McMaster University Comp Sci 4TB3/6TB3, Winter Term 2018/19 — Lab 9
For the Labs on March 12 - March 16,
Due Monday, March 19, 11 pm

Eden Burton, Jenny Wang, Spencer Park
out of 32 points

- *Submission is to be done exclusively through Avenue. Submissions via e-mail will* ***not*** *be accepted. A* ***10% penalty*** *will be accessed for each day the lab is submitted after the due date.*
- You need to download the P0 compiler posted together with this lab, it contains some required modifications.
- This assignment requires access to a Linux, MacOS X, or some other Unix computer. You can log in remotely to either `moore.mcmaster.ca` or to `mills.mcmaster.ca` with ssh. Submissions are tested on `moore.mcmaster.ca`, please check if your submission works there.
- In this lab, you are allowed to work in pairs, provided that you split the work equally and arrive at a common understanding of the solution. However, in that case you must state in your submission the person you worked with, such that similarities in the solution will not be construed as Academic Dishonesty. Working in groups of three or larger is not allowed and will be considered Academic Dishonesty. If you look for someone to work with, we will try to find a match, please contact the TAs.
- You are allowed and encouraged to talk to everyone in the course to get a common understanding of the problem, but you can share partial solutions only with your collaborator, if you work in a pair. The final submission must be your own, that is, you cannot submit identical submissions with two names on them.
- In the lab sessions, the solution to last week's lab questions are discussed and you can get help with this week's lab questions. Attendance at the labs is not checked.

Lab Question 1 (Extending P0 with Bitwise Operations, 26 points)**.** Extend P0 with

- bitwise complement **e**, where **e** is an integer expression
- bitwise and **e & f**, where **e**, **f** are integer expressions
- bitwise or **e | f**, where **e**, **f** are integer expressions

~~~
For example:
```Pascal
program bitwise;
  var x: integer;
  var p: boolean;
  procedure isPowerOfTwo(i: integer; var r: integer);
    {a power of 2 has only one bit set, so i & (i-1)
    will have all bits cleared, except if i is 0}
    begin
      r := i & (i-1)
    end;
  begin
    read(x);
    isPowerOfTwo(x, p);
    if p = 0 then write(1) {is 0 or power of 2}
    else write(0) {is not power of 2}
  end
```
~~~

1. Extending the Scanner [6 points]

   - in *SC*, introduce new symbols *TILDE*, *AMP*, and *BAR*; extend the imports in *P0*, *CGmips* accordingly.
   - extend the production of *symbol* in the text cell above *getSym()* to include -, &, and |.
   - extend *getSym()* to return *TILDE*, *AMP*, *BAR* when recognizing -, &, |.

2. Extending the Parser [12 points]

   - in *P0*, extend the productions of *factor*, *term* and *simpleExpression* in the text cell above the corresponding functions to include the unary operator   to bind as tight as *not*, binary operator & to bind as tight as *and*, *, and binary operator | to bind as tight *or*, +. That is, *a* | *b & c* would be parsed in the same way as *(a)* | *(b & c)*.
   - extend *factor()* to parse unary   similarly to *not*. If the operand is not *Int*, an error *not integer* is generated, if the operand is a constant, the expression is evaluated ("constant folding") by using Python's , otherwise the *genUnaryOp* of the code generator is called.
   - extend *term()* to parse binary & similarly to *and*, *. If not both operands are *Int*, an error *bad type* is generated, if both operands are constant, the expression is evaluated by using Python's &, otherwise *genBinaryOp* of the code generator is called.

- extend *simpleExpression()* to parse binary | similarly to *or*, +. If not both operands are *Int*, an error 'bad type' is generated, if both operands are constant, the expression is evaluated by using Python's |, otherwise *genBinaryOp* of the code generator is called. - extend the first and follow sets of factor and expression accordingly.

3. Extending the Code Generator [8 points]

   - in *CGmips*, extend *genUnaryOp* to generate code for bitwise complement when *op* is *TILDE*. The MIPS architecture does not have an instruction for bitwise complement, only for *r nor s*, which is defined as *(r | s)*. When taking *s* to be *r*, the result is *r*. Follow the pattern for generating the code for *MINUS*.
   - extend *genBinaryOp* to generate code for *&* and | when *op* is *AMP* and *BAR*. Follow the pattern for arithmetic operations.

Testing

Test the generated MIPS code by running it in SPIM, http://spimsimulator.sourceforge.net/. The expected MIPS code for the program below is:

```
compileString("""
program bitwise;
  var x, y: integer;
  begin
    {testing constant folding}
    write(~1);       {writes -2}
    write(2 & 6);    {writes 2}
    write(2 | 4);    {writes 6}
    {testing code generation}
    x := 3; y := 5;
    write(x & y);    {writes 1}
    write(~x);       {writes -4}
    write(x | y)     {writes 7}
    {immediate addressing mode}
  end
""")
```

~~~
```mips
    .data
y_: .space 4
x_: .space 4
    .text
    .globl main
    .ent main
main:
    addi $a0, $0, -2
    li $v0, 1
    syscall
    addi $a0, $0, 2
    li $v0, 1
    syscall
    addi $a0, $0, 6
    li $v0, 1
    syscall
    addi $t2, $0, 3
    sw $t2, x_
    addi $t5, $0, 5
    sw $t5, y_
    lw $t3, x_
    lw $t4, y_
    and $t3, $t3, $t4
    add $a0, $t3, $0
    li $v0, 1
    syscall
    lw $t8, x_
    nor $t8, $t8, $t8
    add $a0, $t8, $0
    li $v0, 1
    syscall
    lw $t1, x_
    lw $t0, y_
    or $t1, $t1, $t0
    add $a0, $t1, $0
    li $v0, 1
    syscall
    li $v0, 10
    syscall
    .end main
```
~~~

Lab Question 2 (Extending P0 with Single Line Comments, 6 points)**.** Add single line comments that start with // and extend until the end of line. Modify SC accordingly. Note that each call to getSym has to assign the next symbol to sym; see how this is done for comments in braces.

```
compileString("""
program singlelinecomment;
  {demonstrates both comments and
  bitwise & with constant operand}
  var x: integer;
  begin
    x := 3;
    write(x & 5); // writes 1
    write(~x)     // writes -4
  end
""")
```