COP5621 - Spring 2020
Assignment 5
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
func scale
formal 8
localloc 4
bgnstmt 6
t1 := param 0
t2 := @f t1
t3 := 0
t4 := cvf t3
t5 := t2 ==f t4
bt t5 B1
br B2
label L1
bgnstmt 7
t6 := 0
reti t6
label L2
B1=L1
B2=L2
bgnstmt 8
t7 := local 0
t8 := 0
t9 := t7 =i t8
label L3
t10 := local 0
t11 := @i t10
t12 := 6
t13 := t11 <i t12
bt t13 B3
br B4
label L4
t14 := local 0
t15 := 1
t16 := @i t14
t17 := t16 +i t15
t18 := t14 =i t17
br B5
label L5
bgnstmt 9
t19 := local 0
t20 := @i t19
t21 := global m
t22 := t21 []f t20
t23 := param 0
t24 := @f t23
t25 := @f t22
t26 := t25 *f t24
t27 := t22 =f t26
br B6
label L6
B3=L5
B4=L6
B5=L3
B6=L4
bgnstmt 10
t28 := 1
reti t28
fend
```

Your assignment is to write the semantic actions for the csem program to produce the desired intermediate code. The following files that will comprise part of your program are in the *~whalley/asg5* directory.

`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. You should copy the `semdum.c` file into your directory as `sem.c`. This file contains stubs for the semantic action routines. While I have given you read access to the other *.c and *.h files and the `makefile`, you should not copy them into your directory. You are only allowed to update the file `sem.c` and will not be allowed to update any other files. You should make additional

functions in this file to abstract common operations. When making your executable, refer to the makefile by using the command *make -f ~whalley/asg5/makefile*, which uses the other *.c and *.h files when producing the executable. This will allow me to make updates (and perhaps occasional fixes to problems) that everyone will instantly receive. The *run.sh* script takes a single intermediate test file as a command line argument and attempts to execute the intermediate code. You should test your intermediate code on the machine *program*.

E-mail only the file `sem.c` as an attachment to *whalley@cs.fsu.edu* before the beginning of class on March 24. Please put COP5621 somewhere on the subject line of the message.

Another Example

This example shows the intermediate code generation for a test function with multiple formal parameters, locals, and actual arguments.

```
test(int a, int b)
{
   double d;
   int i;

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

compiles into

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

Below is the order in which I recommend you implement the semantic routines.

```
fname
fhead
ftail
bgnstmt
id
string
op1
exprs
call

--------------  enough to get through the second example

con
m
doret
set
op2
index
ccexpr
rel
n
backpatch
doif
dofor

--------------  enough to get through the first example

doifelse
dowhile
dodo
ccand
ccor
ccnot
opb
startloopscope
endloopscope
docontinue
dobreak
labeldcl
dogoto
```