

COP5621 - Fall 2008  
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 triples on its standard output. The form of a triple is as follows:

<triple number>.\t<triple operation>

The triple operations in the intermediate language appear below:

|                            |   |
|----------------------------|---|
| <i>op x y</i>              | operate on <i>x</i> and <i>y</i>                      |
| <b>bt</b> <i>x lab</i>     | branch to <i>lab</i> iff <i>x</i> is true             |
| <b>br</b> <i>lab</i>       | branch to <i>lab</i>                                  |
| <b>global</b> <i>name</i>  | yield address of global identifier <i>name</i>        |
| <b>local</b> <i>n</i>      | yield address of local <i>n</i>                       |
| <b>param</b> <i>n</i>      | yield address of parameter <i>n</i>                   |
| <b>con</b> <i>c</i>        | yield value of constant value <i>c</i>                |
| <b>str</b> <i>s</i>        | yield address of character string <i>s</i>            |
| <b>formal</b> <i>n</i>     | allocate the formal having <i>n</i> bytes             |
| <b>alloc</b> <i>name n</i> | allocate the global <i>name</i> having <i>n</i> bytes |
| <b>localloc</b> <i>n</i>   | allocate the local having <i>n</i> bytes              |
| <b>func</b> <i>name</i>    | begin function <i>name</i>                            |
| <b>fend</b>                | end function  |
| <i>lab=y</i>               | define <i>lab</i> to be <i>y</i>                      |

*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* and *y* denote triple numbers. *lab* denotes the location of a triple or a reference to a symbol defined later by a "*lab=y*" command. *op* denotes any of the C operators below:

|  |  |
|--|--|
| <b>== != &lt;= &gt;= &lt; &gt; =   ^ &amp; &lt;&lt; &gt;&gt; + - * / %</b> | operate on <i>x</i> and <i>y</i>               |
| <b>~</b>   | invert <i>x</i>                                |
| <b>-</b>   | negate <i>x</i>                                |
| <b>@</b>   | dereference <i>x</i>                           |
| <b>cv</b>  | convert <i>x</i>                               |
| <b>f</b>   | call function <i>x</i> with <i>y</i> arguments |
| <b>arg</b>   | pass <i>x</i> as an argument                   |
| <b>ret</b>   | return <i>x</i>                                |
| <b>[]</b>  | index <i>y</i> into <i>x</i>                   |

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,

```
float m[6];

scale(float x)
{
    int i;

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

compiles into

```
1.  alloc m 48          13. reti 12          24. local 0          37. *f 36 35
2.  func scale        14. B10=L12         25. con 1           38. =f 33 37
3.  formal 8          label L15          26. @i 24           39. B22=L30
4.  localloc 4       15. local 0         27. +i 26 25        40. B29=L18
    label L5          16. con 0           28. =i 24 27        41. br L24
5.  param 0          17. =i 15 16        29. br B29          42. B11=L15
6.  @f 5             label L18          label L30           label L43
7.  con 0            18. local 0         30. local 0         43. con 1
8.  cvf 7            19. @i 18           31. @i 30           44. reti 43
9.  ==f 6 8          20. con 6           32. global m        45. B23=L43
10. bt 9 B10         21. <i 19 20        33. []f 32 31       label L46
11. br B11           22. bt 21 B22       34. param 0         46. fend
    label L12         23. br B23          35. @f 34
12. con 0            label L24          36. @f 33
```

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 are in the *~whalley/asg5* directory.

|                  |   |
|------------------|---|
| <i>cc.h</i>      | - include file  |
| <i>cgram.y</i>   | - yacc grammar for subset of C                        |
| <i>makefile</i>  | - <i>csem</i> makefile                                |
| <i>scan.c</i>    | - lexical analyzer                                    |
| <i>scan.h</i>    | - defines prototypes for routines in <i>scan.c</i>    |
| <i>sem.h</i>     | - defines prototypes for routines in <i>sem.c</i>     |
| <i>semutil.c</i> | - utility routines                                    |
| <i>semutil.h</i> | - defines prototypes for routines in <i>semutil.c</i> |
| <i>sym.c</i>     | - symbol table management                             |
| <i>sym.h</i>     | - defines prototypes for routines in <i>sym.c</i>     |

The makefile will create an executable called *csem* in the current directory. You should copy the *sem-dum.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, 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. When making your executable, refer to the makefile in the *~whalley/asg5* directory, 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. E-mail only the file *sem.c* to *whalley@cs.fsu.edu* as an attachment before class on October 30.

## Another Example

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

```
main(int a, int b)
{
    float f;
    int i;

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

compiles into

```
1. func main
2. formal 4
3. formal 4
4. localloc 8
5. localloc 4
   label L6
6. str "%d %f %d %d\n"
7. local 1
8. @i 7
9. local 0
10. @f 9
11. param 0
12. @i 11
13. param 1
14. @i 13
15. argi 6
16. argi 8
17. argf 10
18. argi 12
19. argi 14
20. global printf
21. fi 20 5
   label L22
22. fend
```

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

fname  
fhead  
ftail  
id  
string  
op1  
exprs  
call  
m

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

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

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

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