

CS 415 Project - Spring 2000

Department of Computer Science
Rutgers University

Due 7pm, Monday 5/1

1 Building a Code Generator

The purpose of this project is to provide you with an opportunity to apply compiler-construction techniques discussed in class and in the textbook, and to give you experience with generating code for a modern RISC microprocessor.

Given the scanner, parser, type checker, and symbol table that you have already built, you now need to generate assembly code by building up a list of pseudo instructions provided by Prof. Pugh (University of Maryland). You can build this list of instructions by designing and invoking action routines in your yacc grammar.

The target machine is a MIPS R2000 RISC computer. Prof. Larus' (University of Wisconsin-Madison) SPIM S20 simulator `spim` is available on the undergraduate network (called `spim`). You can find the `spim` documentation through our WWW home page. There is also an X-Windows version of `spim`, called `xspim`. Once in `Spim`, you can test your compiler's assembly output (placed in a file called `temp`) by using the "load" command or button in `Spim`.

We will use Prof. Pugh's (University of Maryland) data type `InstructionSeq` to accumulate translated code. The data type is defined in `instruct.h` and `instruct.c`. A good place to start the project is to examine the op codes found in `instruct.h`. You'll need to make several changes to the attributes from the last project:

1. Added `InstructionSeq` attributes for the code, `tchain`, and `fchain` values of each expression. You are required to implement short-circuit evaluation of logical operators.

```
typedef struct _expnode {
    ...;
    InstructionSeq code;
    InstructionSeq tchain, fchain;
    ...;
}* expnode;
```

2. Added `InstructionSeq` attributes for the code and next lists for all statements.

```
typedef struct {
    InstructionSeq code;
    InstructionSeq next;
```

```
    } stmtattr;
```

3. Added InstructionSeq attributes for the code, next, and declaration components of blocks (to avoid having to branch around code for nested procedures to get at the executable portion).

```
typedef struct {
    InstructionSeq code;
    InstructionSeq next;
    InstructionSeq dclcode;
} blockattr;
```

After synthesizing your code in an attribute for program, you should call assignRegs to convert virtual registers to real registers and call printAsm to produce input for SPIM.

You may assume that all arrays have lower bounds of 0, and are passed by reference when they appear as actual parameters. You may also assume that all procedures are uniquely named, and that none are named "main" (which you should use as the name of the program no matter what identifier the user chose).

Stack frame structures and calling conventions are described on page 22 of the SPIM documentation. Register \$gp points to the global data. The frame pointer (\$fp) points at the first word in the frame and the stack pointer (\$sp) points just beyond the frame. Frames are laid out as follows:

Offset	Use
0	dynamic link (old \$fp value)
-4	return address (\$ra)
-8	static link
-12	return value (functions only)
-16	temporary storage for character writes
-20	start of 10-word area to save \$t0 - \$t9

Thus the first offset for a formal parameter or local variable is -60.

When a call is made, arguments are passed in registers \$a0 - \$a4. The first register is reserved for the frame pointer of the routine which encloses the routine being called. This value becomes the called routine's static link. The called routine stores the dynamic link, return address, and static link on entry; and restores the dynamic link and return address on exit. In addition, functions return values in \$v0. A return value must be moved from \$v0 to \$tn to participate in expression evaluation.

2 What To Do

To get started, you should download project4.tar.gz from the class web. As usual, you create the executable by typing make.

WARNING - Start early!!!! This is by far the hardest assignment, not to mention that you have to get all the previous pieces working. If you do start early, the pay-off is that you will get the satisfaction of seeing your compiler actually produce code that you can run.