MC97 Lab 3: Yacc and Control Flow

Due March 20, 1996; Max Hailperin

You only have to turn in the final compiler you have at the end of this lab. However, I strongly urge you to stop and test your compiler at several intermediate points. Be sure also to pay attention to the shift/reduce and reduce/reduce conflicts that Yacc reports and ensure that each one is correctly resolved.

1. Convert your compiler to use Yacc. Address any grading remarks from the previous lab that are still relevant.

2. Add a unary minus operator, of greater precedence than multiplication and division.

3. Change the syntax of a program to be a sequence of zero or more declarations followed by zero or more statements. Each declaration is a type followed by a comma-separated list of one or more identifiers followed by a semicolon. For now, the only type is the reserved keyword `int`. Signal an error if a variable is used (even on the left-hand side of an assignment) without being declared. Allocate a register for each variable when it is declared, rather than waiting until the first time it is assigned. Remove the checking for use before assignment, since we are about to add loops and conditionals, which would make this checking undecidable.

4. Add simple panic-mode error recovery by using error production(s) that skip up through the next semicolon when there is an error and then resume parsing.

5. Add five new statement types:

   (a) `;` (i.e., an empty statement)

   (b) `while (`*expr*`)` *stmt*

   (c) `if (`*expr*`)` *stmt*

   (d) `if (`*expr*`)` *stmt* `else` *stmt*

   (e) `{`*stmt*`*}`

   To implement these, you will have to generate labels and instructions that conditionally or unconditionally branch to those labels. A label is a name followed by a colon; conventionally compiler-generated labels are given the names `L$0`, `L$1`, ... To jump unconditionally to a label, use the MIPS operation `j`. To branch only if a specified register is equal to zero, you can use `beqz`, while `bnez` does the branch only if the register is not equal to zero. For example, here is how a `while` loop might look:

```
while(i){
  print i;
```

```
  i = i - 1;
  }
```

Below is the MIPS code you might generate for this sample loop, assuming `i` is stored in `$s0`. (Notice there is still much room for optimization; this is typical of what your compilers are generating at this point.)

```
L$0:
        move    $t0, $s0    ;compute the test, result is in $t0
        beqz    $t0, L$1    ;if test is false (0), end loop
        move    $t0, $s0    ;do body of loop
        move    $a0, $t0
        li      $v0, 1
        syscall
        move    $t0, $s0
        li      $t1, 1
        sub     $t0, $t0, $t1
        move    $s0, $t0
        j       L$0         ;go back to top of loop
L$1:
```

6. Add six new operators: `<`, `>`, `<=`, `>=`, `==`, and `!=`. They produce integer results, namely 1 if the relation holds and 0 if it doesn't. They are all of equal precedence, less than that of addition and subtraction, and they are non-associative. These operators correspond to the MIPS instructions `slt`, `sgt`, `sle`, `sge`, `seq`, and `sne` respectively. These are used the same way as `add` or `sub`; they all also have immediate versions. For example, `slt $t1, $t2, $t3` sets `$t1` to 1 if `$t2` $<$ `$t3`, otherwise sets it to 0. Further, `sgt $t1, $t2, 5` sets `$t1` to 1 iff `$t2` > 5.