COP-3402 Systems Software
10.17 Thu

We are doing type checking just making sure that symbols have been declared before they're used and they're not multiple-defined.

Code Generation for Variables:
Conceptually, we will be adding a few constructs for variables in our language after project0, such as 'read', 'assign', 'identifier', 'declarations'. 'read' takes a number from the standard input as ASCII coded number, and returns an integer.

There is no way in the given grammar of Project2 to have negative identifiers. If we have a number that has a minus sign before it, that will be represented in 2's complement as a negative number in the representation.

Variables are associated with memory locations on the stack in our case. Stack grows down from the top of memory addresses, whereas heap grows up from the bottom. The way this works in LLVM is that LLVM has an instruction called 'alloca', telling LLVM to make space on the stack for our memory address. alloca returns and address, in LLVM its data type is a pointer, i32. We need to keep track of which memory address corresponds to which variable by symbol table. We have a function called expression that takes the characters in file, and returns a temporary variable that holds the value of that expression.

'read' statement is like 'print', but the difference is it takes input and store the value.
We also need to update 'factor' to support 'identifier' by emitting a load the value of variable into new temporary register.

Declarations allocate stack space. Pseudocode for declaration:

```
declaration():
  assert consume() == 'int'
  ident = consume()
  assert consume == ';'
  if (contains(ident))
error()
  result = newtemp()
  emit result "= alloca"
  put(ident, result)
```

Declaring a variable in SimpleC is generating LLVM instructions that call 'alloca' and save that in some temporary register. That has the memory address of our variable, we save that variable/register name in our symbol table. We map the identifier that we parsed, which is the name of our variable, to the LLVM register name.

'read' stores a value at variable's address. Pseudocode for read:

```
read():
  assert consume() == 'read'
  ident = consume()
  assert consume == ';'
  addr = lookup(ident) or error()
  result = newtemp()
  emit result " = read_integer()"
  emit "store " result ", " addr
```

‘assign’ stores a value at variable’s address. First evaluate the right-hand side expression; this returns the thing to store. Then lookup address and emit store. Compose expression() (from project1) with assignment function. Pseudocode for assign:

```
assign():
  ident = consume();
  assert consume() == '='
  result = expression()
  assert consume() == ';'
  addr = lookup(ident) or error()
  emit "store " result ", " addr
```

‘factor’ gets a value from variable’s address. factor returns a constant int value or a temporary register. We have a variable in our case, so can we return a constant int for the variable? Yes, we can have a load instruction from the address of the variable, make a new temporary register to store that value in. In LLVM, meaning load the value from the memory and put it into a temporary register.

```
factor():
  // ... (the rest of the function)
  elif (next is IDENTIFER):
    ident = consume()
    addr = lookup(ident) or error()
    result = newtemp()
    emit result " = load " addr
    return result
  // ... (the rest of the function)
```

Expression Parser Simplification:
Replace the right-recursive grammar (E',T') with a loop

E -> T E'
E' -> + T E' | - T E | ε

E -> T (+ T)* | T (- T)*

```
expression():
  left = term()
  while (next is PLUS or MINUS):
    op = consume()
    right = term()
    result = newtemp()
    emit result " = " opname(op) " " left ", " right
    left = result
  return left
```

Let's say, we declare int x, and print x, what happens? In C specification, this is called undefined behavior. It will just return garbage value as it is compiler's choice.

Ex:

```
// no constant folding:
// gcc -O0 -o constant.noopt.s -S constant.c
// with constant folding:
// gcc -O3 -o constant.opt.s -S constant.c


int main() {

                int x = 343;

                printf("%d\n", x * 78510);

                return 0;

               }
```

Opt:

```
leaq    .LC0(%rip), %rdi
subq    $8, %rsp
.cfi_def_cfa_offset 16
movl    $26928930, %esi
xorl    %eax, %eax
call    printf@PLT
xorl    %eax, %eax
addq    $8, %rsp
.cfi_def_cfa_offset 8
ret
```

No opt:

```
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
subq    $16, %rsp
movl    $343, -4(%rbp)
movl    -4(%rbp), %eax
imull   $78510, %eax, %eax
movl    %eax, %esi
leaq    .LC0(%rip), %rdi
movl    $0, %eax
call    printf@PLT
movl    $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
```

Demo - Code Generation for Variables:

int x: First, make sure to do a lookup for our error checking and then we add that value to the symbol table (put). Next, we emit an alloca instruction.

Same thing for ‘int y’.

read x; First lookup. Then emit read_integer, and emit store.