EECS 665 – Fall 2015

Project 2

csem

csem reads a C program (actually a subset of C) from its standard input and compiles it into a list of intermediate language quadruples on its standard output. The form of the quadruple operators appear below:

$x := y\ op\ z$	operate on y and z and place result in x
bt x *lab*	branch to *lab* iff x is true
br *lab*	branch to *lab*
$x :=$ **global** *name*	yield address of global identifier *name*
$x :=$ **local** n	yield address of local n
$x :=$ **param** n	yield address of parameter n
$x := c$	yield value of constant value c
$x := s$	yield address of character string s
formal n	allocate the formal having n bytes
alloc *name* n	allocate the global name having n bytes
localloc n	allocate the local having n bytes
func *name*	begin function *name*
fend	end function
lab=y	define *lab* to be y
bgnstmt n	beginning of statement at line n

name denotes an identifier from the C program. n denotes an integer. c denotes a C integer constant. s denotes a string enclosed by double quotes. x, y, and z denote quadruple temporaries. *lab* denotes the location of a quadruple or a reference to a symbol defined later by a "lab=y" command. *op* denotes any of the C operators below:

== != <= >= < > = | ∧ << >> + − * / %	operate on x and y
∼	invert x
−	negate x
@	dereference x
cv	convert x
f	call function y with n arguments
arg	pass x as an argument
ret	return x
[ ]	index z into y

followed by **i** (for the integer version of the operator) or by **f** (for the floating point version). y is omitted for unary operators. You should assume all bitwise operators (|, ∧, &, <<, >>, ∼) and % only operate on integer values.

For example,

```
double m[6];

scale(double x) {
   int i;

   if (x == 0)
      return 0;
   for (i = 0; i < 6; i += 1)
      m[i] *= x;
   return 1;
}
```

compiles into the intermediate operations below (actually only one column)

```
alloc m 48          t7 := local 0          t19 := local 0
func scale          t8 := 0                t20 := @i t19
formal 8            t9 := t7 =i t8         t21 := global m
localloc 4          label L3               t22 := t21 []f t20
bgnstmt 6           t10 := local 0         t23 := param 0
t1 := param 0       t11 := @i t10          t24 := @f t23
t2 := @f t1         t12 := 6               t25 := @f t22
t3 := 0             t13 := t11 <i t12      t26 := t25 *f t24
t4 := cvf t3        bt t13 B3              t27 := t22 =f t26
t5 := t2 ==f t4     br B4                  br B6
bt t5 B1            label L4               label L6
br B2               t14 := local 0         B3=L5
label L1            t15 := 1               B4=L6
bgnstmt 7           t16 := @i t14          B5=L3
t6 := 0             t17 := t16 +i t15      B6=L4
reti t6             t18 := t14 =i t17      bgnstmt 10
label L2            br B5                  t28 := 1
B1=L1               label L5               reti t28
B2=L2               bgnstmt 9              fend
bgnstmt 8
```

Your assignment is to write the semantic actions for the csem program to produce the desired intermediate code. The following files which will comprise part of your program should be downloaded from the class web-page: http://www.ittc.ku.edu/~kulkarni/teaching/EECS665/

`cc.h`	- include file
`cgram.y`	- yacc grammar for subset of C
`makefile`	- csem makefile
`scan.c`	- lexical analyzer
`scan.h`	- defines prototypes for routines in scan.c
`sem.h`	- defines prototypes for routines in sem.c
`semutil.c`	- utitity routines for the semantic actions
`semutil.h`	- defines prototypes for routines in semutil.c
`sym.c`	- symbol table management
`sym.h`	- defines prototypes for routines in sym.c

The makefile will create an executable called csem in the current directory. This file `sem.c` contains stubs for the semantic action routines. While I have provided you access to the other *.c and *.h files, you should not modify them. You are only allowed to update the file sem.c and will not be allowed to update any other files. You can write additional functions in this file to abstract common operations. When making your executable, refer to the makefile provided, which uses the other *.c and *.h files when producing the executable. I have also included the file sem_base.exe that contains my implementation. You can use this executable to verify your output using the `diff` unix command. You can also use this executable to determine the three-address code that should be generated for each construct.

E-mail only the file sem.c as an attachment to your respective Lab TA (either Bharath – `belluru@ku.edu`, or Nilisha – `nilisha.mane@ku.edu`) and CC it to '`prasadk@ku.edu`' before the beginning of class on Wednesday, December 2^{nd}.

Another Example

This example shows a compilation for a test program with multiple formal parameters, locals, and actual arguments.

```
main(int a, int b)
{
    double d;
    int i;
    printf("%d %f %d %d\n", i, d, a, b);
}
```

compiles into

```
func main
formal 4
formal 4
localloc 8
localloc 4
bgnstmt 6
t1 := "%d %f %d %d\n"
t2 := local 1
t3 := @i t2
t4 := local 0
t5 := @f t4
t6 := param 0
t7 := @i t6
t8 := param 1
t9 := @i t8
argi t1
argi t3
argf t5
argi t7
argi t9
t10 := global printf
t11 := fi t10 5
fend
```