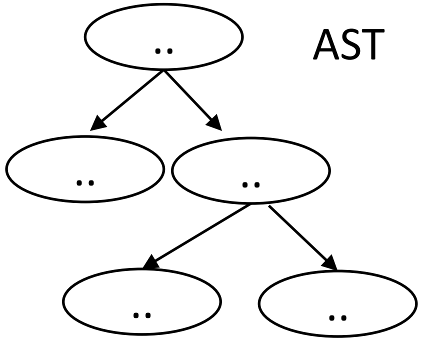
CSE110A: Compilers

May 19, 2023

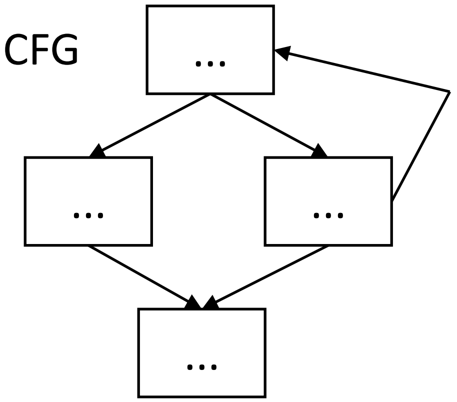
AST



Topics:

- *Finishing up translation into 3 address code*
- *homework review*

CFG



3 address code

```
store i32 0, ptr %2
%3 = load i32, ptr %1
%4 = add nsw i32 %3, 1,
store i32 %4, ptr %1
%5 = load i32, ptr %2
```

Announcements

- HW 4 is out:
  - Due on the 29th
  - Get started early; it is a big assignment!
  - Earlier office hours are less busy than office hours close to the deadline

- Midterm grades are out
  - Come see me in office hours if you want to review your test
  - We will go over the test on Monday

- HW 2 and HW 3 grades are on the way

Announcements

Schedule:

- Hopefully we will finish module 3 today
- Midterm review on Monday
- Moving to module 4 on Wednesday

Quiz

Quiz

How many virtual registers does the following expression need?

```
int a, x, y;

a = ((x + 1) * y - 1) / 2.0;
```

Discussion

- two ways to do this: First

```
a = ((x + 1) * y - 1) / 2.0f;
```

```
vr0 = x   + 1
vr1 = vr0 * y
vr2 = vr1 - 1
vr3 = int2float(vr2)
vr4 = vr3 / 2.0f
vr5 = float2int(vr4)
a = vr5
```

Are all of these necessary?

Assumptions about the IR?

Discussion

- two ways to do this: Second way: use Godbolt

```c
int foo_int(int x, int y) {
   return ((x + 1) * y - 1) / 2.0f;
}
```

use clang with flag: `-emit-llvm`

```
%5 = load i32, ptr %3, align 4, !dbg !20
%6 = add nsw i32 %5, 1, !dbg !21
%7 = load i32, ptr %4, align 4, !dbg !22
%8 = mul nsw i32 %6, %7, !dbg !23
%9 = sub nsw i32 %8, 1, !dbg !24
%10 = sitofp i32 %9 to float, !dbg !25
%11 = fdiv float %10, 2.000000e+00, !dbg !26
%12 = fptosi float %11 to i32, !dbg !25
```

Discussion

- two ways to do this: Second way: use Godbolt

```c
int foo_int(int x, int y) {
   return ((x + 1) * y - 1) / 2.0f;
}
```

use clang with flag: `-emit-llvm`

```
%5 = load i32, ptr %3, align 4, !dbg !20
%6 = add nsw i32 %5, 1, !dbg !21
%7 = load i32, ptr %4, align 4, !dbg !22
%8 = mul nsw i32 %6, %7, !dbg !23
%9 = sub nsw i32 %8, 1, !dbg !24
%10 = sitofp i32 %9 to float, !dbg !25
%11 = fdiv float %10, 2.000000e+00, !dbg !26
%12 = fptosi float %11 to i32, !dbg !25
```

we probably wouldn't count loads for our purposes

Quiz

How many labels do you need for the following expression?

```
int x, y;
...
if (x==0){
...
} else if (y>1) {
...
}else {
...
}
```

```
int x, y;
...
if (x==0){
...
} else if (y>1) {
...
} else {
...
}
```

where do we need the labels?

```
int x, y;
...
if !(x == 0) goto elseif;
...
goto end;
elseif:
if !(y>1) goto else:
...
goto end;
else:
...
}
end:
```

where do we need the labels?

Quiz discussion

- What about Godbolt?

Quiz discussion

- Follow up question:
  - How would we extend our language to support “else if”?

Quiz

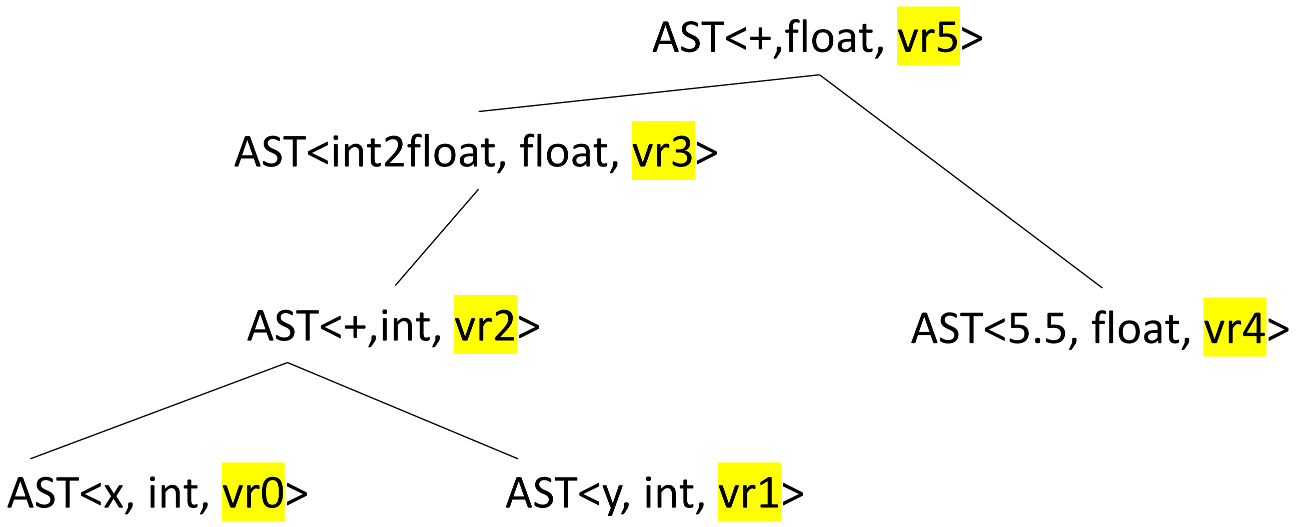
The number of virtual registers is equal to the number of nodes in the AST

- ○ True
- ○ False

Converting AST into Class-IR

```
int x;
int y;
float w;
w = x + y + 5.5
```

After type inference



We will start by adding a new member to each AST node:

A virtual register

Each node needs a distinct virtual register

Discussion

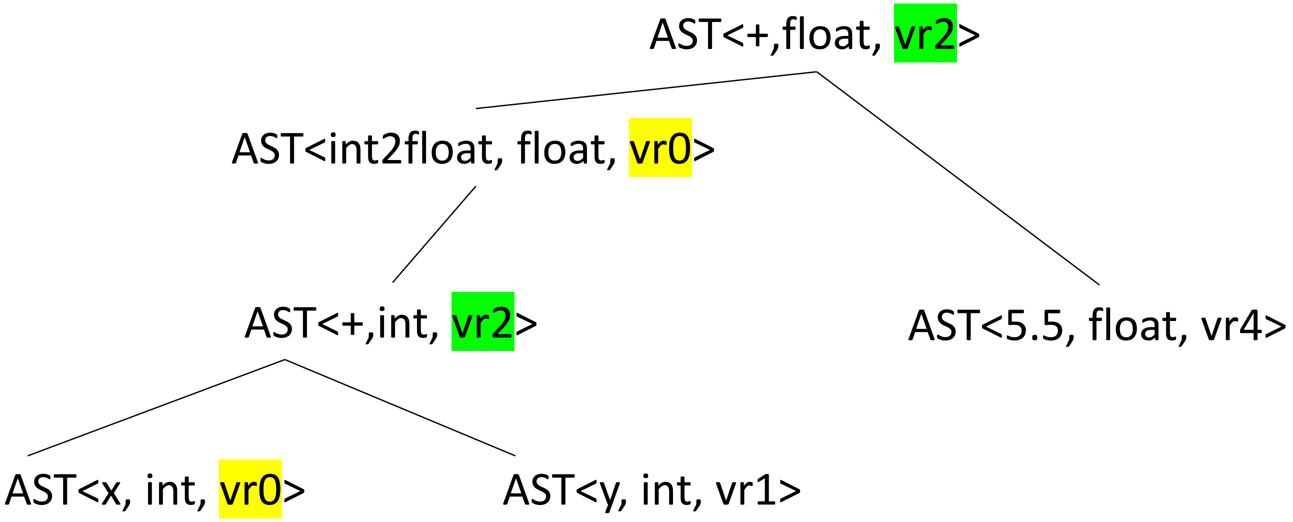
- The easiest (and most common way) is to allocate a virtual register for each node

- You might not need nodes for some variables or literal
  - depends on the IR and type system

- You could potentially re-use virtual registers, but typically this isn’t done at this point.

Converting AST into Class-IR

```
int x;
int y;
float w;
w = x + y + 5.5
```



potentially registers could be reused if they are not used again

Quiz

Discuss a few optimizations that you could imagine doing as you convert an AST into 3 address code

Quiz

Discuss a few optimizations that you could imagine doing as you convert an AST into 3 address code

Loop unrolling
computing constants (e.g., 5 + 6)

Review

- Class IR:
  - or ClassIeR

- Converting an AST into ClassIeR

Class-IR

Inputs/outputs (IO): 32-bit typed inputs

e.g.: `int x, int y, float z`

Program Variables (Variables): 32-bit untyped virtual register

given as `vrX` where `X` is an integer:

e.g. `vr0, vr1, vr2, vr3` …

we will assume input/output names are disjoint from virtual register names

Class-IR

binary operators:

```
dst = operation(op0, op1);
```

```
operations can be one of:
[add, sub, mult, div, eq, lt]
```

```
each operation is followed by an i or f, which
specifies how the bits in the registers are
interpreted
```

Class-IR

binary operators:

```
dst = operation(op0, op1);
```

```
operations can be one of:
[add, sub, mult, div, eq, lt]
```

all of dst, op0, and op1 must be untyped virtual registers.

Class-IR

binary operators:

```
dst = operation(op0, op1);
```

```
Examples:

vr0 = addi(vr1, vr2);
vr3 = subf(vr4, vr5);

x = multf(vr0, vr1); not allowed!
vr0 = addi(vr1, 1);  not allowed!
```

We'll talk about how to do this using other instructions

Class-IR

Control flow

`branch(label);`

- branches unconditionally to the label

`bne(op0, op1, label)`

- if op0 is not equal to op1 then branch to label
- operands must be virtual registers!

`beq(op0, op1, label)`

- Same as bne except it is for equal

Class-IR

Assignment

```
vr0 = vr1

one virtual register can be assigned to another
```

Class-IR

Assignment

```
vr0 = vr1

one virtual register can be assigned to another

Examples:
vr0 = 1; not allowed
vr1 = x; not allowed
```

Class-IR

```
unary get untyped register
dst = operation(op0);

operations are: [int2vr, float2vr]

Example:
```

Given IO: int x and float y

```
vr1 = int2vr(x);
vr2 = float2vr(2.0);
```

Class-IR

```
unary get typed data
dst = operation(op0);

operations are: [vr2int, vr2float]

Example:
```

Given IO: int x and float y

```
x = vr2int(vr1);
y = vr2float(vr3);
```

Compiler pragmatics

- New terminology I learned recently:
  - Implementation details

- We need to talk about different ID types (IO, VRs)
- We need to talk about scopes

Class-IR

unary conversion operators:

```
dst = operation(op0);
```

operations can be one of:

[vr_int2float, vr_float2int]

converts the bits in a virtual register from one type to another. *op0 and dst must be a virtual register!*

Class-IR

unary conversion operators:

```
dst = operation(op0);

Examples:

vr0 = vr_int2float(vr1);
vr2 = vr_float2int(1.0); not allowed!
```

Two different ID nodes

Gets compiled into an untyped virtual register

```
class ASTVarIDNode(ASTLeafNode):
    def __init__(self, value, value_type):
        super().__init__(value)
        self.node_type = value_type
```

Gets compiled into a typed IO variable

```
class ASTIOIDNode(ASTLeafNode):
    def __init__(self, value, value_type):
        super().__init__(value)
        self.node_type = value_type
```

Two different ID nodes

What we are compiling

```
void test4(float &x) {
  int i;
  for (i = 0; i < 100; i = i + 1) {
    x = i;
  }
}
```

Class-IR

What we are compiling

```
void test4(float &x) {
  int i;
  for (i = 0; i < 10; i = i + 1) {
    x = i;
  }
}
```

IO variables

program variables

```
int main() {
  int a = 0;
  test1(a);
  cout << a << endl;
  return 0;
}
```

What does this print?

What we are compiling

IO variables

```
void test4(float &x) {
  int i;
  for (i = 0; i < 100; i = i + 1) {
    x = i;
  }
}
```

program variables

Every time you access an IO variable, you need to convert it to a vr first using float2vr or int2vr

```
class ASTIOIDNode(ASTLeafNode):
...
    def three_addr_code(self):
        if self.node_type == Types.INT:
            return "%s = int2vr(%s);" % (self.vr, self.value)
        if self.node_type == Types.FLOAT:
            return "%s = float2vr(%s);" % (self.vr, self.value)
```

What we are compiling

IO variables

```
void test4(float &x) {
   int i;
   for (i = 0; i < 100; i = i + 1) {
      x = i;
   }
}
```

program variables

Every time you access a program variable, it does not need to be converted.

Because its value is a virtual register, you can even just use its value as its virtual register

```
class ASTVarIDNode(ASTLeafNode):
...

    def three_addr_code(self):
        return "%s = %s;" % (self.vr, self.value)
```

building an expression AST, we parse a unit at the base

```
unit := ID
      |  ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match.value
   data_type = # get type from symbol table
   eat("ID")
   return ASTIDNode(id_name, data_type)
}
```

Previously we had just one ID node

building an expression AST, we parse a unit at the base

```
unit := ID
     |   ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match.value
   data_type = # get type from symbol table
   eat("ID")
   return ASTIDNode(id_name, data_type)
}
```

building an expression AST, we parse a unit at the base

```
unit := ID
      |  ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match.value
   id_data = # get id_data from the symbol table
   eat("ID")
   return ASTIDNode(id_name, ...)
}
```

id_data should contain:

id_type*: IO or Var*

data_type*: int or float*

building an expression AST, we parse a unit at the base

```
unit := ID
      |  ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match.value
   id_data = # get id_data from the symbol table
   eat("ID")
   if (id_data.id_type == IO)
       return ASTIOIDNode(id_name, id_data.data_type)
   else
       return ASTVarIDNode(id_name, id_data.data_type)
}
```

id_data should contain:

id_type*: IO or Var*

data_type*: int or float*

Getting back to our statements:

```
statement := declaration_statement
           |  assignment_statement
           |  if_else_statement
           |  block_statement
           |  for_loop_statement
```

When we declare a variable, we need to mark it as a program variable in the symbol table

Getting back to our statements:

```
statement := declaration_statement
           |  assignment_statement
           |  if_else_statement
           |  block_statement
           |  for_loop_statement
```

We need to use symbol table data for something else. What?

Getting back to our statements:

```
statement := declaration_statement
           |   assignment_statement
           |   if_else_statement
           |   block_statement
           |   for_loop_statement
```

We need to use symbol table data for something else. What?

Scopes! Class IR has no {}s, so we need to manage scopes

Scopes

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

What does y hold?

Scopes

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

How can we get rid of the {}'s?

What does y hold?

Scopes

Let’s walk through it with a symbol table

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

Scopes

Let's walk through it with a symbol table

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

HT0

symbol table hash table stack

Scopes

rename

Let's walk through it with a symbol table

```
int x_0;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

make a new unique name for x

HT0

```
x: (INT, VAR, "x_0")
```

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

HT0

x: (INT, VAR, "x_0")

symbol table hash table stack

Scopes

rename

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

make a new unique name for y

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

Scopes

search

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

HT0

x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")

symbol table hash table stack

Scopes

```
int x_0;
int y_0;
x_0 = 5;
{
  int x;
  x = 6;
  y = x;
}
```

replace with new name

Let's walk through it with a symbol table

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x;
  x = 6;
  y = x;
}
```

new scope. Add x with a new name

HT1	x: (INT, VAR, "x_1")
HT0	x: (INT, VAR, "x_0") y: (INT, VAR, "y_0")

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x = 6;
  y = x;
}
```

new scope. Add x with a new name

HT1	x: (INT, VAR, "x_1")
HT0	x: (INT, VAR, "x_0") y: (INT, VAR, "y_0")

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x = 6;
  y = x;
}
```

lookup

new scope. Add x with a new name

HT1

x: (INT, VAR, "x_1")

HT0

x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y = x;
}
```

lookup

new scope. Add x with a new name

HT1	x: (INT, VAR, “x_1”)
HT0	x: (INT, VAR, “x_0”) y: (INT, VAR, “y_0”)

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y = x;
}
```

lookup

new scope. Add x with a new name

HT1

x: (INT, VAR, “x_1”)

HT0

x: (INT, VAR, “x_0”)
y: (INT, VAR, “y_0”)

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y_0 = x_1;
}
```

lookup

new scope. Add x with a new name

HT1	x: (INT, VAR, “x_1”)
HT0	x: (INT, VAR, “x_0”) y: (INT, VAR, “y_0”)

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y_0 = x_1;
}
```

new scope. Add x with a new name

HT1	x: (INT, VAR, "x_1")
HT0	x: (INT, VAR, "x_0") y: (INT, VAR, "y_0")

symbol table hash table stack

No more need for {}

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
int x_1;
x_1 = 6;
y_0 = x_1;
```

new scope. Add x with a new name

HT1

```
x: (INT, VAR, "x_1")
```

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

No more need for {}

symbol table hash table stack

Scopes

What happens with multiple scopes?

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
}
{
  int x;
  x = 1;
  y = x;
}
```

Class-IR

Remind ourselves what we are compiling

```
void test4(float &x) {
  int i;
  for (i = 0; i < 100; i = i + 1) {
    x = x + i;
  }
}
```

We only need new names for program variables, not for IO variables

building an expression AST, we parse a unit at the base

```
unit := ID
     |  ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match[1]
   id_data = # get id_data from the symbol table
   eat("ID")
   if (id_data.id_type == IO)
       return ASTIOIDNode(id_name, id_data.data_type)
   else
       return ASTVarIDNode(id_data.new_name, id_data.data_type)
}
```

id_data should contain:

***id_type**: IO or Var*

***data_type**: int or float*

***new_name**: new unique name*

Look at homework

See everyone on Monday

- Reviewing midterm