Project 2 – Intermediate Code Generation

Due date: May 8, 2013 at midnight

Description: In this third part of the compiler project, you will be asked to extend the compiler you have developed for miniC, from project 1, with intermediate code generation using the three-address instruction described in class.

The project shall be developed individually. You are encouraged to discuss with other students, ask teachers, search for ideas on the Internet. ***But, do not copy code***. The solutions will be tested in a UNIX environment as with previous projects. The evaluations of the solutions are based on these tests. The compiler can be developed in many environments. But, when the solutions are delivered, it must be possible to generate the lexical analyzer and the parser from their source codes, and compile the results. See the instruction on how to turn-in your projects.

Intermediate Code Generation. In the first part of the project, given an input, you constructed an abstract syntax tree (AST). In order to generate the intermediate code, you will now traverse this tree in depth-first order and perform the translations. The intermediate code should be "abstract" three-address code as described in class. To simplify this project and allow you to focus on the critical aspects of intermediate code generation rather than the minute details, we make the simplifying assumption that you will only deal with input programs that do not have structure accesses. Also, all array variables will be one-dimensional. As a result you will focus on generating correct code for assignments, including function calls, and control-flow instructions. Note that you still need to generate all temporary variables when generating the code that is responsible for the evaluation of a given expression, including the array access expressions.

Function/procedure calls. For a function/procedure call `p(x1,x2,…,xn)`, you should generate code that explicitly uses the notion of activation records (AR) on the stack and saves and retrieves the values of the various arguments as offsets from the AR pointer (ARP). As such you will need to make use of three registers defined in the target (simulated) architecture, namely, the Stack Pointer ($sp), the Frame Pointer ($fp), the Return Address ($ra) and the Program Counter ($pc).

The example below illustrates the use of these registers for an invocation of a procedure `f1` with two arguments "a" and "b", which have been assigned offsets 1 and 2 off of the stack pointer (which itself is set at a specific offset of the frame pointer register ($fp).

Notice that for simplification, in the simulator you will be using to validate your generated code, all objects are addressed as in an array, independent of their data type sizes. There are many ways to generate correct code. The example below is just a specific illustration. You are welcome to implement you own compilation strategy as long as the simulator produces correct results.

```
        .section .text "main"
main:    $sp = $fp             # construct dummy
         $sp = $sp - 1         # frame at the
        *$sp = -1              # beginning of the
         $sp = $sp - 1         # main program
        *$sp = 0               # execution.
         $sp = $sp - 1         #
```

```
        # initialization of global variables: a, b, and c
        a = 1
        b = 2
        c = 1

# call section c = f1(a,b)
       *$sp = a       # setting the first argument slot and
        $sp = $sp - 1 # advancing the $sp to the next slot
       *$sp = b       # setting the second argument slot and
        $sp = $sp - 1 # advancing the $sp to the next slot
       *$sp = $fp     # saving the old frame point in the AR
        $fp = $sp     # setting new $fp to the current $sp
        $ra = $pc + 6 # setting the $pc to "jump" over the next
        $sp = $sp - 1 # instructions and "advancing" the $sp
       *$sp = $ra     # saving the return address in the AR
        $sp = $sp - 1 # advancing the $sp
       *$sp = 0       # setting the return value of the AR
        $pc  = &f1    # jump to function f1 - the next instruction
                      # is executed on the return from the function.
        $r3 = $sp - 2 # retrieve the return value
        c = *$r3      # load from AR and integrate return value
        halt

f1:     $r1 = $fp + 1 # get the address of the second argument
        $r1 = *$r1    # retrieve second argument $r1 = b
        $r0 = $fp + 2 # get the address of the first argument
        $r0 = *$r0    # retrieve first argument $r0 = a
        $r2 = $r0 + $r1 # compute the actual function as $r2 = a + b
        $r3 = $fp - 2 # get address of the return value in AR
       *$r3 = $r2     # depositing return value
                      # in the return slot of AR
        $ra = $fp - 1 # get the address of the return address
        $ra = *$ra    # retrieve the address of the return address
        $sp = $fp     # pop the stack frame (adjust $sp)
        $fp = *$fp    # pop the stack frame (adjust $fp)
        $pc =  $ra    # return back to caller
```

Note that in the case of a more complicated expression for the values of x1, …, xn, your code needs to generate the code that evaluates these expressions into scalars, say t1, …, tn and then generate the pre-call instruction section; call instruction followed by the post-return section that integrate the result of the function in the caller invocation site. This is simpler than it looks once you have a scheme to create the skeleton of the AR on the stack and know the offsets of the individual arguments. These offsets should have been calculated as part of your work in project 1.

In order to generate the three-address code for a function/procedure call, it may be convenient to first save the values of the actual parameters in a queue as in the following pseudo code:

```
  statement -> IDENTIFIER '(' argument_expression_list ')'
    {s="";
     while not empty(queue)
       s=append(s,gen(´putparam´ dequeue(queue)));
     statement.code=append(s,gen(´putparam´, IDENTIFIER.place));}

  argument_expression_list -> argument_expression_list, argument_expression
    {enqueue(argument_expression.place);}

  argument_expression_list -> argument_expression
    {initialize queue to contain only argument_expression.place}
```

Output of your project

As in the second part of the project, the parser repeatedly calls the lexer, which returns the next token, and during parsing the input a parse tree is explicitly created. Now, add code that traverses the parse tree depth-first and generate three-address code:

```
main(){
  do{yyparse();}
  while(!feof(yyin));
  /* code for traversing (dfs) the syntax tree and generating three-address code */
}
```

Let the source code of the lexer be in file lex3.l and the source code to the parser in file yacc3.y. If the input to the parser is not grammatically correct then the parser says on which line the first error is, and then terminate (just as in the first part of the project). If there are no errors, the output should be the three-address code.

Validating your Output using Three-Address Instruction Simulator:
You can validate your output by using the three-address instruction simulator made available at the class web site. This is the same simulator we are going to use to grade your project. To this effect you can generate a data section of the assembly as shown below.

```
        .section .data
a:      .word    0
b:      .word    0
c:      .word    0
```

How to Generate a Parser Executable:
Generate the lexer and parser using flex and yacc, respectively:

```
lex lex2.l
yacc -d -v yacc2.y
```

The results consist of the files lex.yy.c, y.tab.c and y.tab.h. Compile and link:

```
gcc -c lex.yy.c
gcc –c y.tab.c
gcc lex.yy.o y.tab.o -ll
```

Some test cases with the correct results can be found at the class website. Test and make sure that everything works.

Turn-in Instructions:
As with the first programming assignment you will turn in your project if you are student number `XYZ`, you need to create a file named `XYZ.proj2.tar.zip` created using the `zip` and `tar` utilities, and make sure that the commands

```
unzip XYZ.proj2.tar.zip
tar -xvf XYZ.proj2.tar
cd XYZ.proj2
make
```

results in the creation of a folder `XYZ.proj2` with executable file `a.out` in it. Please make sure you do understand this. Also, inside this file there cannot be any input and/or output files and your makefile will have to create the lexer and parser files using the flex/lex and yacc/bison commands.