

**COSC 461/561**  
csem - semantic routines for C programs

*csem* reads a C program (actually a subset of C) from its standard input and compiles it into LLVM intermediate representation (IR) on its standard output. It should support the following program constructs and statements in the C programming language:

program construct	example
<i>variable declaration</i>	<code>int foo;</code>
<i>array declaration</i>	<code>int foo[20];</code>
<i>function definition</i>	<code>double max(double x, double y);</code>
<i>function call</i>	<code>x = max(y, z);</code>
<i>label definition</i>	<code>LOOP1:</code>
<i>goto statement</i>	<code>goto LOOP1;</code>
<i>if statement</i>	<code>if ( x &lt; y ) { x = x + 1; }</code>
<i>if-else statement</i>	<code>if ( x &lt; y ) { x = x + 1; } else {y = y + 1;}</code>
<i>while statement</i>	<code>while ( x &lt; y ) { x = x + 1; }</code>
<i>do-while statement</i>	<code>do { x = x + 1; } while ( x &lt; y );</code>
<i>for statement</i>	<code>for ( x = 0; x &lt; 10; x++ ) { y = y + 1; }</code>
<i>continue statement</i>	<code>continue;</code>
<i>break statement</i>	<code>break;</code>
<i>return statement</i>	<code>return x;</code>

You will also need to support arithmetic and conditional expressions as well as assignment statements with the following operators. The bitwise operators (`|`, `&`, `<<`, `>>`, `~`) and mod (`%`) only operate on integer types. Your implementation should assume that all other operators support integer and floating point types and make appropriate conversions between types when necessary.

operator	description
<code>==</code> <code>!=</code> <code>&lt;=</code> <code>&gt;=</code> <code>&lt;</code> <code>&gt;</code> <code>=</code> <code> </code> <code>&amp;</code> <code>&lt;&lt;</code> <code>&gt;&gt;</code> <code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>%</code>	binary operators
<code>~</code> <code>-</code>	unary operators (inversion, negation)
<code>[ ]</code>	index into array

Your assignment is to write the semantic actions for the *csem* program to produce the appropriate LLVM IR. The following files, which should be downloaded from the course website (on Canvas), will comprise part of your program:

<code>makefile</code>	- csem makefile
<code>src/cgram.y</code>	- yacc grammar for subset of C
<code>src/main.cpp</code>	- main entry point for csem
<code>src/scan.c</code>	- lexical analyzer
<code>src/sem.cpp</code>	- semantic action routines
<code>src/semutil.c</code>	- utility routines for the semantic actions
<code>src/sym.c</code>	- symbol table management
<code>include/cc.h</code>	- defines several macros and globals for csem
<code>include/scan.h</code>	- defines prototypes for routines in scan.c
<code>include/sem.h</code>	- defines prototypes for routines in sem.cpp
<code>include/semutil.h</code>	- defines prototypes for routines in semutil.c
<code>include/sym.h</code>	- defines prototypes for routines in sym.c

The makefile will create an executable called `csem` in the current directory. For your assignment, you will fill in the definitions for the semantic action routines. This `sem.cpp` file contains stubs for each of the semantic action routines you will need to implement. While I have provided you access to the other `*.c`, `*.h`, and `*.cpp` files, you should not need to modify them. The yacc file `cgram.y` contains the entire grammar specification for the source language and also should not be modified. You can make additional functions and data structures in `sem.cpp` to abstract common operations and complete your implementation. When making your executable, refer to the makefile provided, which uses the other source files to produce the `csem` executable.

We have also included the executable file `ref_csem` that contains our working implementation. The executable was built on and will work on all of the hydra machines at UTK. The makefile also includes several targets to facilitate building and testing example input programs with different compilers. The following provides a brief explanation of the capabilities included in the makefile:

- To build the `csem` executable from source, simply type `make`:

```
> make
```

- To build an example input with `csem`, use `csem_build`:

```
> make csem_build INPUT=input4
```

This command will build the program `inputs/input4.c` with `csem`. The output files from the build will be written to the test folder. The executable file corresponding to the built input program will be in the top-level of your assignment directory and will be named `csem.exe`.

- If your implementation does not yet generate correct LLVM IR, you can use the `csem_ll` target to stop compilation after running `csem`:

```
> make csem_ll INPUT=input4
```

After running this command, the LLVM IR generated by `csem` will be in `test/csem_input4.ll`.

- To build an example input with `ref_csem`, use `ref_build`:

```
> make ref_build INPUT=input4
```

The output files from this build will also be written to the test folder. The built executable file will be named `ref_csem.exe`.

- To ensure your compiler is working correctly with a given input, use the `diff` target. This command will build the given input with both `ref_csem` as well as with `csem`, and will then run the resulting programs and compare their outputs with the Linux `diff` utility:

```
> make diff INPUT=input4
```

- To clean out the built program files in the test folder, use `test_clean`:

```
> make test_clean
```

- To clean out everything, including the built files associated with `csem`, use `clean`:

```
> make clean
```

You will also submit a project report, similar to the reports you submitted for previous projects. Specifically, you will submit a short (one to two page, single-spaced) document that describes:

1. (in your own words) the program you set out to write,
2. your approach (i.e. design and relevant implementation details) for writing this program,
3. how you debugged and tested your solution, and
4. any issues you had in completing the assignment.

You should upload a gzipped tar file (created with `tar cvzf ...`) with your source files and a pdf of your report to the Canvas course website before midnight on the assignment due date. Partial credits will be given for incomplete efforts. However, a program that does not compile or run will get 0 points. Point breakdown is below:

• backpatching	(10)
• loops, if statements	(10)
• conditional expressions	(10)
• type casting	(10)
• function call / return	(5)
• continue / break	(5)
• goto / labels	(5)
• logical expressions	(5)
• unary operators	(5)
• bitwise operators	(5)
• binary operators	(5)
• relational operators	(5)
• set expressions	(5)
• string code generation	(2)
• array indexing	(3)
• project report	(10)

Example 1 input (input2.c):

```
double m[6];

int scale(double x) {
    int i;

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

int main() {
    int i;
    double x;

    i = 0;
    while (i < 6) {
        m[i] = i;
        i = i + 1;
    }

    print("%5.1f %5.1f %5.1f\n", m[0], m[2], m[5]);

    x = 5;
    scale (x);

    print("%5.1f %5.1f %5.1f", m[0], m[2], m[5]);

    return 0;
}
```

Example 1 output:

```
; ModuleID = '<stdin>'
source_filename = "<stdin>"
```

```
@0 = private unnamed_addr constant [19 x i8] c"%d %3.2f %d %3.2f\0A\00", align 1
```

```
declare i32 @print(ptr, ...)
```

```
define i32 @main() {
    %a = alloca i32, align 4
    %c = alloca i32, align 4
    %b = alloca double, align 8
    %d = alloca double, align 8
    store i32 3, ptr %a, align 4
    store double 4.000000e+00, ptr %b, align 8
    store i32 5, ptr %c, align 4
    store double 6.000000e+00, ptr %d, align 8
```

```
%1 = load i32, ptr %a, align 4
%2 = load double, ptr %b, align 8
%3 = load i32, ptr %c, align 4
%4 = load double, ptr %d, align 8
%5 = call i32 @print(ptr @0, i32 %1, double %2, i32 %3, double %4)
ret i32 0
}
```

Example 2 input (input5.c):

```
double m[6];

int scale(double x) {
    int i;

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

int main() {
    int i;
    double x;

    i = 0;
    while (i < 6) {
        m[i] = i;
        i = i + 1;
    }

    print("%5.1f %5.1f %5.1f\n", m[0], m[2], m[5]);

    x = 5;
    scale (x);

    print("%5.1f %5.1f %5.1f", m[0], m[2], m[5]);

    return 0;
}
```

Example 2 output:

```
; ModuleID = '<stdin>'
source_filename = "<stdin>"
```

```
@m = global [6 x double] zeroinitializer
@0 = private unnamed_addr constant [19 x i8] c"%5.1f %5.1f %5.1f\0A\00", align 1
@1 = private unnamed_addr constant [18 x i8] c"%5.1f %5.1f %5.1f\00", align 1
```

```
declare i32 @print(ptr, ...)
```

```
define internal i32 @scale(double %x) {
    %x1 = alloca double, align 8
    store double %x, ptr %x1, align 8
    %i = alloca i32, align 4
    %1 = load double, ptr %x1, align 8
    %2 = fcmp oeq double %1, 0.000000e+00
    br i1 %2, label %L0, label %L1
```

```

L0:                                     ; preds = %0
    ret i32 0

L1:                                     ; preds = %0
    store i32 0, ptr %i, align 4
    br label %L2

L2:                                     ; preds = %L3, %L1
    %3 = load i32, ptr %i, align 4
    %4 = icmp slt i32 %3, 6
    br i1 %4, label %L4, label %L5

L3:                                     ; preds = %L4
    %5 = load i32, ptr %i, align 4
    %6 = add i32 %5, 1
    store i32 %6, ptr %i, align 4
    br label %L2

L4:                                     ; preds = %L2
    %7 = load i32, ptr %i, align 4
    %8 = getelementptr double, ptr @m, i32 %7
    %9 = load double, ptr %x1, align 8
    %10 = load double, ptr %8, align 8
    %11 = fmul double %10, %9
    store double %11, ptr %8, align 8
    br label %L3

L5:                                     ; preds = %L2
    ret i32 1
}

define i32 @main() {
    %i = alloca i32, align 4
    %x = alloca double, align 8
    store i32 0, ptr %i, align 4
    br label %L6

L6:                                     ; preds = %L7, %0
    %1 = load i32, ptr %i, align 4
    %2 = icmp slt i32 %1, 6
    br i1 %2, label %L7, label %L8

L7:                                     ; preds = %L6
    %3 = load i32, ptr %i, align 4
    %4 = getelementptr double, ptr @m, i32 %3
    %5 = load i32, ptr %i, align 4
    %6 = sitofp i32 %5 to double
    store double %6, ptr %4, align 8
    %7 = load i32, ptr %i, align 4
    %8 = add i32 %7, 1

```

```
store i32 %8, ptr %i, align 4  
br label %L6
```

```
L8:                                     ; preds = %L6  
%9 = load double, ptr @m, align 8  
%10 = load double, ptr getelementptr (double, ptr @m, i32 2), align 8  
%11 = load double, ptr getelementptr (double, ptr @m, i32 5), align 8  
%12 = call i32 (ptr, ...) @print(ptr @0, double %9, double %10, double %11)  
store double 5.000000e+00, ptr %x, align 8  
%13 = load double, ptr %x, align 8  
%14 = call i32 @scale(double %13)  
%15 = load double, ptr @m, align 8  
%16 = load double, ptr getelementptr (double, ptr @m, i32 2), align 8  
%17 = load double, ptr getelementptr (double, ptr @m, i32 5), align 8  
%18 = call i32 (ptr, ...) @print(ptr @1, double %15, double %16, double %17)  
ret i32 0  
}
```