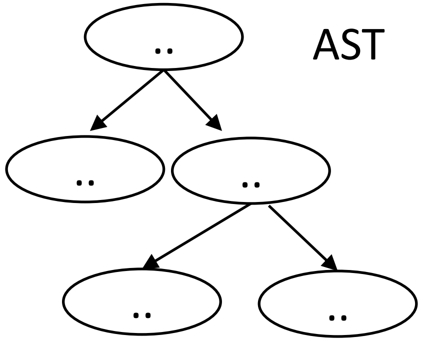
CSE110A: Compilers

May 13, 2024

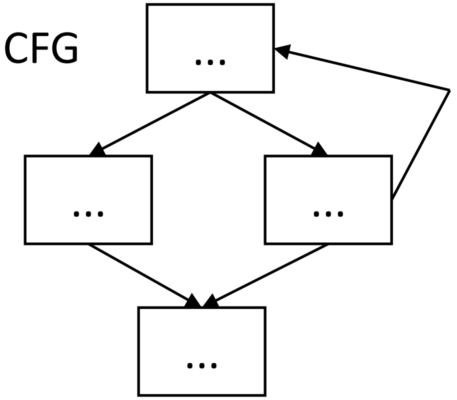
AST



Topics:

- *Module 3: Intermediate representations*
    - *Linear IRs*

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

- Homework 1 grades are out
  - You have until Monday to raise any issues
  - Planning on homework 2 grades out by this Friday
  - Grading midterm this Friday and hoping to get grades by end of tonight
  - Grading questions are done in-person in my (or TA) office hours starting next week
- Homework 3 was due on Friday
- Homework 4 is out
  - It is a longer assignment (2 weeks) Get started early!
  - Create an AST from your parser: Do type checking, Create linear code
  - The homework repo has a solution of HW 3.

Announcements

- Absences this week
  - I will be gone Wednesday and Friday
  - Friday will be midterm review.
    - This lecture will not be recorded and slides will not be provided
    - Attend in person
    - If you want to discuss your midterm, come after the review
  - Wednesday lecture will be canceled
    - Get started on HW 4
  - No office hours on thursday

Announcements

- Mentors are reporting that they have many slots available, please take advantage of them.

Quiz

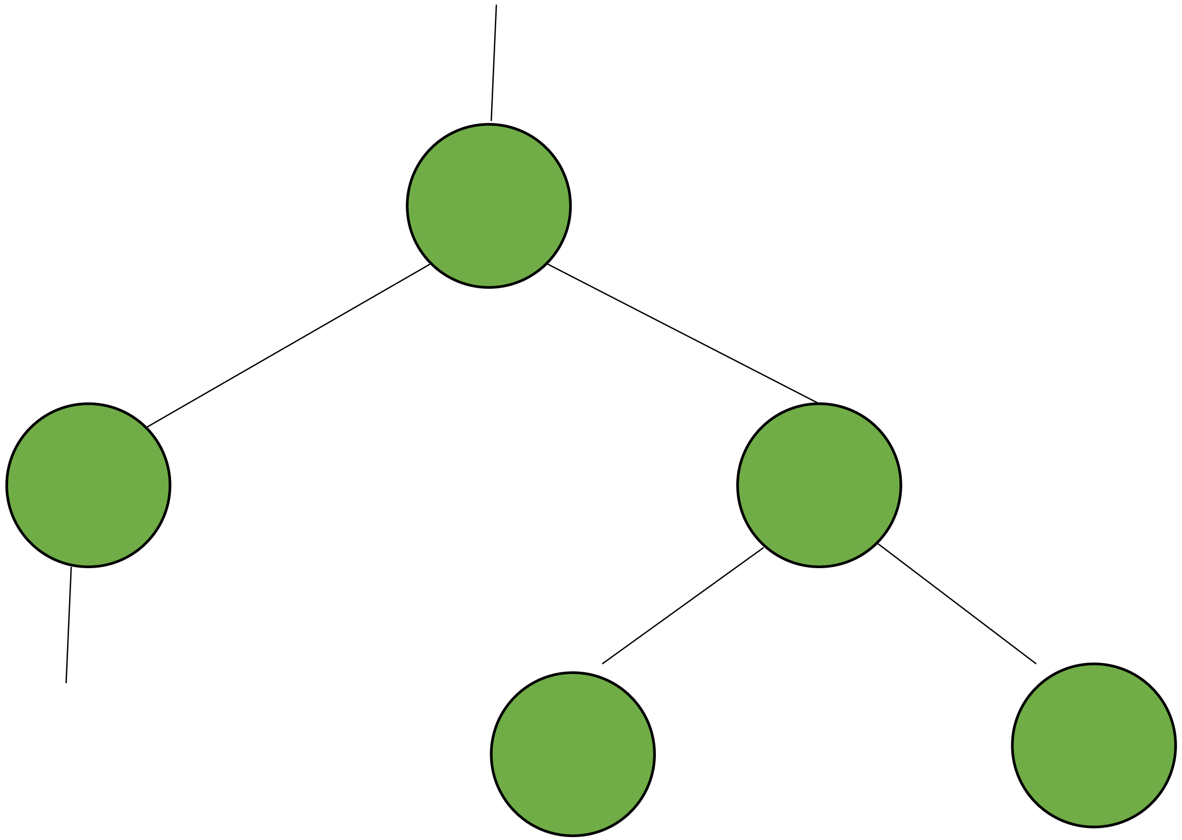
Quiz

We can infer the type of an expression using in-order traversal on the AST

- ○ True
- ○ False

Discussion

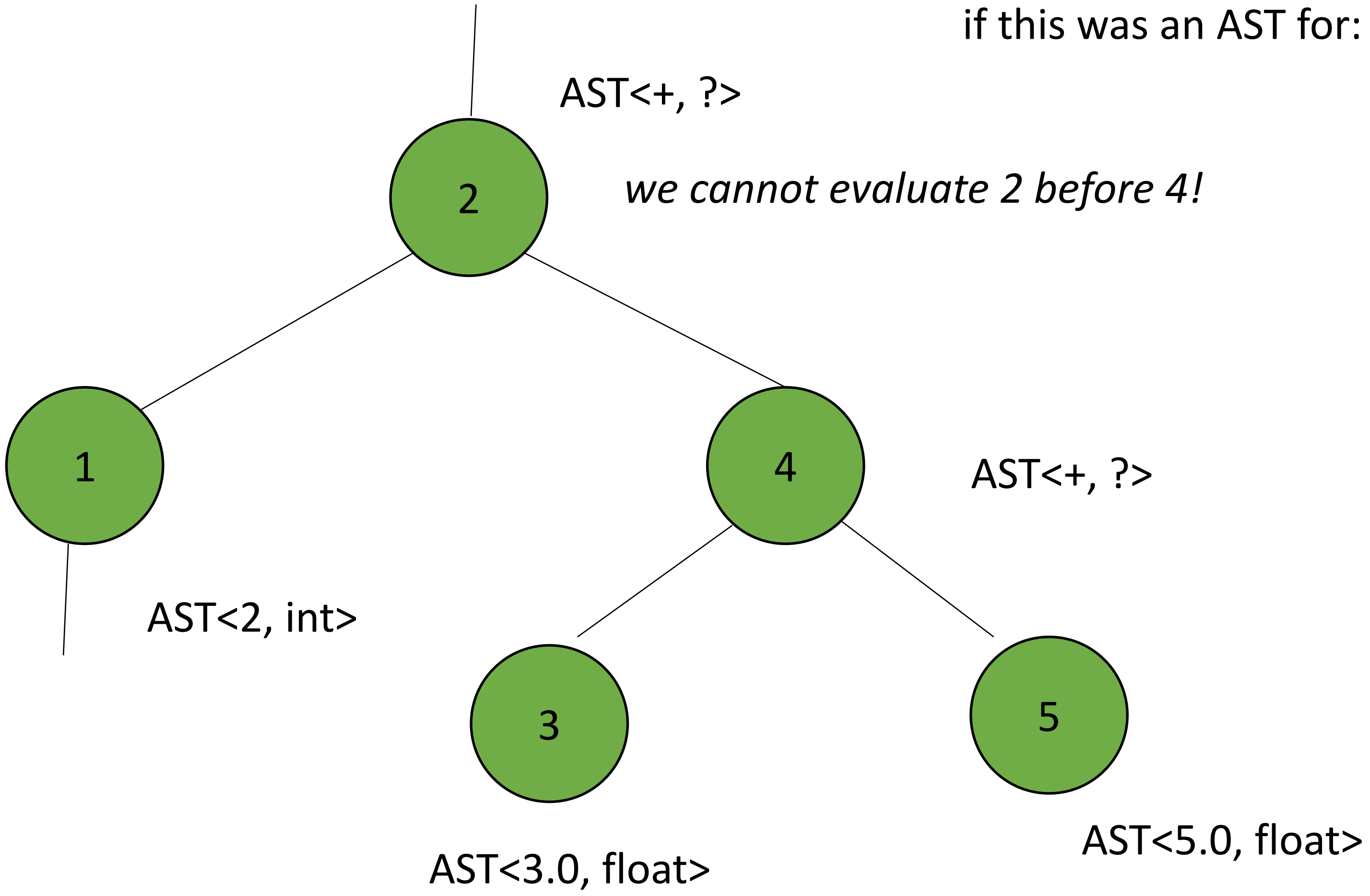
What is the in order traversal order?



Discussion

What is the in order traversal order?

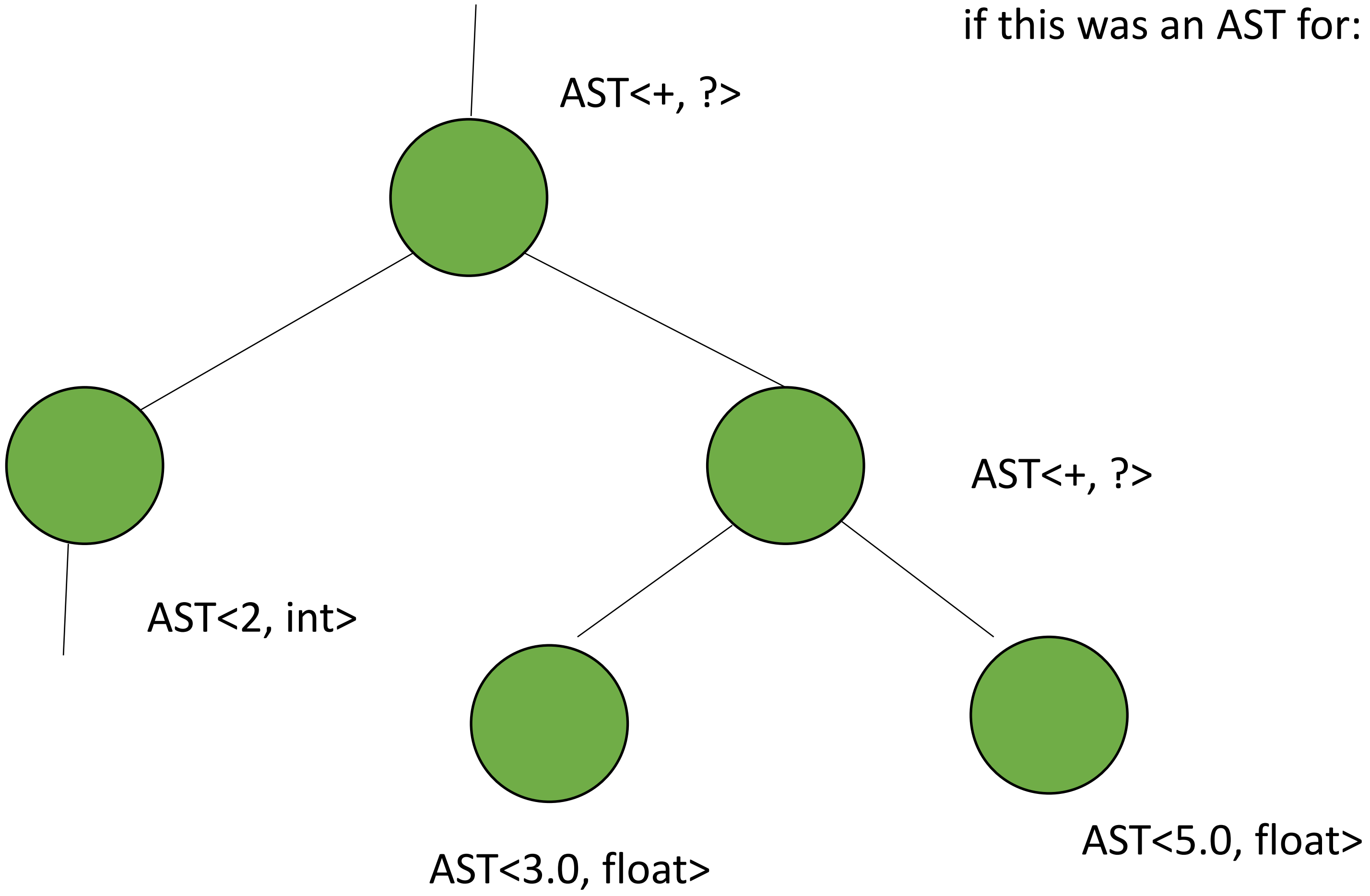
if this was an AST for: “2 + (3.0 + 5.0)”



Discussion

What is the post order traversal order?

if this was an AST for: “2 + (3.0 + 5.0)”

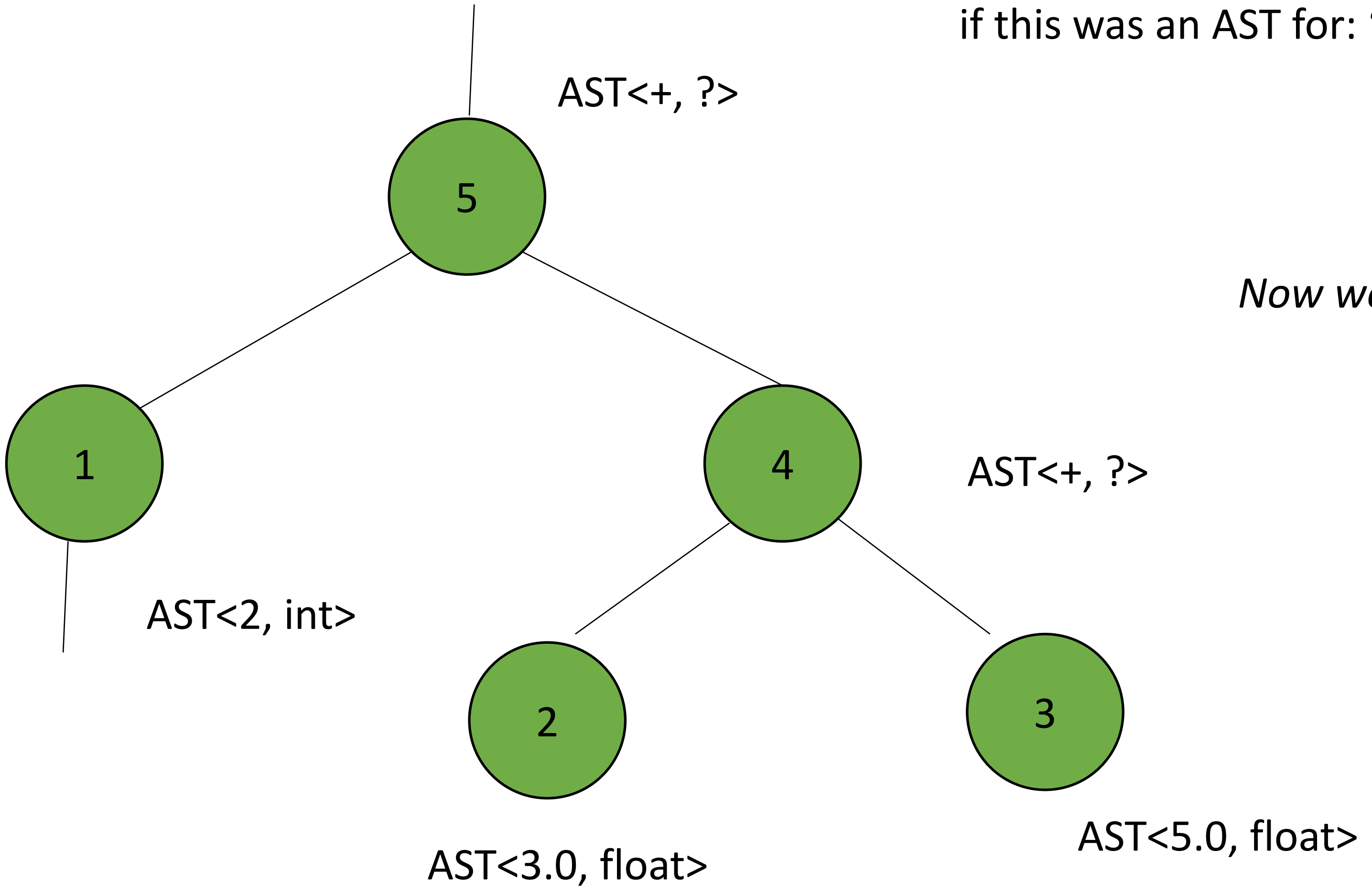


Discussion

What is the post order traversal order?

if this was an AST for: “2 + (3.0 + 5.0)”

Now we can do type inference

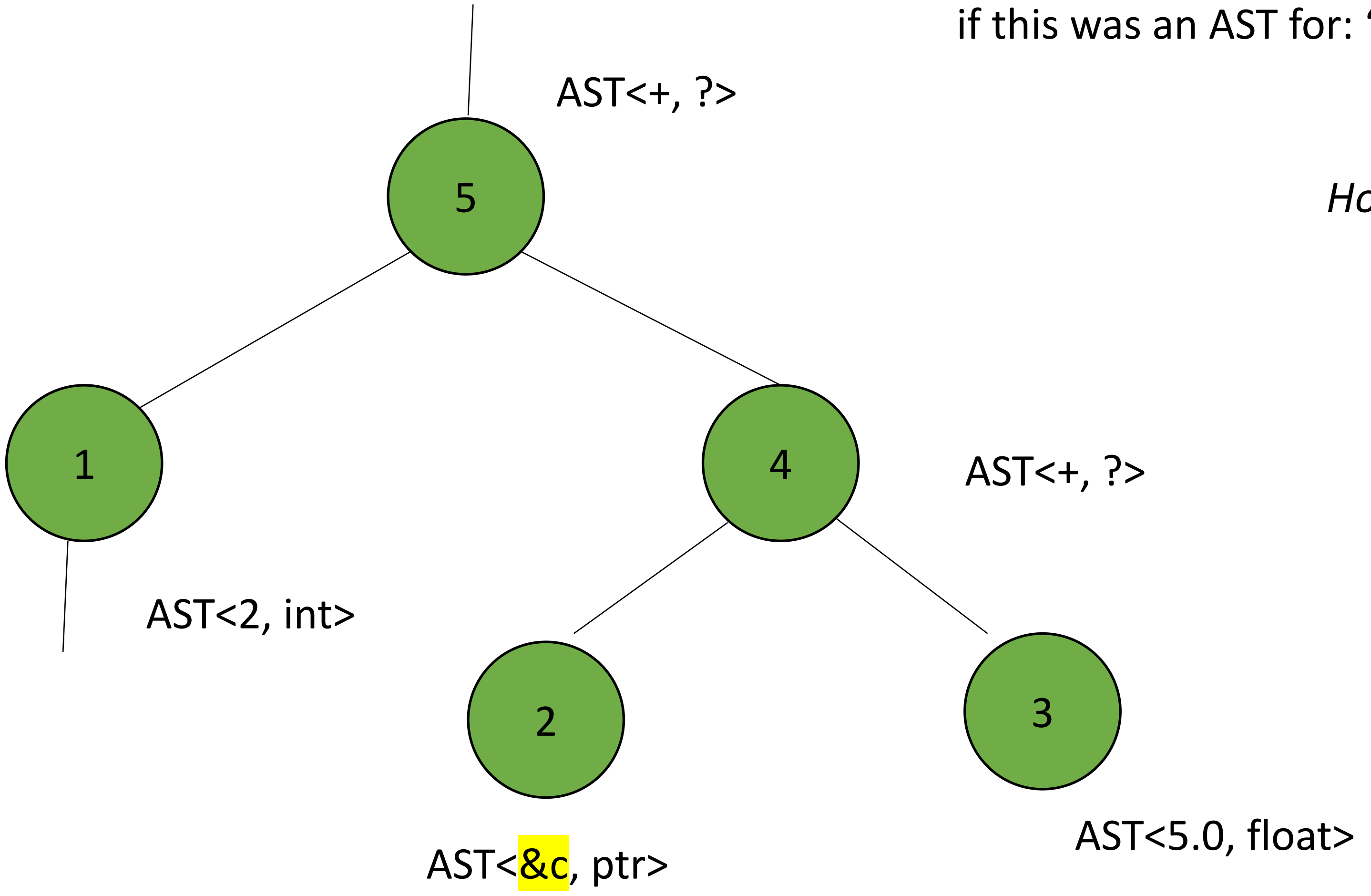


Discussion

What is the post order traversal order?

if this was an AST for: “2 + (&c + 5.0)”

How does this change things?



Quiz

Why do we use an AST as an IR? Why not directly go to a linear IR (or 3 address code)? Write a few sentences about this design choice.

Quiz

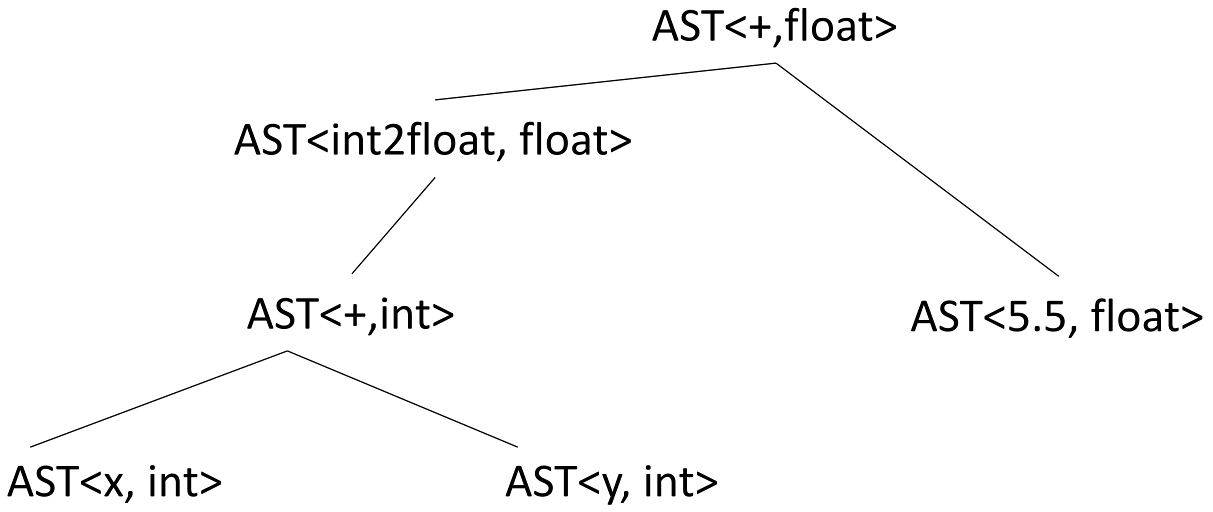
What are some ways that your favorite language lets you modify the type system? i.e., expand the type conversion tables.

Homework overview

Converting AST into Class-IR

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

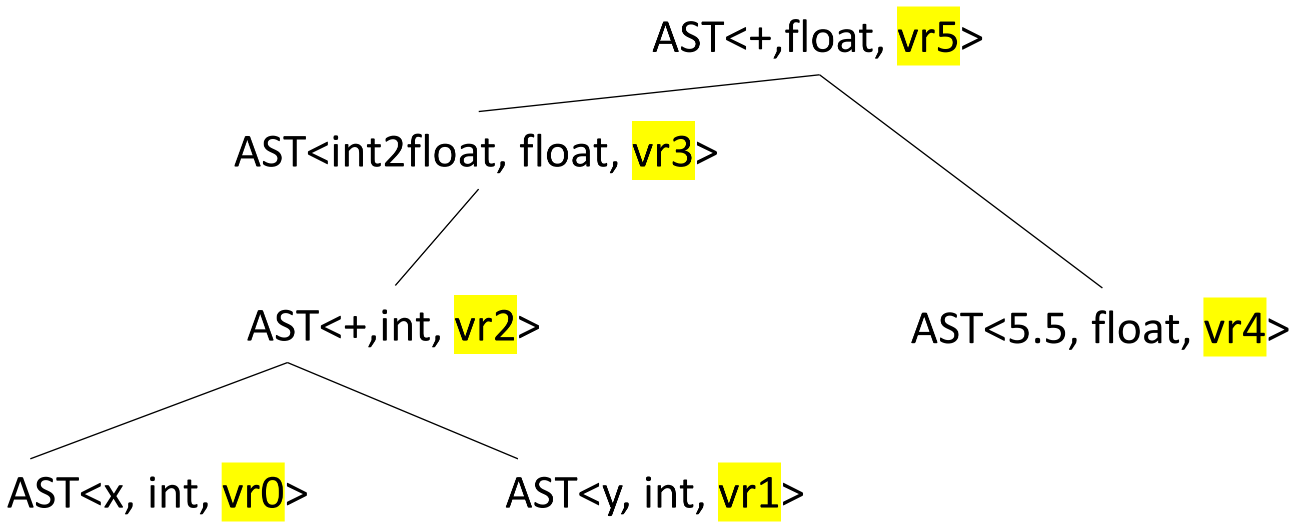
After type inference



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:

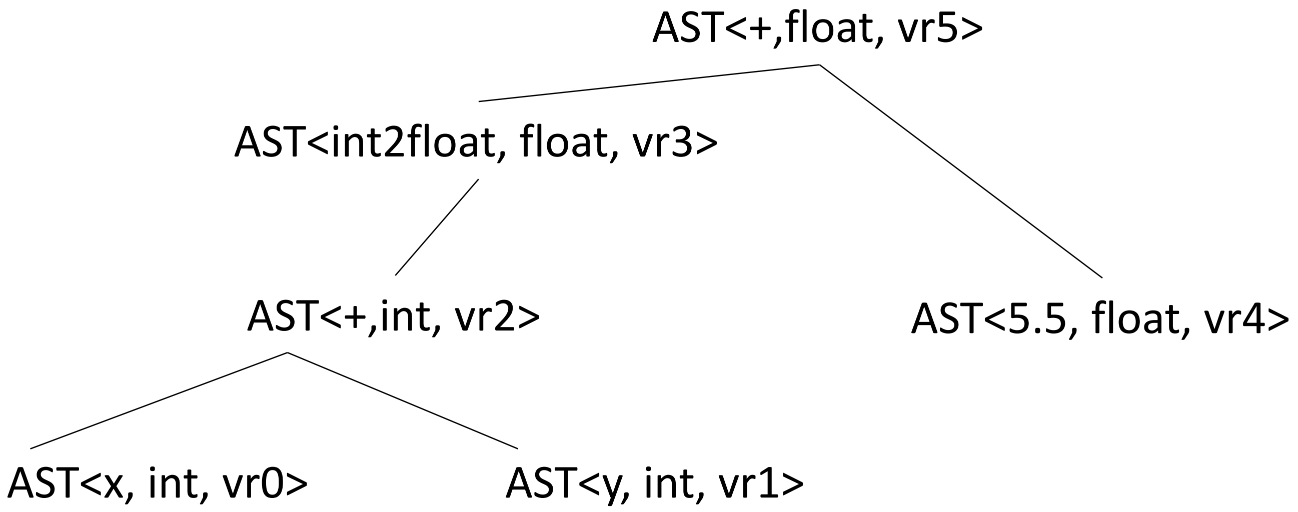
A virtual register

Each node needs a distinct virtual register

Converting AST into Class-IR

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

After type inference

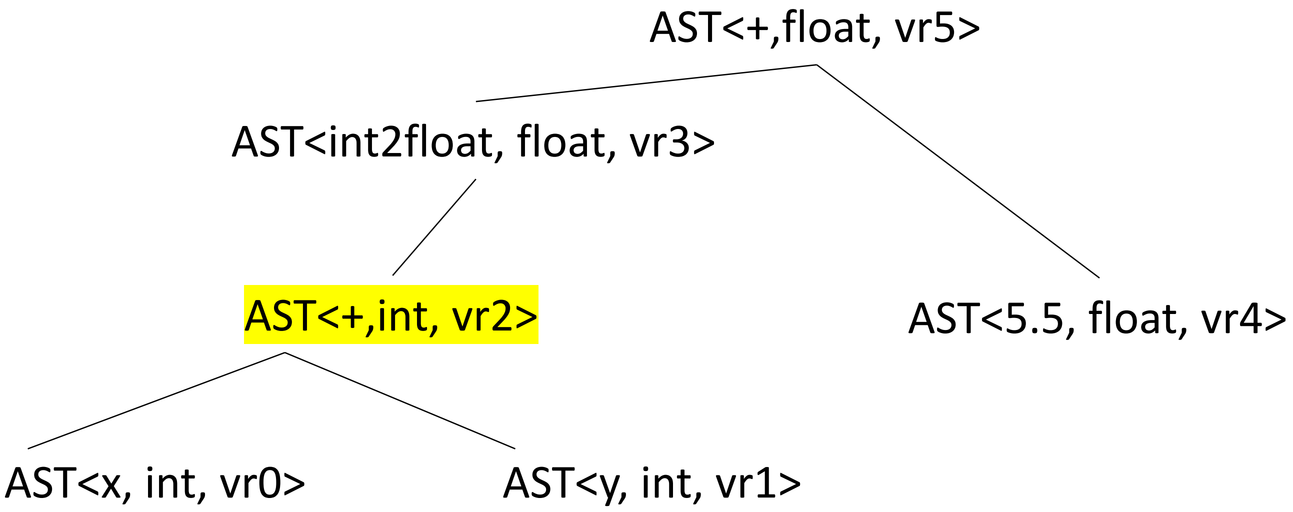


Next each AST node needs to know how to print a 3 address instruction

Converting AST into Class-IR

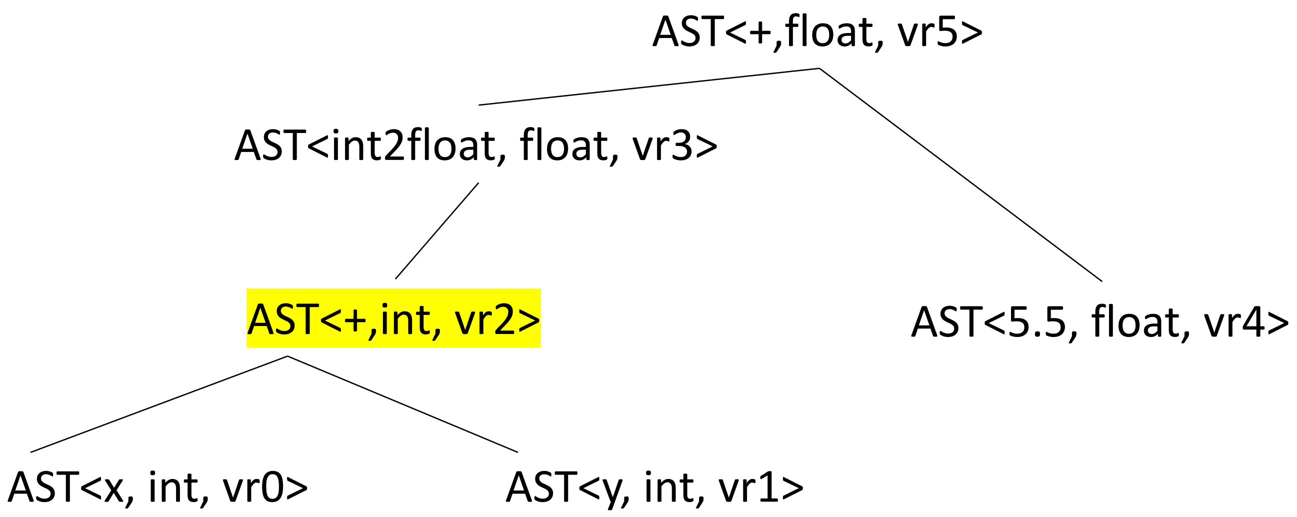
```
int x;
int y;
float w;
w = x + y + 5.5
```

After type inference



Next each AST node needs to know how to print a 3 address instruction

Let’s look at add

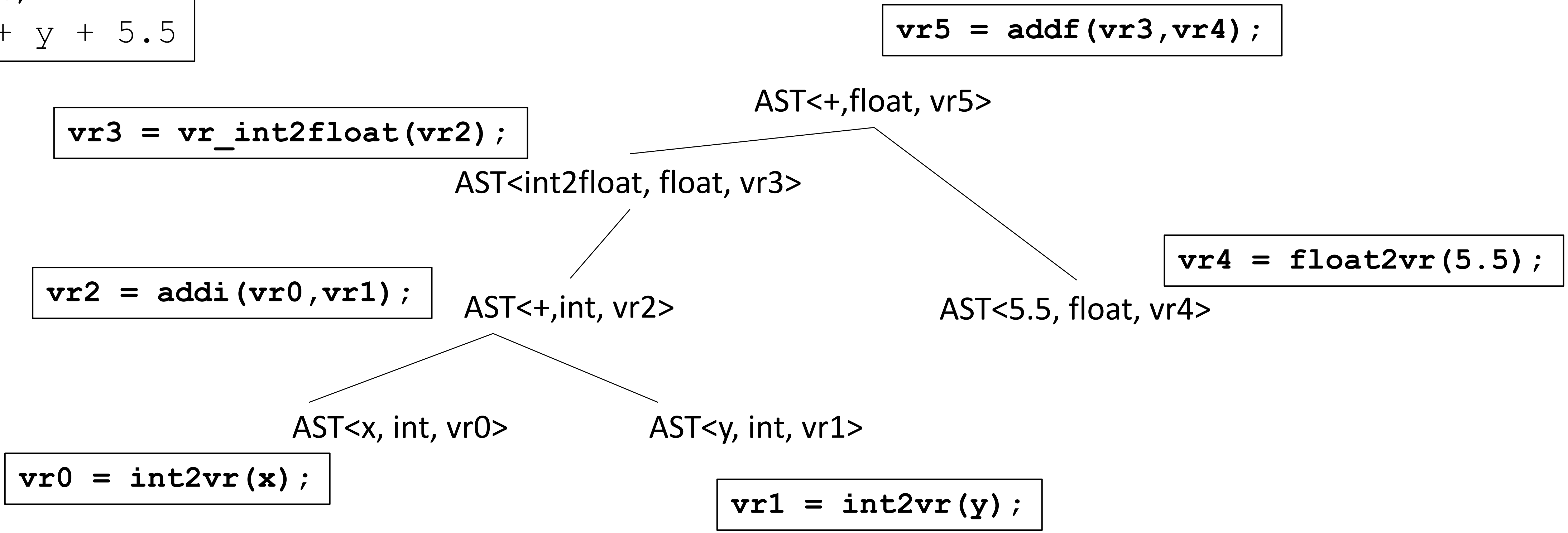



```
def get_op(self):
        if self.node_type is Types.INT:
            return "addi"
        else:
            return "addf"
```

```
return "%s = %s(%s,%s);" %
                    (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

```
vr2 = addi(vr0,vr1);
```

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



```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
   ...
   # get resources
   end_label  = mk_new_label()
   else_label = mk_new_label()
   vrX        = mk_new_vr()

   # make instructions
   ins0 = "%s = int2vr(0)" % vrX
   ins1 = "beq(%s, %s, %s);" %
          (expr_ast.vr, vrX, else_label)
   ins2 = "branch(%s)" % end_label

   # concatenate all programs
   return program0 + [ins0, ins1] + program1
          + [ins2, label_code(else_label)]
          + program2 + [label_code(end_label)]
}
```

```
if (program0) {
  program1
}
else {
  program2
}
```

We need to convert this to 3 address code

```
  program0;
  vrX = int2vr(0)
  beq(expr_ast.vr, vrX, else_label);
  program1
  branch(end_label);
else_label:
  program2
end_label:
```

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

We did these two

You do these two for your homework

Draw out for loops just like how we did with the if statements!

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

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

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 < 100; 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[1]
   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[1]
   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[1]
   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[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_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")

No more need for {}

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")
```

No more need for {}

HT0

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

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;
}
```

Scopes

What happens with multiple scopes?

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

What if x is uninitialized?

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*

See everyone on Monday!

- Finish up talking about intermediate representaitons