#### **P4 target Code and Storage**

### Invocation

./comp  [file]

where file is as before with implicit the extension. **Wrong invocations will not be graded**.

#### **Details**

The program is to parse the input, generate a parse tree, perform static semantics, and then generate a target file. Any error should display detailed message, including line number if available (depending on scanner). Since P1-P3 were already tested, this focuses on testing the target code and and storage allocation. Testing will be performed by interpreting the generated targets to assess if the runtime behavior is correct.

The produced target name should be

**a.asm** if keyboard input  
    **file.asm** if ***file***was the argument without the implicit extension.

The program has parts as discussed. Code generation is done during a parse tree traversal - additional traversal after any prior traversals. Storage allocation is done using some P3 information as well as some traversal information after the last traversal.

Storage Global option

To allocate storage for any variable, you need to use the storage directive in the target language. Note that the target language does allow variable names starting with "\_" even though the documentation says start with a letter - this was done so that temporary variables, that is variables created by the compiler, can start with \_ not to clash with user variables in the input program.

Traversal

The parse tree is equivalent to the program in left-to-right traversal. The program semantics for both the input and the target language state the execution should be from the first instruction down. Therefore, the target needs to be generated performing left to right traversal of the tree so the target will also have top to down semantics (in some nodes the left to right routine may not be followed and some nodes may be preorder, some inorder, some postorder depending on the semantics of the node).

* This traversal should follow static semantics traversal(s)

Grading

We will not be testing for any P1, P2, P3 errors at this time and will only use input programs without such errors. As before, we will also test with spaces between tokens and without the comments. Also note test programs will be constructed to test each of the following separately as much as possible but some must be combined for example all tests are mostly impossible without working output.

Total 100, plus 20 extra as noted

1. Style, coding and architectural 10%

2. Proper output files using both keyboard and file input 5%

3. Performance, using only file inputs, 85%

1. Single and nested conditional 5%+10%=15%
2. Single and nested loop 5%+10%=15%
3. Expressions in assignments, prints, and conditions 15%
4. Input/output 10%
5. Assignment 10%
6. Conditions 10%
7. All at once 10% (large complex program)

4. Extra 20 points for anyone using the stack to handle temporary variables. You must note this on the first line of your README file.

#### **To test your P4 (entire project)**

#### 1. Compile your P4 as comp

#### 2**.** Write one of the test file on Delmar as a source e.g. test1.sp23

#### 3. Run the test file with your compiler project comp test1

#### 4. The above, if no errors in the source and your compiler, should produce test1.asm

#### a. If there are errors, it must be your compiler. If you suspect a faulty test file, let me know

#### 5. Interpret the output to see if it does what it should.

./VirtMach test1.asm

a. If it doesnt, you should open test1.asm to see what is wrong in the target and then fix your P4 accordingly

b. You may also run it in the URL interpreter line by line to see where the problem is

**P4 Runtime Semantics**

Statements and Operations

* Basic semantics as in C - program executes sequentially from the beginning to the end, one statement at a time
  + means that the target needs to be generated from the tree left to right
* Conditional <if> statement is like the else-less if statement in C
* Loop <loop> statement is like the while loop in C
* Assignment evaluates the expression on the right and assigns to the ID on the left
* Arithmetical binary operations are standard except that / is integer division and unary - is negation
* All expressions are evaluated before being used
* Relational operators in <RO>
  + <= is less equal
  + >= is greater equal
  + < and > are less and greater respectively
  + = is equality
  + ~ is NOT equal
* IO reads/prints a 2-byte signed integer
* All data is 2-byte signed integer

Data

* All data is 2-byte signed integers
* Assume no overflow in operations

Storage

* Global option will assume all input variables are global, allocated in the data pool of the target language using the storage directives
* Temporary variable, if any, can be on the stack but the simpler option is to allocate them in the same global pool

Target Language

* VM ACCumulator assembly language as given

P4 Code Generation Support

Code generation

The input program is now represented as a parse tree. In fact, every statement of the input is represented as a subtree, with every BNF nonterminal represented as a node. We now know the runtime (dynamic) semantics of the input language, and the objectives is to express the same semantics  in the target language knowing the target semantics. We thus generate the target one node at a time, mapping the input language semantics to the target language semantics. Separately we have to may the storage semantics (variables and persistence).

* The parse tree is equivalent to the program in left-to-right traversal. The  execution should be from the first instruction down. Therefore, the target needs to be generated performing left to right traversal of the tree so the target will also have top to down semantics
* All nodes the parse tree are either
  + code generating
  + not code generating - most without token(s) but not all
* When visiting a node during the code generation traversal that is not code generating
  + if no children, return and continue the traversal
  + if children, continue traversal, most likely left to right, then return
* When visiting a code generating node during the code generation traversal
  + **the code is written to the target**
  + **each node with the same label always generates the same code**
    - regardless of location in the tree
    - may be one of multiple cases, one per production in BNF
    - the only difference may come from the token(s) found in the node (*e.g.*, different operator)
  + some actions can be preorder, some inorder, some postorder, depending the node (determined by the label) and its semantics
  + if a value is to be produced, always leave the result in the ACCumulator
  + if a value was produced by another node you traversed to, assume the result is in the ACCumulator
  + if you placed a value in the ACCumulator but still need to continue traversal in the subtrees, the ACCululator may need to be saved into a temporary storage with a unique name if using persistent storage for temporaries
    - use temporary variables named not to conflict with input variable names such as starting with '\_' or starting with 't' which we assume not to be used in the input
* At the end of the traversal, print STOP to the target
  + to be followed by listing of global variables and temporaries, seein storage allocation
    - temporaries can be global, or on the stack
* Variables in the input program will require
  + variable creation upon definition - see storage allocation
  + variable access upon use  - global option or a temporary variable that is global just uses the variable name

Below is code fragment that can be reused - how to create new names for temporary variables (start with T) and labels (start with L). The partial traversal does different things based on the node label. If needed, this generates new temporary variable or new label. The traversal code is from a similar project with differing BNF so do not use directly.

Keep in mind the generation of a variable or a label, and its actual use, if separated by a call to another node (may generate new variables or labels) - will require the variable and label to be locally saved, see examples discussions

static LabelCntr=0; /\* counting unique labels generated \*/  
static VarCntr=0; /\* counting unique temporaries generated \*/  
typedef enum {VAR, LABEL} nameType;  
static char Name[20]; /\* for creation of unique names \*/  
  
static char \*newName(nameType what)  
{ if (what==VAR) // creating new temporary  
 sprintf(Name,"T%d",VarCntr++); /\* generate a new label as T0, T1, etc \*/  
 else // creating new Label  
 sprintf(Name,"L%d",LabelCntr++); /\* new lables as L0, L1, etc \*/  
 return(Name);  
}  
  
static void recGen(nodeType \*nodeP,FILE \*out) // recursive traversal   
{ char label[20], label2[20], argR[20]; // local storage for temporary or label  
 if (nodeP==NULL)  
 return; // no nodes  
 switch (nodeP->nodeId) // perform different actions based on the node label  
 { case readNode: fprintf(out,"\tREAD\t%s\n",nodeP->str);  
 break;  
 case assignNode: recGen(stats->child1,out); /\* evaluate rhs \*/  
 fprintf(out,"\tSTORE\t%s\n",nodeP->tokenP.str);  
 break;  
 case ifNode: recGen(nodeP->child3,out); /\* exprRight \*/  
 argR = newName(VAR);  
 recGen(nodeP->child1,out); /\* exprLeft \*/  
 fprintf(out,"%SUB %s\n",argR); /\* ACC <- exprLeft - exprRight \*/  
 label = newName(LABEL);  
 if (codeP->child2->token == ==Tk) { /\* False is ACC != 0 \*/  
 fprintf(out,"BRNEG %s\n",label);  
 fprintf(out,"BRPOS %s\n",label);  
 }  
 recGen(nodeP->child4,out); /\* dependent statements \*/  
 fprintf(out,"%s:\tNOOP\n",label);  
 break;  
// etc.

#### **Example code per node**

Since the objective is generate the target mapping the input semantics to the output semantics, and the input is now represented as a tree with each node representing a nonterminal, the objective is to map each type of node (based on label, which traces to nonterminal, which traces to a part of the input language) to the target. For example, since the semantics of the output in the source language is to print the value of the expression (after evaluating the expression), when visiting the output node in the tree we need to use the target instructions to evaluate the expression and then output the result and thus see the example

# **P4 Test Files - Code Generation**

Compile the input source into the asm and then execute through the virtual machine to assess runtime semantics as noted. The examples are the same for local and global option.

There should be no syntax nor semantics errors so let me know if you see something suspicious.

#p4\_1 - testing output, print 1

main   
 print 1 .  
end

$p4\_2 - testing also input, echo input

main   
 print 1 .  
 scan aa .  
 start  
 let aa = 2 .  
 print aa .  
 stop  
end

$p4\_3 - testing expression, print 1

main   
 print 3 \* 7 / 8 + 5 - - - 4 .   
end

#p4\_4 - testing cond, print 1 if input >0

let aa = 1 .  
main   
 scan aa .  
 cond ( aa > 0 )   
 print 1 .  
end

#p4\_5 - nested cond, print 1 if input is teen age

let aa = 1 .  
main   
 scan aa .  
 cond ( aa >= 13 )   
 cond ( aa <= 19 )  
 print 1 .  
end

$p4\_6 - loop, print input, input-1...down to 1

let aa = 1 .  
main   
 scan aa .   
 loop ( aa > 0 )   
 start  
 print aa .  
 aa ~ aa - 1 .  
 stop  
end

#p4\_7 - testing double loop, print input, input-1...down to 1, then start again with input-2, input-3...down to 1 etc

let aa = 1   
 ab = 2 .  
main   
 scan aa .   
 loop ( aa > 0 )   
 start  
 ab ~ aa .  
 loop ( ab > 0 )  
 start  
 print ab .  
 ab ~ ab - 1 .  
 stop  
 aa ~ aa - 2 .  
 stop  
end