EECS665

Compiler Construction

Drew Davidson
Ruturaj Vaidya

Lecture: LEEP2 G415
MWF 3:00-3:50

Lab: Eaton 1005B

ANOUNCEMENTS | LAB | SCHEDULE | MATERIALS | ASSIGNMENTS

Project 6

Due on December 6th 11:59 PM

Not accepted late

Updates

1. None yet

Overview

For this assignment you will write a code generator that generates MIPS assembly code (suitable as input to the Spim interpreter) for Lil' C programs.

Specifications

General Information

Similar to the fourth and fifth assignments, the code generator will be implemented by writing codeGen member functions for the various kinds of AST nodes. *See the on-line Code Generation notes (as well as lecture notes) for lots of useful details.*

In addition to implementing the code generator, you will also update the Makefile and the main program (and call it P6.cpp) so that, if there are no errors (including type errors), the code generator is called after the type checker. The code generator should write code to the file named by the second command-line argument.

Note that your main program should no longer call the unparser, nor should it report that the program was parsed successfully.

Also note that you are *not* required to implement code generation for structs or anything struct-related (like dot-accesses).

Getting Started

As in prior projects, you are encouraged to build the new code atop your prior implementation. You may find the following tarball of starter files helpful to get you started: p6.tgz Please note that this code is designed to be a starting point for working on project 6, not a complete solution to P5.

Some useful code-generation methods can be found in the file `code_generation.cpp`. Note that to use the functions and constants defined in that file you will need to prefix the names with Codegen.; for example, you would write: Codegen::genPop(Codegen::T0) rather than genPop(T0). Also note that an output stream is declared as a public member of the Codegen class. The code-generation methods in Codegen.java all write to that stream, so you should use it when you open the output file in your main program. You should also close that stream at the end of code generation.

Finally, there is an example file hello.lilc and the corresponding MIPS code `hello.s`.

Spim

The best way to test your MIPS code is using the simulator SPIM, which is officially supported by

The best way to test your MIPS code is using the simulator SPIM, which is officially supported by the class. You are also free to use MARS or QtSpim, but installation is up to you in that case.

Accessing spim:

- Installed on the lab computers at `/home/a807d786/spim_deploy/bin/spim`
- Available as a binary package / .deb at `https://sourceforge.net/projects/spimsimulator/files/`
- Available as source code the following svn repository: `https://svn.code.sf.net/p/spimsimulator/code/ spimsimulator-code` (run `svn checkout https://svn.code.sf.net/p/spimsimulator/code/ spimsimulator-code`)

Accessing QtSpim:

- Available as a binary at the following link: `https://sourceforge.net/projects/spimsimulator/files/`
- Also available as source as port of the same svn repository (`https://svn.code.sf.net/p/spimsimulator/code/`), but building it is somewhat painful.

Both of these tools use the same backend, but I recommend using QtSpim on your own machine since it has a much more modern interface.

You should make sure that your program exits gracefully. To do so, the end of your main function should issue syscall 10 (exit) by doing

```
li $v0, 10
syscall
```

(Note that this means that a program that contains a function that calls `main` won't work correctly. That's fine for the purposes of this project).

Changes to old code

Important changes:

1. Add to the name analyzer or type checker (your choice), a check whether the program contains a function named main. If there is no such function, print the error message: "No main function". Use 0,0 as the line and character numbers.

2. Add a new "`offset`" field to the `SymbolTableEntry` class (or to the appropriate subclasses). Change the name analyzer to compute offsets for each function's parameters and local variables (i.e., where in the function's Activation Record they will be stored at runtime) and to fill in the new offset field. Note that each scalar variable requires 4 bytes of storage. You may find it helpful to verify that you have made this change correctly by modifying your unparser to print each local variable's offset.

Suggested changes:

1. Modify the name analyzer to compute and save the total size of the local variables declared in each function (e.g., in a new field of the function name's symbol-table entry). This will be useful when you do code generation for function entry (to set the SP correctly).

2. Either write a method to compute the total size of the formal parameters declared in a function, or modify the name analyzer to compute and store that value (in the function name's symbol-table entry). This will also be useful for code generation for function entry.

3. Change the definition of class WriteStmtNode to include a (private) field to hold the type of the expression being written, and change your typecheck method for the WriteStmtNode to fill in that field. This will be useful for code generation for the *write* statement (since you will need to generate different code depending on the type of the expression being output).

Non-obvious semantic issues

- All parameters should be passed by value.
- The *and* and *or* operators (`&&` and `||`) are *short circuited*, just as they are in C/C++. That means that their right operands are only evaluated if necessary (for all of the other binary operators, both operands are always evaluated). If the left operand of "&&" evaluates to *false*, then the right operand is not evaluated (and the value of the whole expression is *false*); similarly, if the left operand of "||" evaluates to *true*, then the right operand is not evaluated (and the value of the whole expression is *true*).

- Boolean values should be output as 1 for *true* and 0 for *false* (and that is probably how you should represent them internally as well).

- Boolean values should also be input using 1 for *true* and 0 for *false*.

Suggestions for how to work on this assignment

1. Modify name analysis or type checking to ensure that a main function is declared.

2. Modify name analysis so that the code generator can answer the following questions:

    - Is an Id local or global?
    - If local, what is its offset in its function's AR?
    - For each function, how many bytes of storage are needed for its params, and how many are needed for its locals?

3. Implement code generation for each of the following features; be sure to test each feature as it is implemented!

    - global variable declarations, function entry, and function exit (write a test program that just declares some global variables and a main function that does nothing)
    - int and bool literals (just push the value onto the stack), string literals, and WriteStmtNode
    - IdNode (code that pushes the value of the id onto the stack, and code that pushes the address of the id onto the stack) and assignments of the form id=literal and id=id (test by assigning then writing)
    - expressions other than calls
    - statements other than calls and returns

- statements other than calls and returns
- call statements and expressions, return statements (to implement a function call, you will need a third code-generation method for the IdNode class: one that is called only for a function name and that generates a jump-and-link instruction)