

Introduction

In this lab you will modify your parser to generate 3-address code which can be executed by my Mad Squirrel Interpreter.

Your Mission

This assignment has three parts:

I. Fix any remaining problems from Projects 1, 2 and 3 (30 pts)

II. Adding "booleans" to your symbol table (30 pts)

Update your symbol table so that it can support "boolean" variables. We will need this so that we can insert temporary variables into the symbol table.

III. Write an "generateCode" function for each ParseNode subclass. (30 pts)

The generateCode function will not return anything, but will print out 3 address code representing the node. For some of your nodes, you may need to take a parameter (which tells the function what variable to store the result in). However, most of them will not need parameters.

Generating code for a "statements" node:

To generate code for a statements node, simply generate code for each of the statements by calling "generateCode" recursively on the children of the node.

Generating code for a "statement" node:

Simply generate code for the appropriate descendant (i.e. an "eatStatement" node).

Declaration Nodes:

You don't need to generate any code for these, since they are taken care of in the symbol table, rather than in the code.

While Nodes:

You do NOT need to create code for while loops, unless you want to earn glitter points. Instead just print a "NOP" instruction.

Here's how I would implement a while loop:

Create a new temporary variable to represent the boolean condition and insert it into the symbol table. Generate the code for computing the boolean expression by calling "generateCode" on the child node. I would pass the temporary variable as a parameter to the function, so that it can store the result in the correct place.

Then add a line with a unique label (for instance, "TopOfLoopX", where you replace X with a unique integer identifier for the loop). The line can just have a "NOP" instruction.

Then use a recursive call to "generateCode" for the statementsNode to create code for what's inside the While Loop. At the bottom of the loop, put code for checking the loop condition again, using a DIFFERENT temporary variable. Then add an ifgoto line at the bottom that jumps to the labeled line if the loop condition is true.

If Nodes:

Here is one way to generate code for an if-Node:

- A. Create a new temporary variable. Insert it into the symbol table.
- B. Create a unique string, "ElseLabelX", where X is a unique integer, for THIS if statement.
- C. Create another unique string, "EndIfX", where X is a unique integer, for THIS if statement.
- D. Call "generateCode" on the ParseNode which represents the condition. Pass your temporary variable as a parameter to the function so that it knows where to store the result.
- E. Print an "ifgoto" command that jumps to the "else label" you created in step B whenever the condition is false.
- F. Use recursion to generate code for the statements in the "if" part.
- G. Print a "goto" to jump to the "ENDIF" label.
- H. Print the "ElseLabel" you wrote in step B and use recursion to generate code for the statements in the "else" part.

Eat Nodes:

To write code for an eat node, create a new temporary variable (and insert it into the symbol table). Use a recursive call to "generate code" to print code which computes the value of the int Expression and stores it in your temporary variable. Then print a line with the syntax:

```
eat x
```

Where "x" is your temporary variable.

Push Nodes:

You do not need to generate code for a push node. However, if you want to implement this code for glitter, I would do the following:

To generate code for a push node, create a new temporary variable (and insert it into the symbol table). Use a recursive call to "generate code" to print code which computes the value of the "direction expression" and stores it in your temporary variable. Then print a line with the syntax:

```
push x
```

where "x" is the temporary variable.

Assignment Nodes:

Use recursion to generate code for the right hand side and store the result in a temporary variable. Then print a line which copies the temporary into the left hand side.

Boolean Expression Nodes:

One way to do generate code for a boolExpr node is for the

generateCode

function to take a parameter: the symbol table index of the temporary variable to store the result in.

You should be able to generate code for the == comparison of integers and the > expression of integers. You do not need to handle any

other Boolean Expressions, unless you want to do so for glitter.

Int Expression Nodes and their descendants:

You need to handle three cases:

A. The Int expression is a constant.

Generate the code:

$x = c$

Where x is the result variable and c is the constant.

B. The Int Expression is the sum of two int expressions

Generate the code:

$x = a + b$

Where a and b are the int expressions (you will need to first create temporary variables for both a and b, then use recursion to generate code for them).

C. The Int Expression is the difference of two int expressions.

As above, use recursion to calculate two temporary variables (a and b), then generate the code:

$x = a - b$

where "x" is the result variable.

You do not need to handle any other case, except as a way to earn

glitter.

Tree Expressions:

**You do not need to generate code for Tree Expressions.
Just generate "NOP".**

Direction Expressions:

This should be similar to generating code for an integer constant.

You do not need to handle the case that the direction is a variable name.

To handle the parentheses case, simply use recursion to make the child node generate code for itself.

III. Glitter

(10 pts)

Glitter points are assigned for extending the project assignment in a creative way. The best way to earn glitter is to add code generation support for some of the language features we have omitted.

**As usual, submission is through the course interface at
<http://narnia.homeunix.com/~robert/submit>**