CS75 Project 3a: Basic Code Generation for C- -

Spring 2007, Meeden

Due by midnight, Monday, April 9

Introduction

For this project you will add code generation to your C- - compiler. Your compiler will take the name of a C- - program file as input and produce a MIPS assembly language program as output. To execute the MIPS program produced you can use the `spim` or `xspim` simulators.

Of the four projects we will complete in this course, this project (consisting of parts a and b) is the most difficult, but also the most interesting because you finally see how it all fits together. You should start by reviewing the handout provided in class describing the MIPS assembly language.

I strongly advise you to do a small section of code generation at a time and to test each section thoroughly before attempting additional sections. You should create a test suite of C- - programs to test your compiler. As you add new sections to the code generator you should re-test your compiler on your entire test suite to be sure that your didn't break something that was previously working.

Part A: Basic Functionality

In order to break down the process of code generation into small, incremental steps, we will initially ignore variables and function calls. We will assume that all programs only have a `main` function that deals directly with numbers. For each step described below, I have provided at least one test program.

1. Because you will need `write` and `writeln` to test all aspects of code generation, you should begin by implementing them along with all of the arithmetic operators.

```
int main() {
  write 5*2-1;
  writeln;
}
```

2. Implement the relational operators.

```
int main() {
  write 4 == 5;  //0
  write 4 >= 5;  //0
  write 4 > 5;   //0
  write 4 < 5;   //1
  write 4 != 5;  //1
  write 4 <= 5;  //1
  writeln;
}
```

3. Implement the boolean operators. Remember that in the C- - language `and` and `or` are short-circuit operators. In other words, if the first operand of an `and` is false, then false is returned and the second operand is never evaluated. Similarly, if the first operand of an `or` is true, then true is returned and the second operand is never evaluated.

```
// tests or
int main() {
  write (4 == 5) || (4 > 5);  //0
  write 1 || 0;               //1
  write (1 == 2) || (5 < 6);  //1
  writeln;
}
```

```
// tests and
int main() {
  write (4 != 5) && (5 != 6); //1
  write 1 && 0;               //0
  write 0 && 1;               //0
  writeln;
}

// tests not
int main() {
    write !(3 < 4); //0
    write !(4 < 3); //1
    write !!1;      //1
    writeln;
}
```

4. Implement `if` statements.

```
int main() {
  if (5 < 3)
    write 1;
  else
    write -1;
  writeln;
  if (5 < 6)
    write 1;
  else
    write -1;
  writeln;
}
```

5. Implement `while` loops, but don't worry about `break` statements yet.

```
// Should go into an infinite loop printing 2's
int main() {
  while (1)
    write 2;
}

/*
   Should output:
   1
   3
   While loop should never be executed.
*/
int main() {
  write 1;
  writeln;
  while (0) {
    write 2;
    writeln;
  }
  write 3;
  writeln;
}
```

Once all of this functionality is in place, then you can move on to the more complicated aspects of code generation in Part B of this project.