,) (ID, x) (ASSIGN,) (NUM, 0) (SEM,) (IF,) (LBRAC,) (ID, x) (EQ,) (NUM, 0) (RBRAC,) (LCBRAC,) (ID, x) (ASSIGN,) (LBRAC,) (ID, x) (PLUS,) (NUM, 1) (RBRAC,) (SEM,) (RCBRAC,),
where $derivation = 1, 2, 4, 7, 2, 5, 8, 10, 3, 6, 9, 14, 11, 10, 3, 5, 8, 12, 11, 10$.