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**
variable declaration	`int foo;`
array declaration	`int foo[20];`
function definition	`double max(double x, double y);`
function call	`x = max(y, z);`
label definition	`LOOP1:`
goto statement	`goto LOOP1;`
if statement	`if ( x < y ) { x = x + 1; }`
if-else statement	`if ( x < y ) { x = x + 1; } else {y = y + 1;}`
while statement	`while ( x < y ) { x = x + 1; }`
do-while statement	`do { x = x + 1; } while ( x < y );`
for statement	`for ( x = 0; x < 10; x++ ) { y = y + 1; }`
continue statement	`continue;`
break statement	`break;`
return statement	`return x;`

You will also need to support arithmetic and conditional expressions as well as assignment statements with the following operators. The bitwise operators (|, $\wedge$, &, <<, >>, $\sim$) 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**
== != <= >= < > = | $\wedge$ << >> + − * / %	binary operators
$\sim$ −	unary operators (inversion, negation)
[ ]	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 are part of the starter directory, will comprise part of your program:

`makefile`	- csem makefile
`src/parse.c`	- recursive descent parser for subset of C
`src/sym.c`	- symbol table management
`src/semutil.c`	- utitity routines for the semantic actions
`src/main.cpp`	- main entry point for csem
`src/sem.cpp`	- semantic action routines
`include/cc.h`	- defines several macros and globals for csem
`include/scan.h`	- defines prototypes for routines in scan.c
`include/parse.h`	- defines prototypes for routines in parse.c
`include/semutil.h`	- defines prototypes for routines in semutil.c
`include/sym.h`	- defines prototypes for routines in sym.c
`include/sem.h`	- defines prototypes for routines in sem.cpp

The makefile will create an executable called `csem` in the current directory. For your assignment, you will fill in the defintions for the semantic action routines in sem.cpp. This sem.cpp file contains stubs for each of the semantic action routines you will need to implement. You are not allowed to modify any files in the starter directory other than sem.cpp. 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_exe`:

  ```
  > make csem_exe 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_csem_exe`:

  ```
  > make ref_csem_exe 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.sh` script. This script builds 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:

```
> ./diff.sh input4
```

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

```
> make clean
```

You should upload your completed sem.cpp file 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)
• goto / labels	(10)
• function call / return	(5)
• continue / break	(5)
• logical expressions	(5)
• unary operators	(5)
• bitwise operators	(5)
• binary operators	(5)
• relational operators	(5)
• assign expressions	(5)
• string code generation	(5)
• array indexing	(5)

Example 1 input (input2.c):

```
int main()
{
  int a, c;
  double b, d;

  a = 3;
  b = 4;
  c = 5;
  d = 6;

  print("%d %3.2f %d %3.2f\n", a, b, c, d);
  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(i32 %a, i32 %c) {
  %a1 = alloca i32, align 4
  store i32 %a, ptr %a1, align 4
  %c2 = alloca i32, align 4
  store i32 %c, ptr %c2, align 4
  %b = alloca double, align 8
  %d = alloca double, align 8
  store i32 3, ptr %a1, align 4
  store double 4.000000e+00, ptr %b, align 8
  store i32 5, ptr %c2, align 4
  store double 6.000000e+00, ptr %d, align 8
  %1 = load i32, ptr %a1, align 4
  %2 = load double, ptr %b, align 8
  %3 = load i32, ptr %c2, align 4
  %4 = load double, ptr %d, align 8
  %5 = call i32 (ptr, ...) @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, i32 %i) {
  %x1 = alloca double, align 8
  store double %x, ptr %x1, align 8
  %i2 = alloca i32, align 4
  store i32 %i, ptr %i2, 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 %i2, align 4
  br label %L2

L2:                                               ; preds = %L3, %L1
  %3 = load i32, ptr %i2, align 4
  %4 = icmp slt i32 %3, 6
  br i1 %4, label %L4, label %L5

L3:                                               ; preds = %L4
  %5 = load i32, ptr %i2, align 4
  %6 = add i32 %5, 1
  store i32 %6, ptr %i2, align 4
  br label %L2

L4:                                               ; preds = %L2
  %7 = load i32, ptr %i2, 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(i32 %i) {
  %i1 = alloca i32, align 4
  store i32 %i, ptr %i1, align 4
  %x = alloca double, align 8
  store i32 0, ptr %i1, align 4
  br label %L6

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

L7:                                               ; preds = %L6
  %3 = load i32, ptr %i1, align 4
  %4 = getelementptr double, ptr @m, i32 %3
  %5 = load i32, ptr %i1, align 4
  %6 = sitofp i32 %5 to double
  store double %6, ptr %4, align 8
```

```
  %7 = load i32, ptr %i1, align 4
  %8 = add i32 %7, 1
  store i32 %8, ptr %i1, 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
}
```