Project 3 – Intermediate Code Generation

Due date: June 13, 2008 at midnight

Description: In this third part of the compiler project, you will be asked to extend the parser for miniC, from project 2, 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 second part of the project, given an input, you constructed a syntax tree. 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 intemediate code generation rather than the minute details, we make the simplifying assumption that you will only deal with input programs that have integer variables, either scalars or arrays. There will be not structure accesses or character variables.

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 fo rthe 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 of the form

```
param x1
param x2
   .
   .
   .
param xn
call p,n
```

where `xi` is the place with the value of argument i and `n` is the number of actual parameters to `p`. 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 call instruction.

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 call 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 second part of the project). If there are no errors, the output should be the three-address code.

Generated Code Example
If there are no errors then the parser shall output the three address code for the input program. For instance parsing the program below on the left should result in the code on the right.

```
int A[10];
int main(){
  int i;
  int max;
  int sum;
  i=0; sum = 0;
  while(i < 10){
    max = findMax(i);
    A[i] = max;
    Sum = sum + max;
    i = i + 1;
  }
  print_int(sum);
}
```

```
main:
        i = 0
        sum = 0
    L1: if i < 10 goto L2
        goto L3
    L2: putparam i
        max = call findMax, 1
        A[i] = max
        t1 = sum + max
        sum = t1
        t2 = i + 1
        i = t2
        goto L1
    L3: putparam sum
        call print_int, 1
        halt
```

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. The test codes we are going to be using alongside the ones we are providing, will make use of the *iread* and *iwrite* functions to read and write integer values to the console. See the example codes available with the simulator source code available at the web site.

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

```
flex lex3.l
yacc -d -v yacc3.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: In your Unix account on the tlinux machine, in the folder xfer, place a file named proj3.tar.zip created using the zip and tar utilities:

```
tar –cvf gcc lex.yy
```

Make sure that the commands

```
unzip proj3.tar.zip
tar -xvf proj3.tar
make
```

results in an executable file a.out and that

```
a.out < test.in
```

results in an output as is described above.