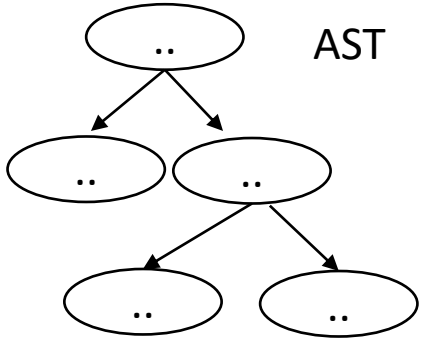
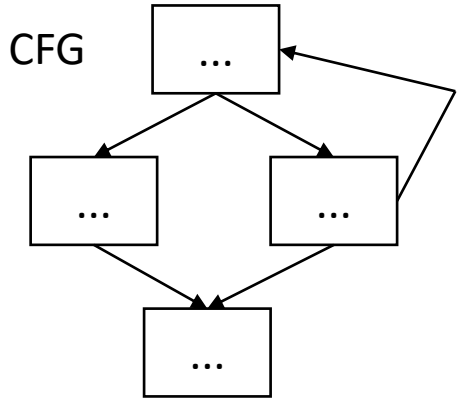
CSE110A: Compilers

May 6, 2022



Topics:

- *more 3-address code*
- *converting AST to 3-address code*
- *converting control flow statements to 3-address code*



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 1 grades are released
  - You have until Monday to let us know about any issues

- Midterm is out
  - There is a piazza post with clarifications
  - No late midterms will be accepted
    - My advice is to try to turn it in by tonight
    - that way if there are any issues, you have until Friday to turn it in
  - As always, no help guaranteed outside of business hours
    - After 5 PM on Friday, you are on your own

Announcements

- Expect HW 3 on Monday by midnight
  - It will be similar to HW 2 in terms of workload and conceptual depth
  - ***I suggest you start early***
  - HW 2 was difficult, but most of our office hours had slots in the first week it was assigned!
  - It will build on the parser of HW 2
    - You can use your own with some small modifications
    - or we will provide one
  - The idea is that you should be able to plug in parts from all the homeworks to have one big project at the end!

Announcements

- Schedule update
  - We will likely need 1 or 2 more days of the IR module
  - So we will start the optimization module next Wednesday or Friday
  - I'll adjust the schedule accordingly

- With homeworks:
  - Originally scheduled to have 5 homeworks but due to time constraints, we will likely only have 4 homeworks.
  - It's the first time I'm teaching this class. All of the homeworks, exams, and lectures are new, so we've had to adapt.
  - Apologies to those who want more compiler homeworks. You can always take CSE211 if you are interested ☺

Quiz

Quiz

What are the differences between type-checking and type-inference

Discussion

- Type inference:
  - Rules about how types convert between each other implicitly
  - Examples:
    - 2 + 3.0
    - int x = 6.0
    - 7.0 < 6.0
  - Assigns a type to each expression

- Type checking:
  - Happens during type inference.
  - If a type cannot be given to an expression then it raises an error
  - Examples?

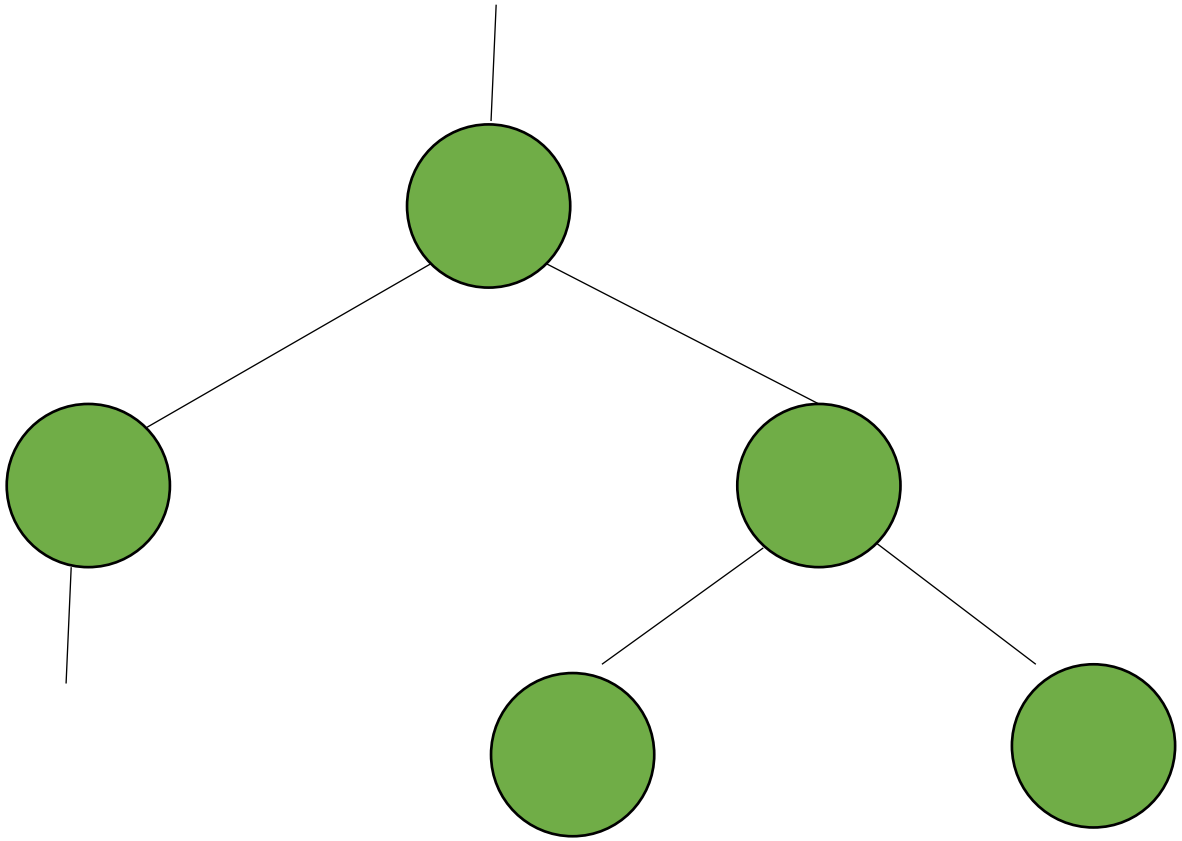
Quiz

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

- ◯ True
- ◯ False

Discussion

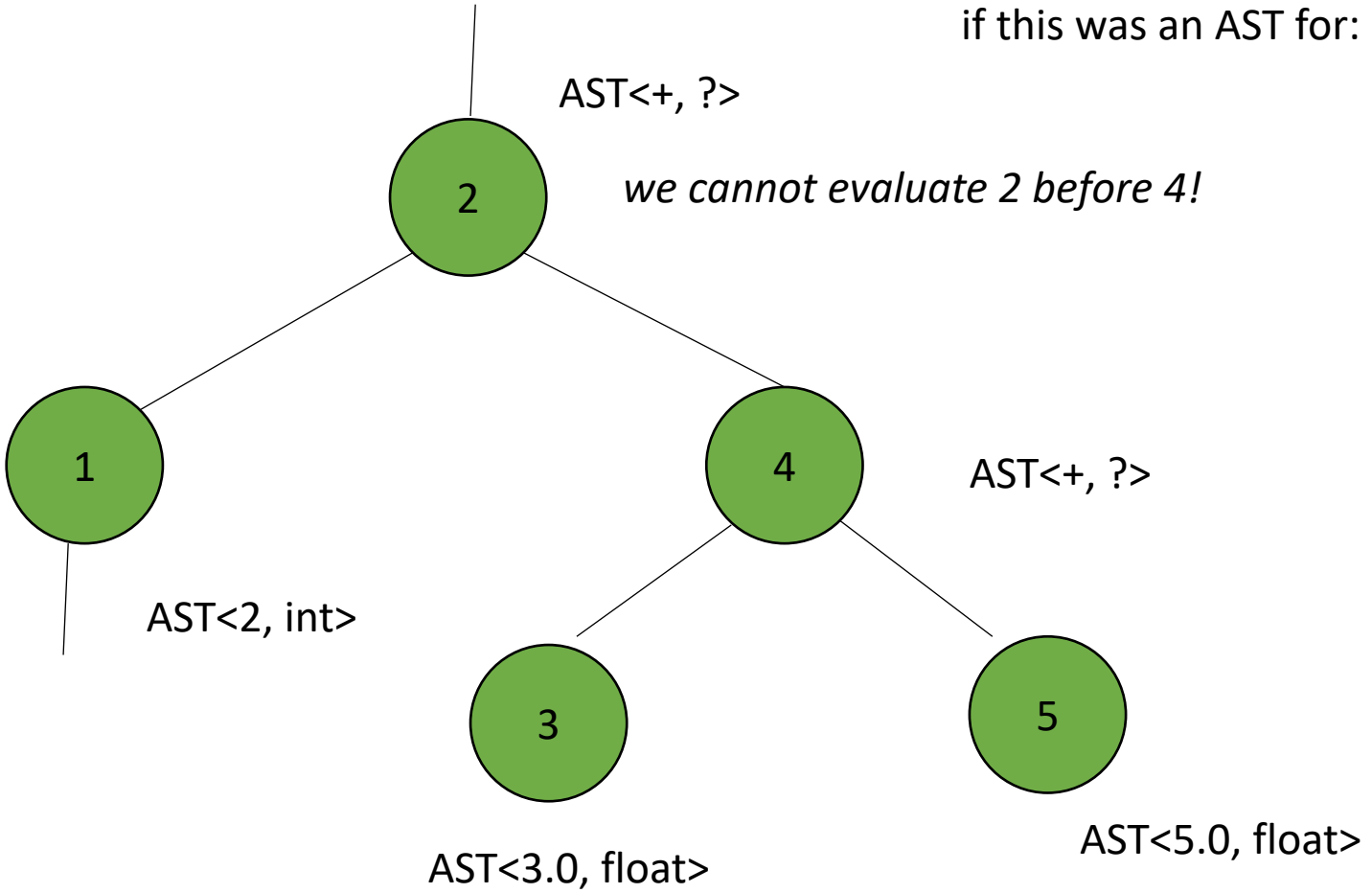
What is the in order traversal order?



Discussion

What is the in order traversal order?

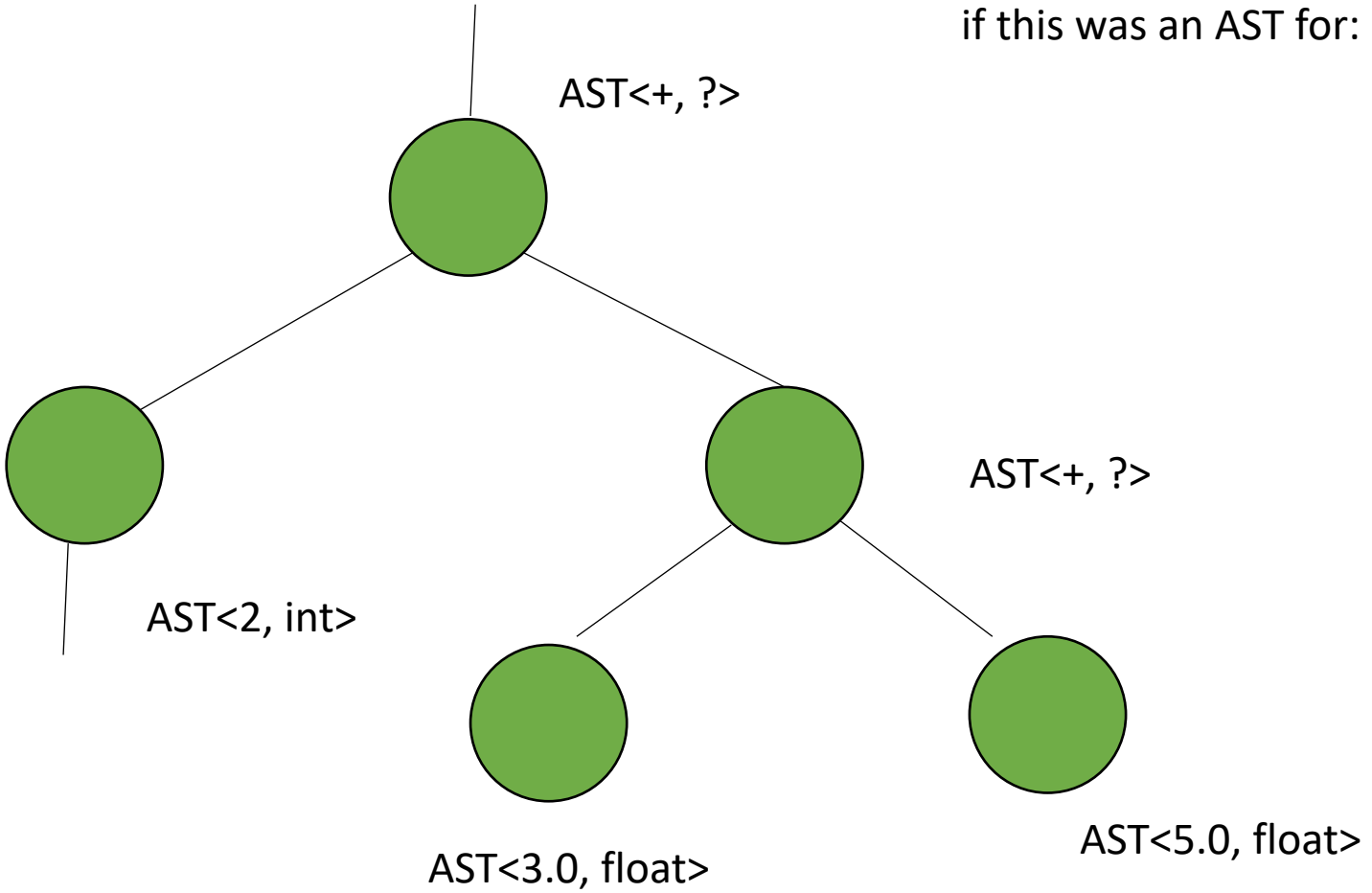
if this was an AST for: “2 + (3.0 + 5.0)”



Discussion

What is the post order traversal order?

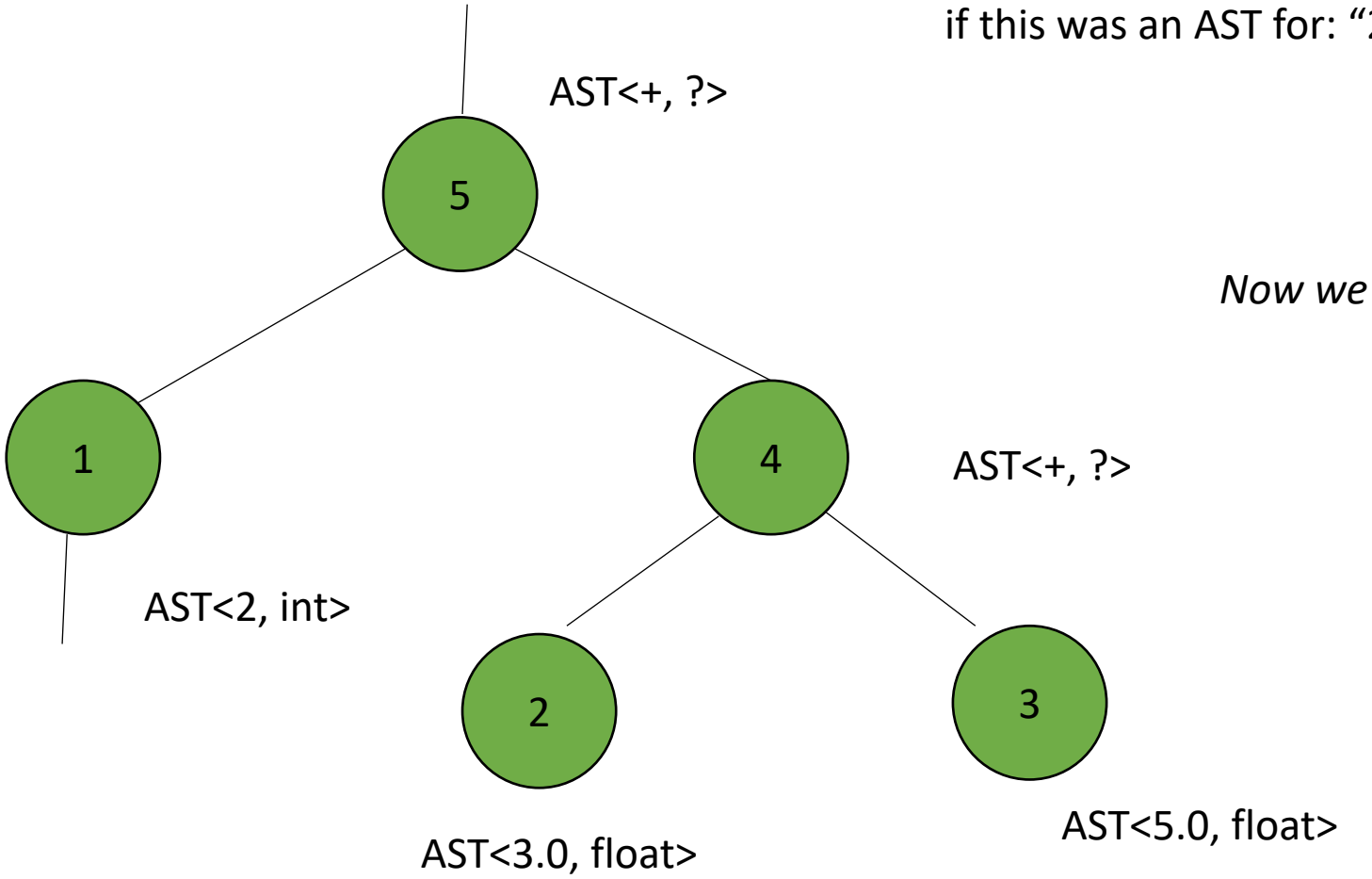
if this was an AST for: “2 + (3.0 + 5.0)”



Discussion

What is the post order traversal order?

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



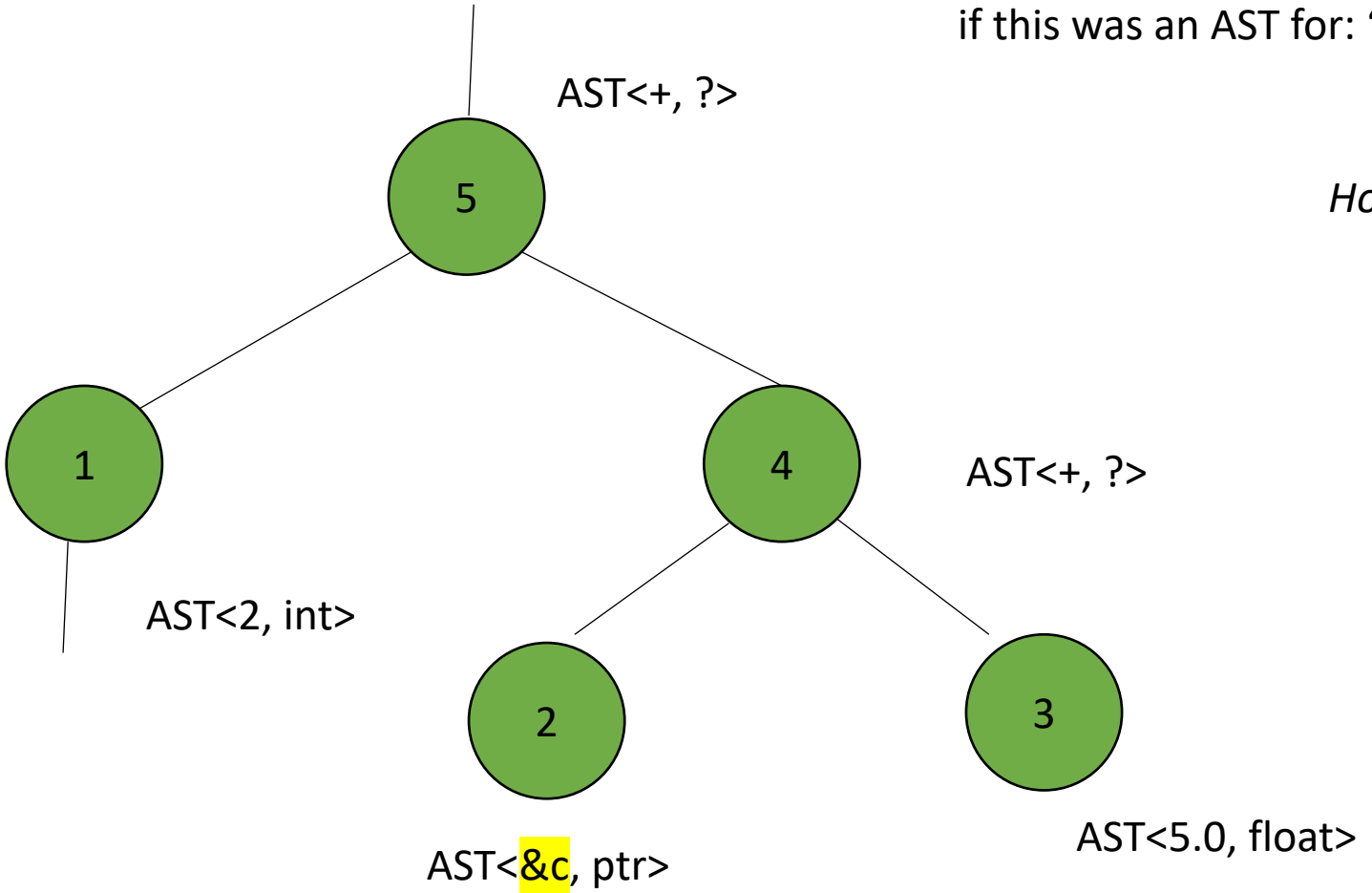
Now we can do type inference

Discussion

What is the post order traversal order?

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

How does this change things?



Discussion

Example

Does this express pass C’s type checking?

```
int *c;
int x;
c + x = 6.0;
```

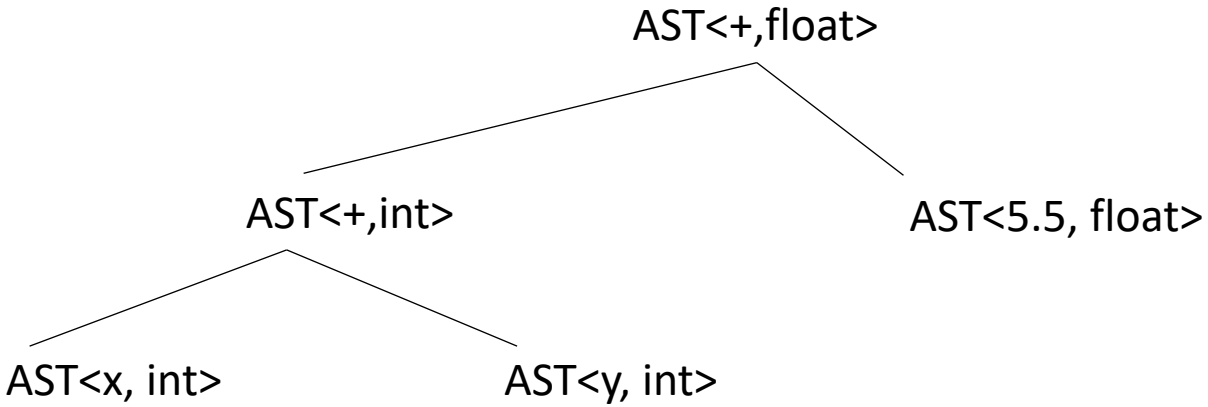
Quiz

Which form of intermediate code most closely resembles actual machine instructions?

- ◯ abstract syntax tree
- ◯ control flow graph
- ◯ stack machine code
- ◯ three address code

Discussion

AST



Much closer to parse tree

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
```

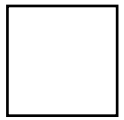
Much closer to machine code

Different IRs

- Several types of linear code:
  - 1 address code
  - 2 address code
  - 3 address code

used for stack machines, some ideas are used in the JVM and web assembly. Creates compact code

```
push 2
push 4
multiply
push 8
subtract
```



Execute this code as an exercise

Different IRs

- Several types of linear code:
  - 1 address code
  - 2 address code
  - 3 address code

used for stack machines, some ideas are used in the JVM and web assembly. Creates compact code

Stack machine can be a useful IR, but I'm not sure any machine that actually uses it.

Jeremy mentioned that the original Nintendo processor used 2-address code for its ISA. In those cases the destination registration is built into the instruction

Quiz

The instructions in the three address code depend on the ISA of the target architecture

- ○ True
- ○ False

Discussion

book

```
r0 ← x + y;
r1 ← 5 * 7;
r2 ← r0 / r1
```

this class

```
vr0 = addi(x,y);
vr1 = multi(5,7);
vr2 = divi(vr0,vr1);
```

LLVM IR

```
%8 = add nsw i32 %6, %7
%11 = mul nsw i32 5, 7
%15 = sdiv i32 %13, %14
```

It depends

Your 3-address code should be close enough to make the final translation to ISA easier

But it should be general enough to be able to target many backends.

Discussion

book

$r_0 \leftarrow x + y;$
$r_1 \leftarrow 5 * 7;$
$r_2 \leftarrow r_0 / r_1$

this class

```
vr0 = addi(x,y);
vr1 = multi(5,7);
vr2 = divi(vr0,vr1);
```

LLVM IR

```
%8 = add nsw i32 %6, %7
%11 = mul nsw i32 5, 7
%15 = sdiv i32 %13, %14
```

Types are a consideration here

It depends

Your 3-address code should be close enough to make the final translation to ISA easier

But it should be general enough to be able to target many backends.

Discussion

book

```
r0 ← x + y;
r1 ← 5 * 7;
r2 ← r0 / r1
```

this class

```
vr0 = addi(x,y);
vr1 = multi(5,7);
vr2 = divi(vr0,vr1);
```

LLVM IR

```
%8 = add nsw i32 %6, %7
%11 = mul nsw i32 5, 7
%15 = sdiv i32 %13, %14
```

Virtual registers are a consideration here
e.g., how many, if they are typed, etc.

Types are a consideration here

It depends

Your 3-address code should be close enough to make the final translation to ISA easier

But it should be general enough to be able to target many backends.

Review

- We went over some of this stuff pretty quickly last time

3-address code

- We will call our code Class-IR

Example:

Virtual registers

```
vr0 = addi(x,y);
vr1 = multi(5,7);
vr2 = divi(vr0,vr1);
```

typed instructions

3-address code

Control flow in 3 address code

- Similar to an ISA:
  - We have labels
  - and branch instructions
    - `branch x` - branch unconditionally to label z
    - `bne x,y,z` - branch to z if x and y are not equal

What does this code do?

```
label0:
  vr0 = addi(x,y);
  vr1 = multi(5,7);
  vr2 = divi(vr0,vr1);
  branch label0;
  vr3 = ...
  vr4 = ...
```

3-address code

Control flow in 3 address code

- Similar to an ISA:
  - We have labels
  - and branch instructions
    - `branch x` - branch unconditionally to label z
    - `bne x,y,z` - branch to z if x and y are not equal

What does this code do?

```
label0:
  vr0 = addi(vr0,-1);
bne vr2 0 label0;
  vr3 = ...
  vr4 = ...
```

Class-IR

- A deeper dive
- You will need to be familiar with this language for the next two homeworks
- It is untyped
  - checks your type inference
- There is a slightly modified version of Class-IR that can compile and execute in C++, which is how you will test it

Class-IR

Inputs/outputs: 32-bit typed inputs

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

Types: 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]
```

this gets us closer to assembly

```
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]
```

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

We should have an AST binary operator for each of these. They should also be close to your production rules in the grammar.

We want to make translations as easy as possible!

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 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!
```

Class-IR

unary get typed data

```
dst = operation(op0);

operations are: [vr2int, vr2float]

Example:
```

Given inputs: int x and float y

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

Class-IR

unary get untyped register

```
dst = operation(op0);

operations are: [int2vr, float2vr]

Example:
```

Given inputs: int x and float y

```
rv1 = int2vr(x);
rv2 = float2vr(2.0);
```

Example

```
adding the values 1 - 9 in to an input/output variable: int x
```

Example

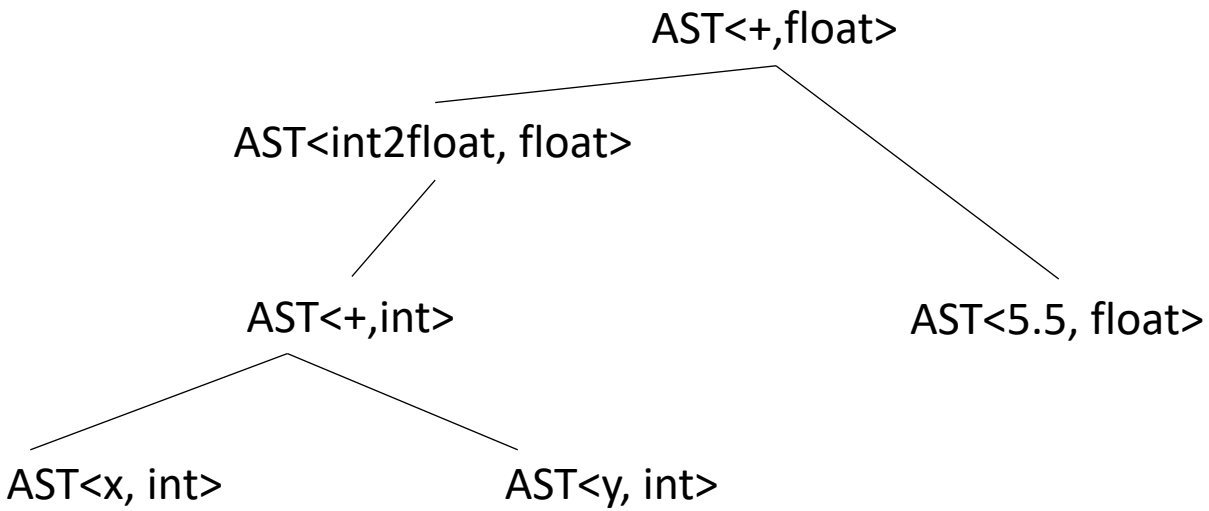
```
adding the values 1 - 9 in to an input/output variable: int x

  vr0 = int2vr(1);
  vr1 = int2vr(1);
  vr2 = int2vr(10);
loop_start:
  vr3 = lti(vr0, vr2);
  bne(vr3, vr1, end_label);
  vr4 = int2vr(x);
  vr5 = addi(vr4,r0);
  x = vr2int(vr5);
  vr0 = addi(vr0, vr1);
  branch(loop_start);
end_label:
```

Converting AST into Class-IR

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

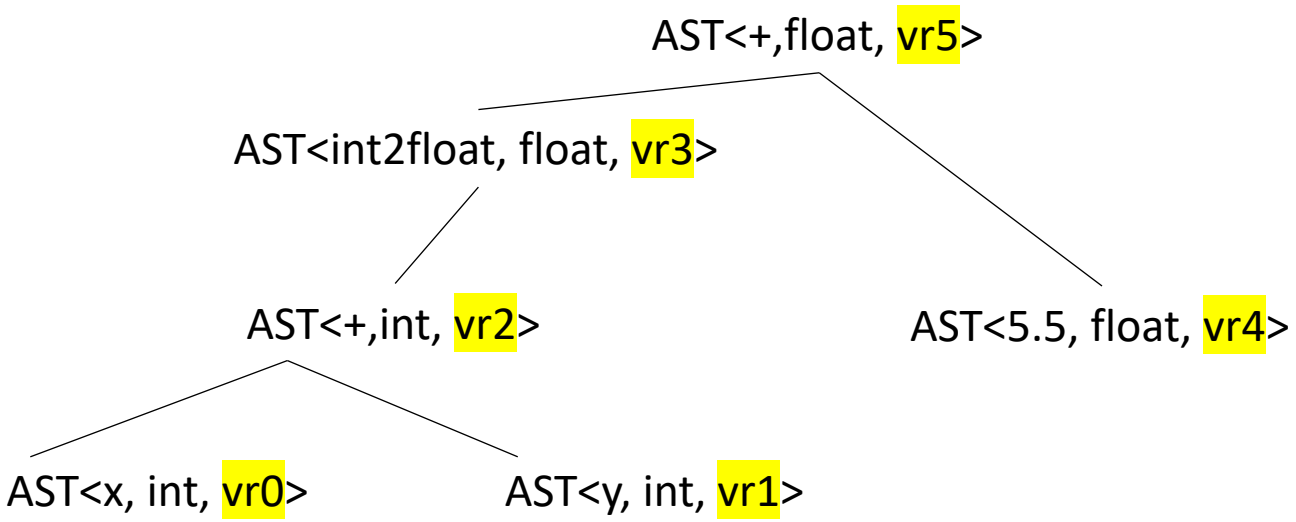
After type inference



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

A reminder on where we are with our code

Our base AST Node needs a virtual register

```
class ASTNode():
    def __init__(self):
        self.node_type = None
        self.vr = None
        pass

    def set_vr(self, r):
        self.vr = r

    def get_vr(self):
        return self.vr
```

A simple class that can allocate virtual registers

```
class VRAllocator():
    def __init__(self):
        self.count = 0


    def get_new_register(self):
        vr = "vr" + str(self.count)
        self.count += 1
        return vr
```

A reminder on where we are with our code

Our base AST Node needs a virtual register

```
class ASTNode():
    def __init__(self):
        self.node_type = None
        self.vr = None
        pass

    def set_vr(self, r):
        self.vr = r

    def get_vr(self):
        return self.vr
```

To provide each node in the AST a virtual node, simply traverse the tree (any order), get a new virtual register, and set the virtual register

A simple class that can allocate virtual registers

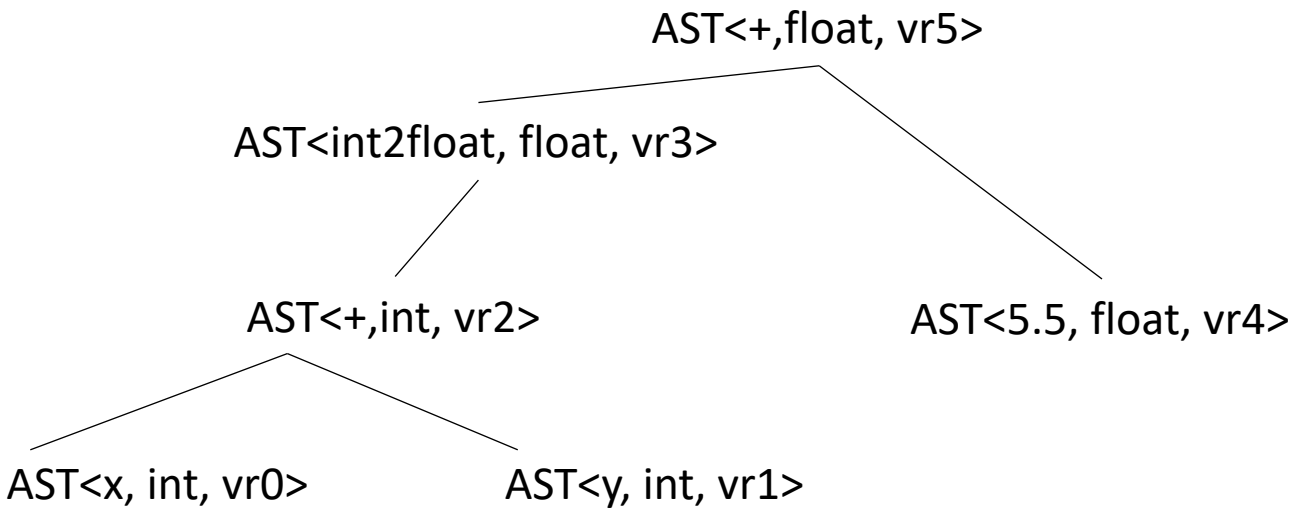
```
class VRAllocator():
    def __init__(self):
        self.count = 0


    def get_new_register(self):
        vr = "vr" + str(self.count)
        self.count += 1
        return vr
```

Converting AST into Class-IR

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

After type inference

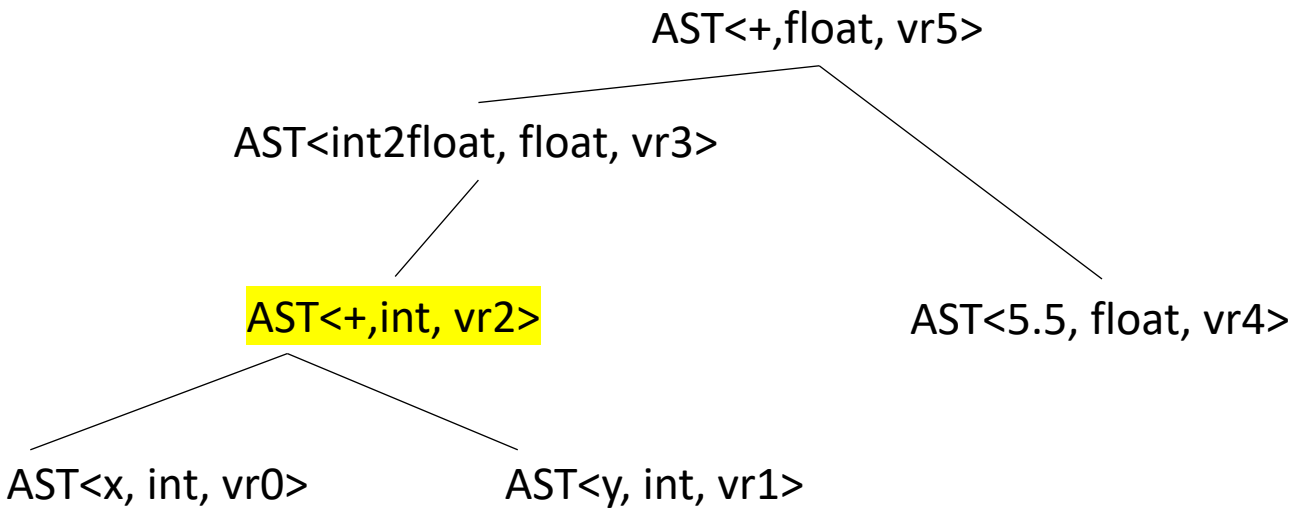


Next each AST node needs to know how to print a 3 address instruction

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

```
class ASTPlusNode(ASTBinOpNode):
    def __init__(self, l_child, r_child):
        super().__init__(l_child,r_child)

    # return a string of the three address instruction
    # that this node encodes
    def three_addr_code(self):
       ??
```

```
class ASTPlusNode(ASTBinOpNode):
    def __init__(self, l_child, r_child):
        super().__init__(l_child,r_child)

    # return a string of the three address instruction
    # that this node encodes
    def three_addr_code(self):
       ??
```

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

```
class ASTPlusNode(ASTBinOpNode):
    def __init__(self, l_child, r_child):
        super().__init__(l_child,r_child)

    # return a string of the three address instruction
    # that this node encodes
    def three_addr_code(self):
       ??
```

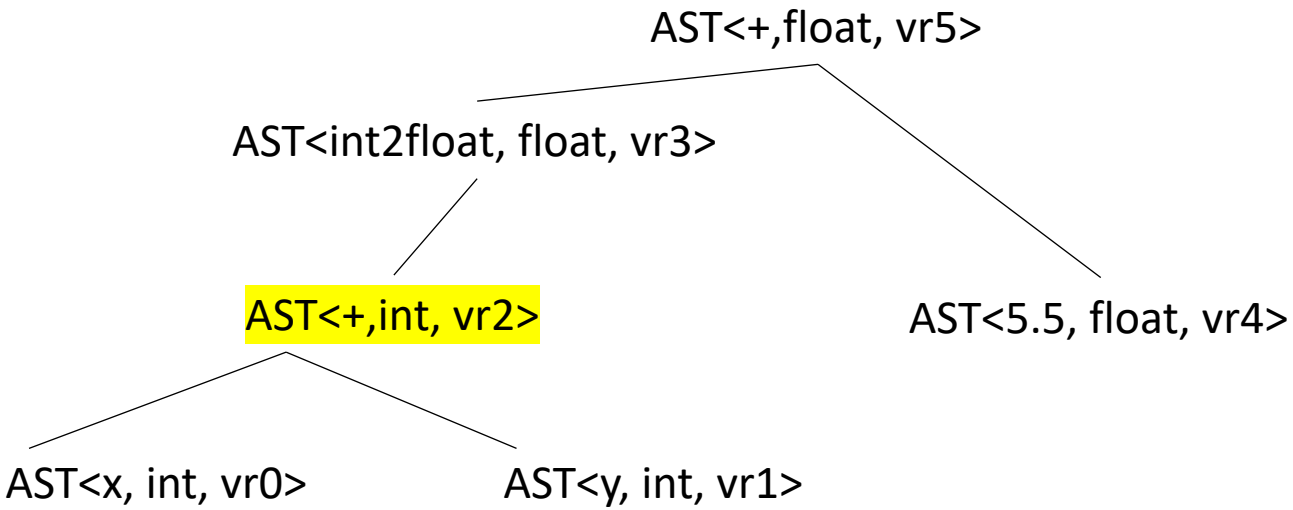
```
return "%s = %s(%s,%s);" %
                    (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

What is this one?

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

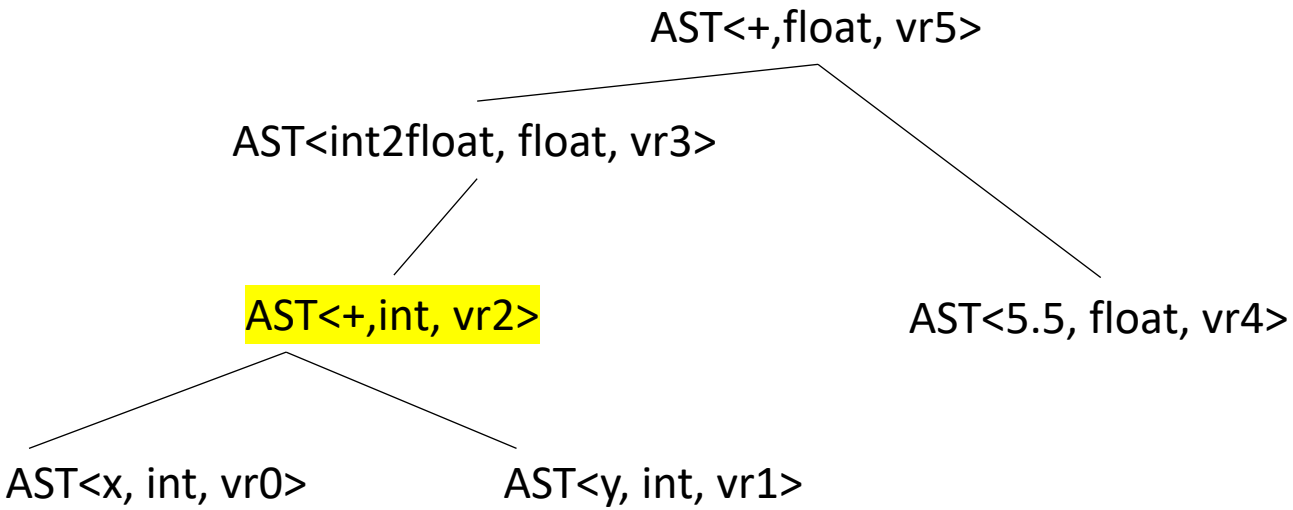
```
return "%s = %s(%s,%s);" %
                    (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

What is this one?




```
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)
```




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

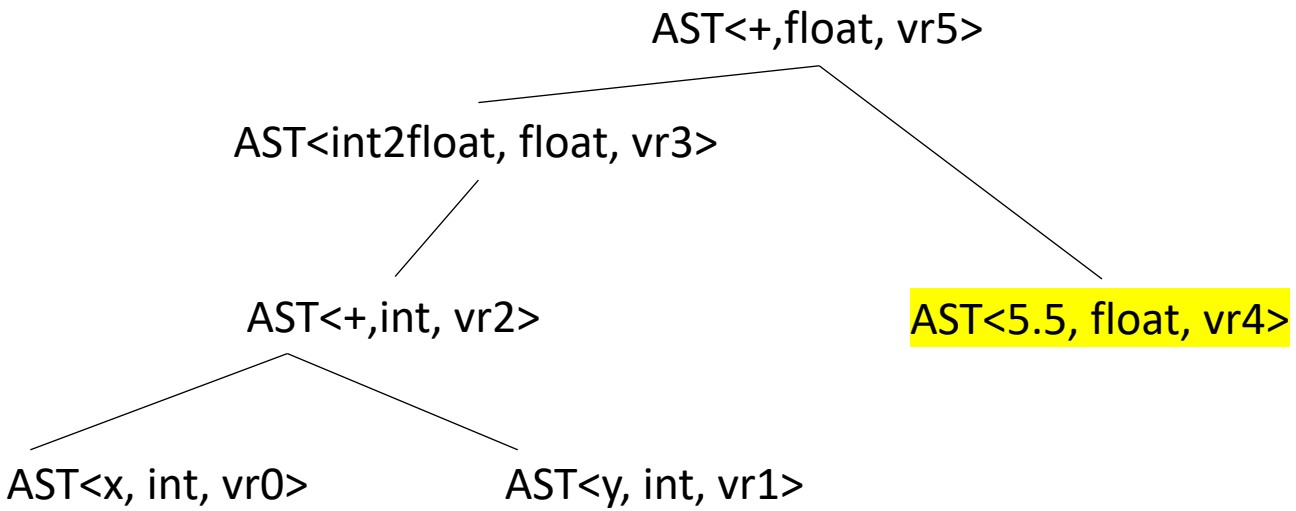
```
return "%s = %s(%s,%s);" %
                    (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

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

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 a leaf

```
class ASTIDNode(ASTLeafNode):
    def __init__(self, value, value_type):
        super().__init__(value)
        self.set_type(value_type)


    # return a string of the three address instruction
    # that this node encodes
    def three_addr_code(self):
       ??
```

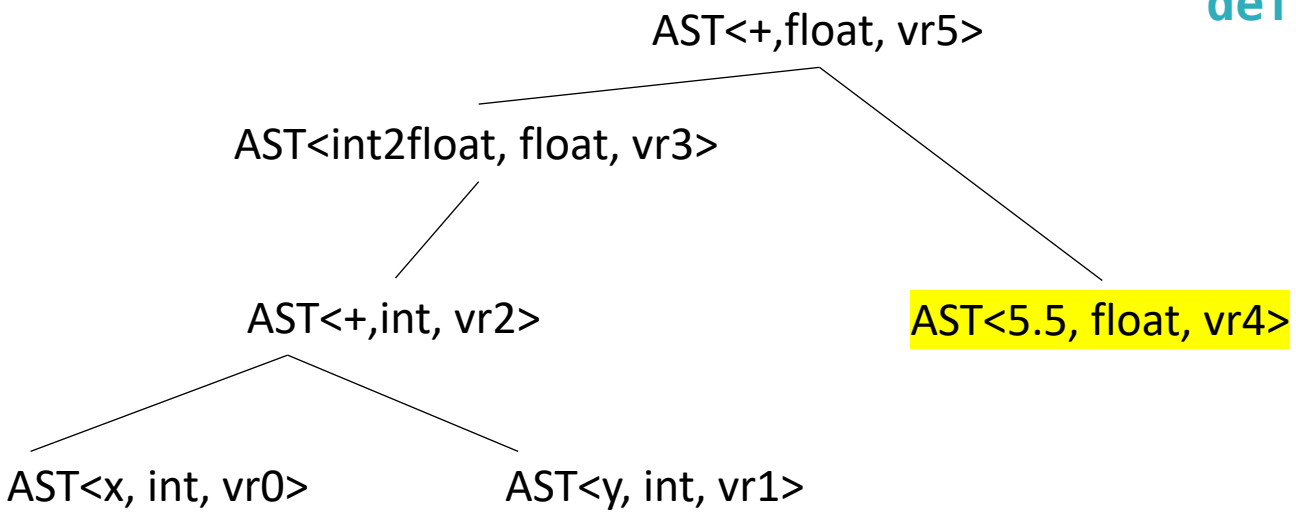
```
class ASTIDNode(ASTLeafNode):
    def __init__(self, value, value_type):
        super().__init__(value)
        self.set_type(value_type)


    # return a string of the three address instruction
    # that this node encodes
    def three_addr_code(self):
        ??
```

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

```
def get_op(self):
        if self.node_type is Types.INT:
            return ”int2vr"
        else:
            return ”float2vr"
```

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



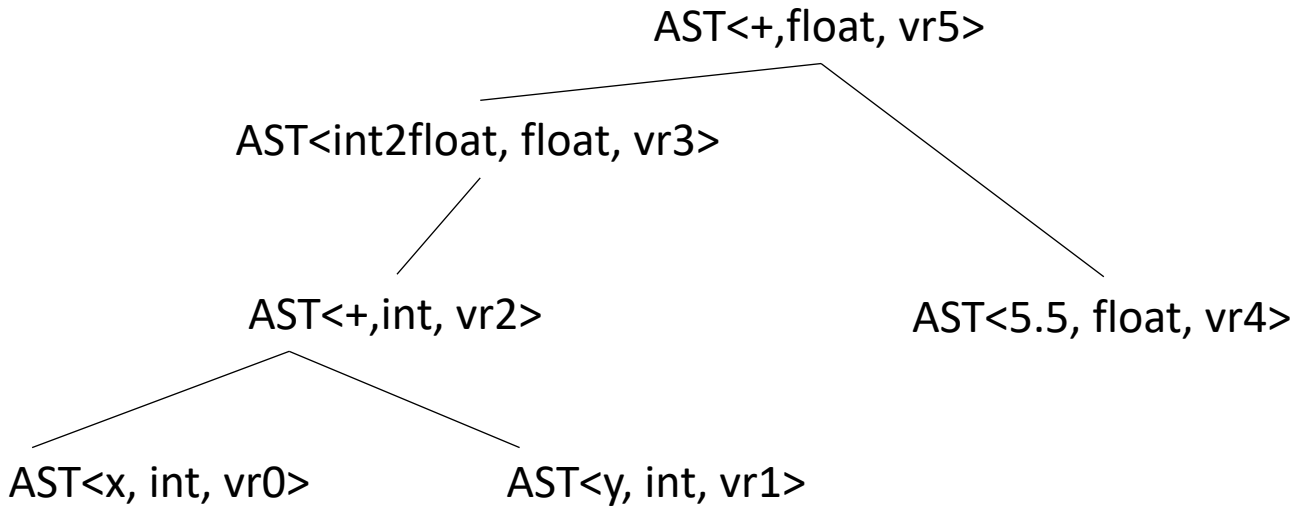

```
def get_op(self):
        if self.node_type is Types.INT:
            return ”int2vr"
        else:
            return ”float2vr"
```

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

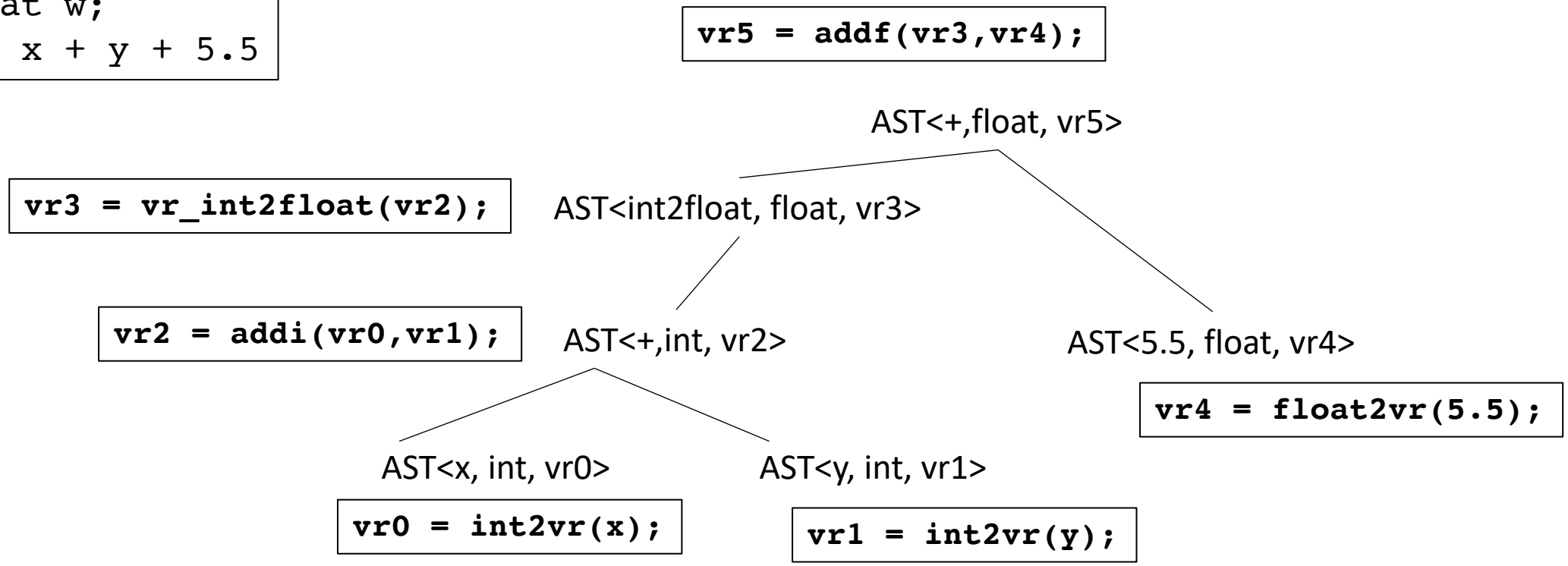
```
vr4 = float2vr(5.5);
```

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

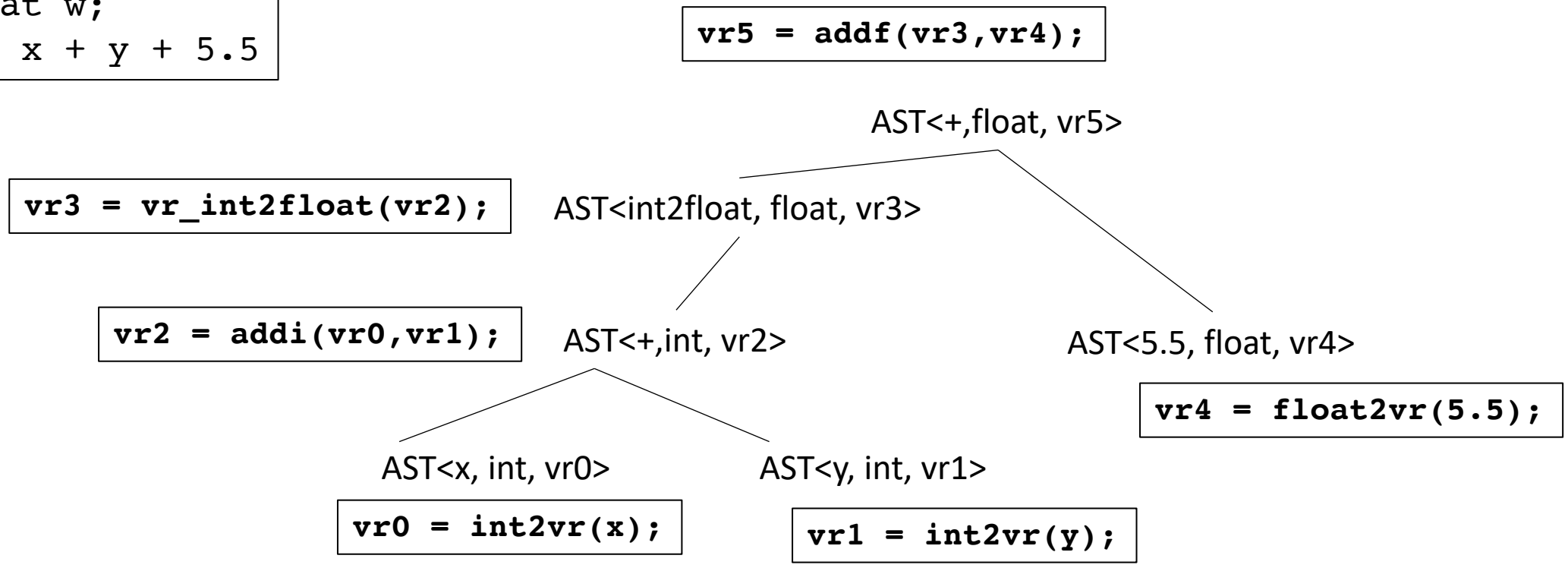
we can get 3 address instructions for each node



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

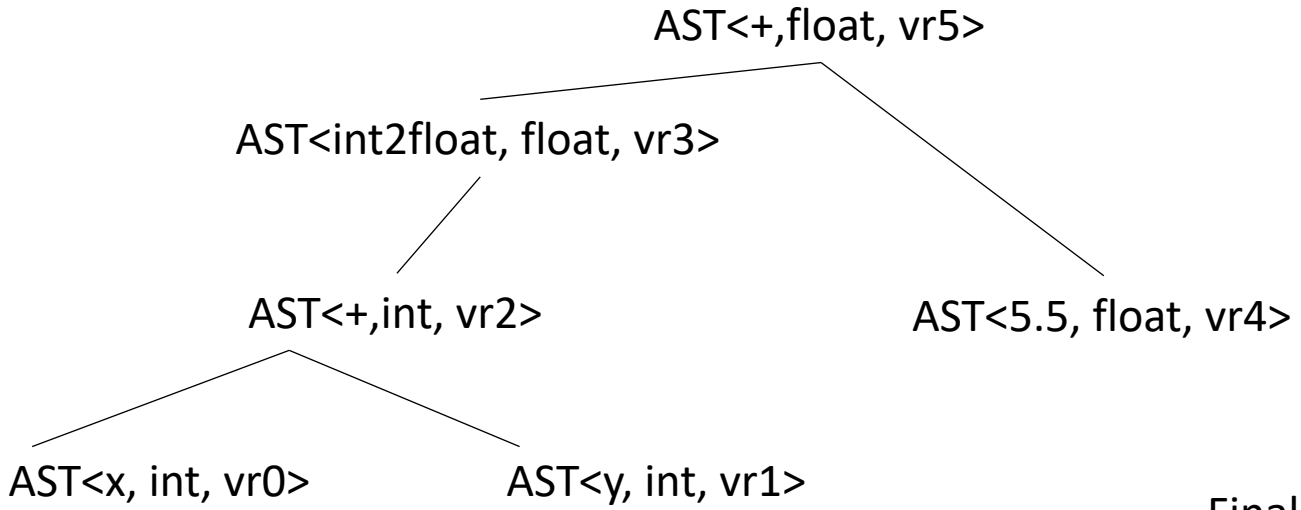


```
int x;
int y;
float w;
w = x + y + 5.5
```



What now?
We can create a 3 address program doing a post-order traversal

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



We can create a 3 address program doing a post-order traversal

Final program

```
vr0 = int2vr(x);
vr1 = int2vr(y);
vr2 = addi(vr0,vr1);
vr3 = vr_int2float(vr2);
vr4 = float2vr(5.5);
vr5 = addf(vr3,vr4);
```

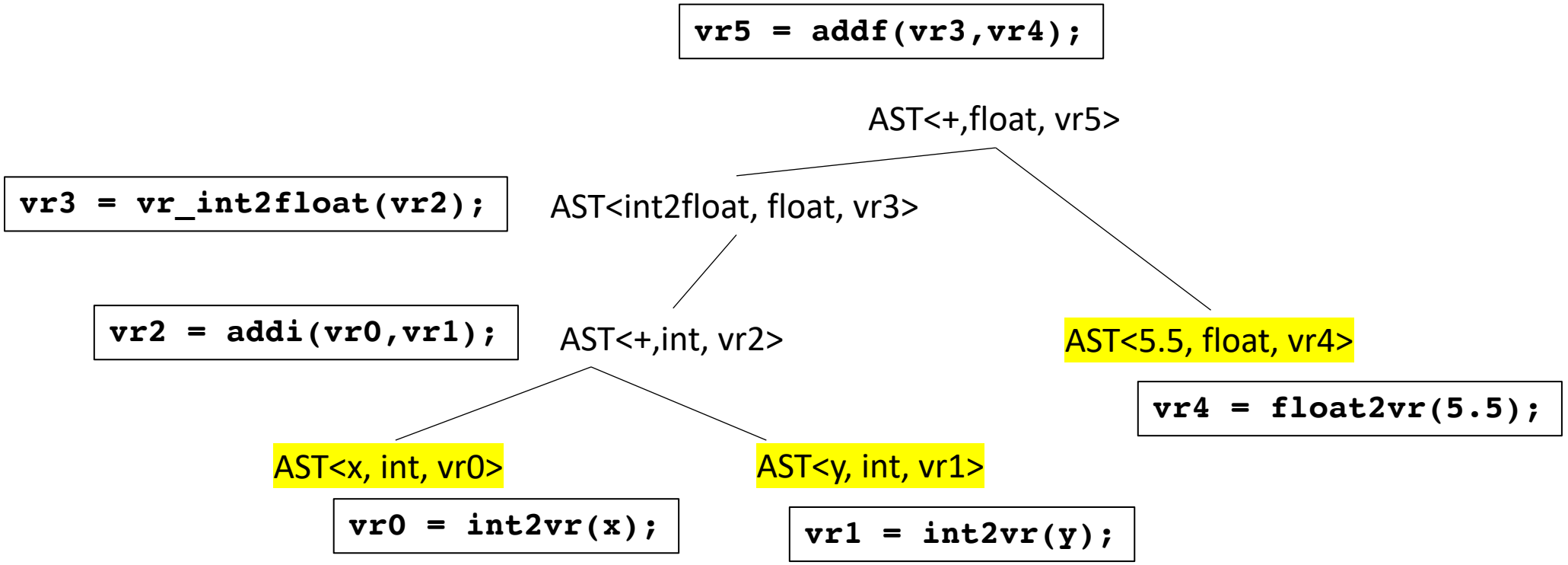
How does this actually look in code?

Each AST node (expression) needs a function:

- `linearize_expr()`

Gives a list of 3 address instructions for the expression

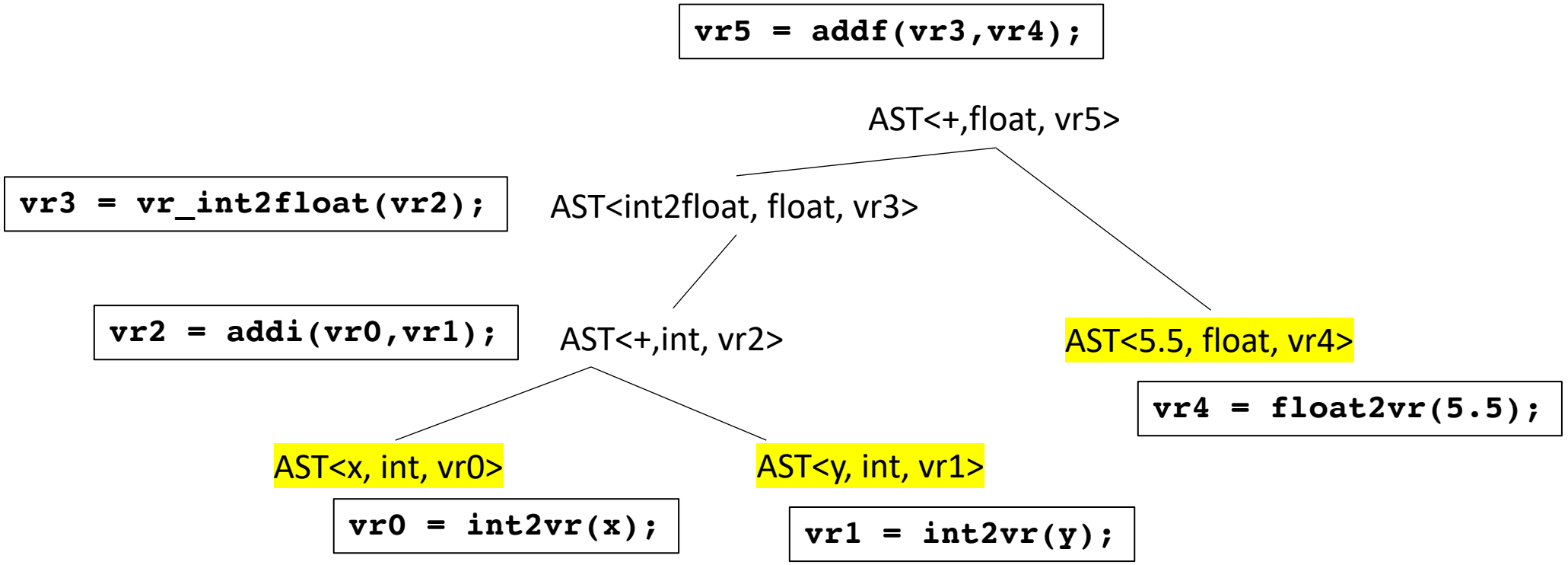
As always, we’re going to use recursion



How do you linearize a leaf node?

```
class ASTIDNode(ASTLeafNode):
    def __init__(self, value, value_type):
        super().__init__(value)
        self.set_type(value_type)


    def linearize_expr(self):
      ???
```

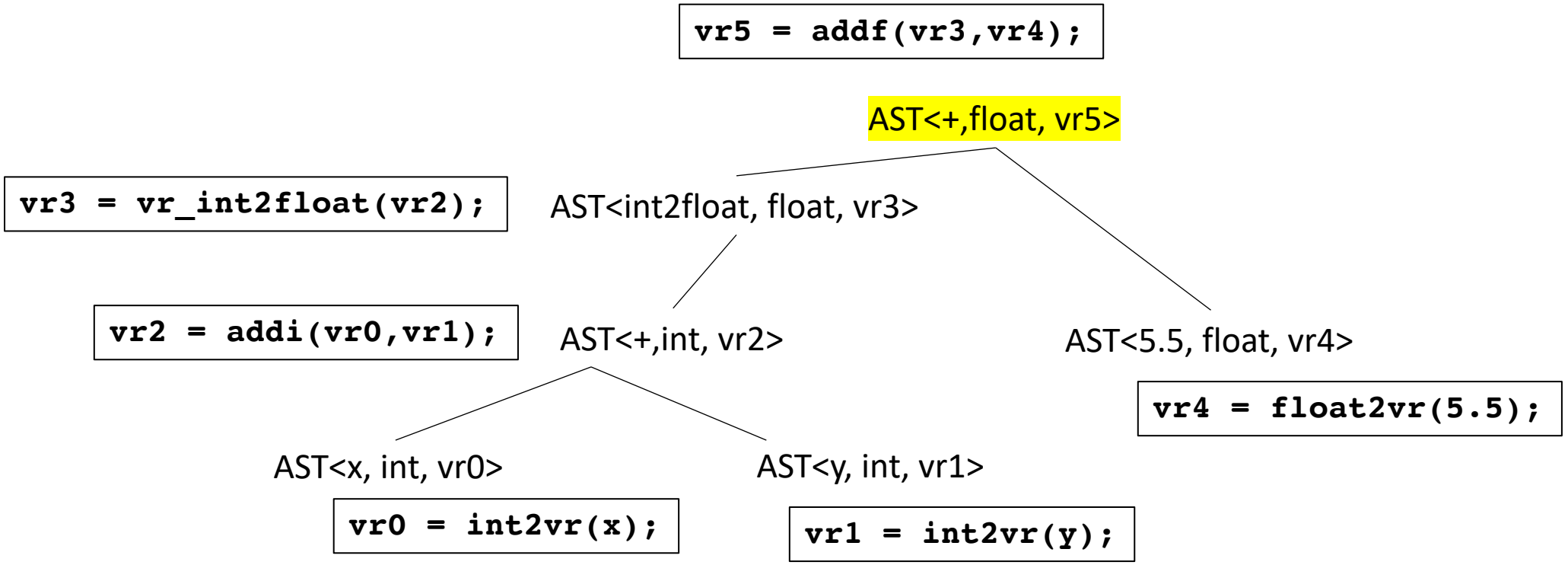


How do you linearize a leaf node?

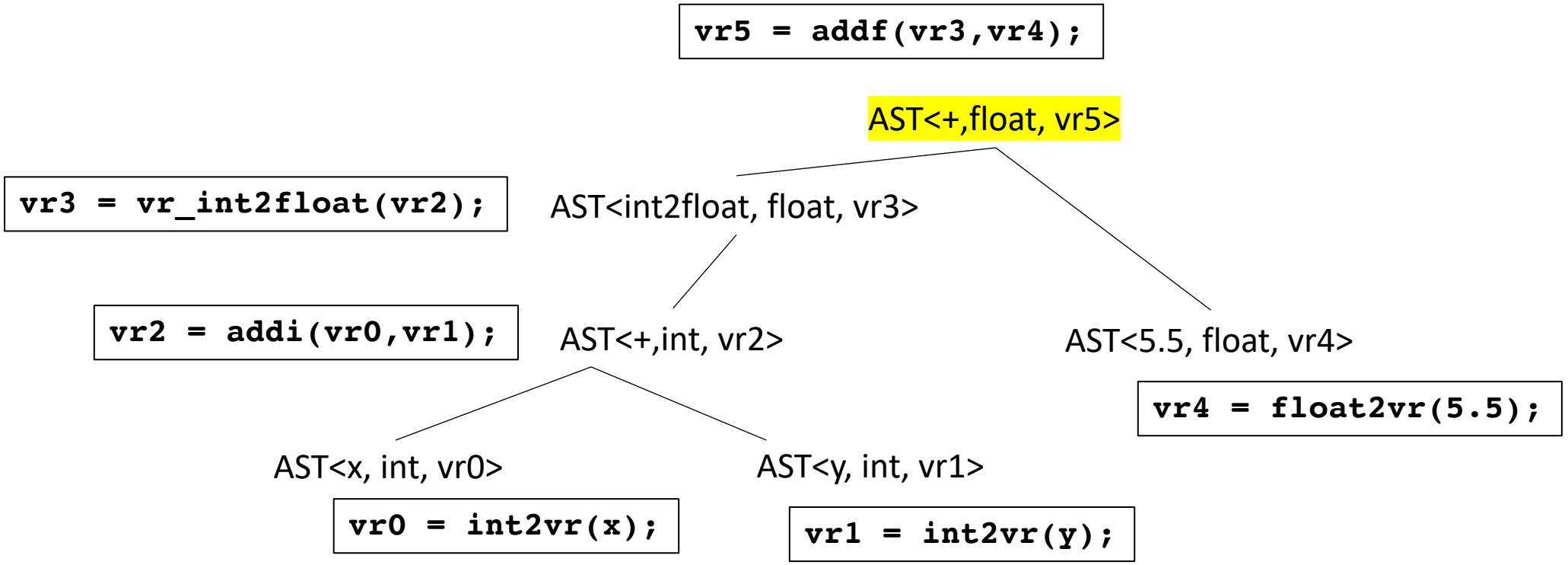
```
class ASTIDNode(ASTLeafNode):
    def __init__(self, value, value_type):
        super().__init__(value)
        self.set_type(value_type)


    def linearize_expr(self):
        return [self.three_addr_code()]
```

It needs to return a list, so we just put the leaf's instruction in the list

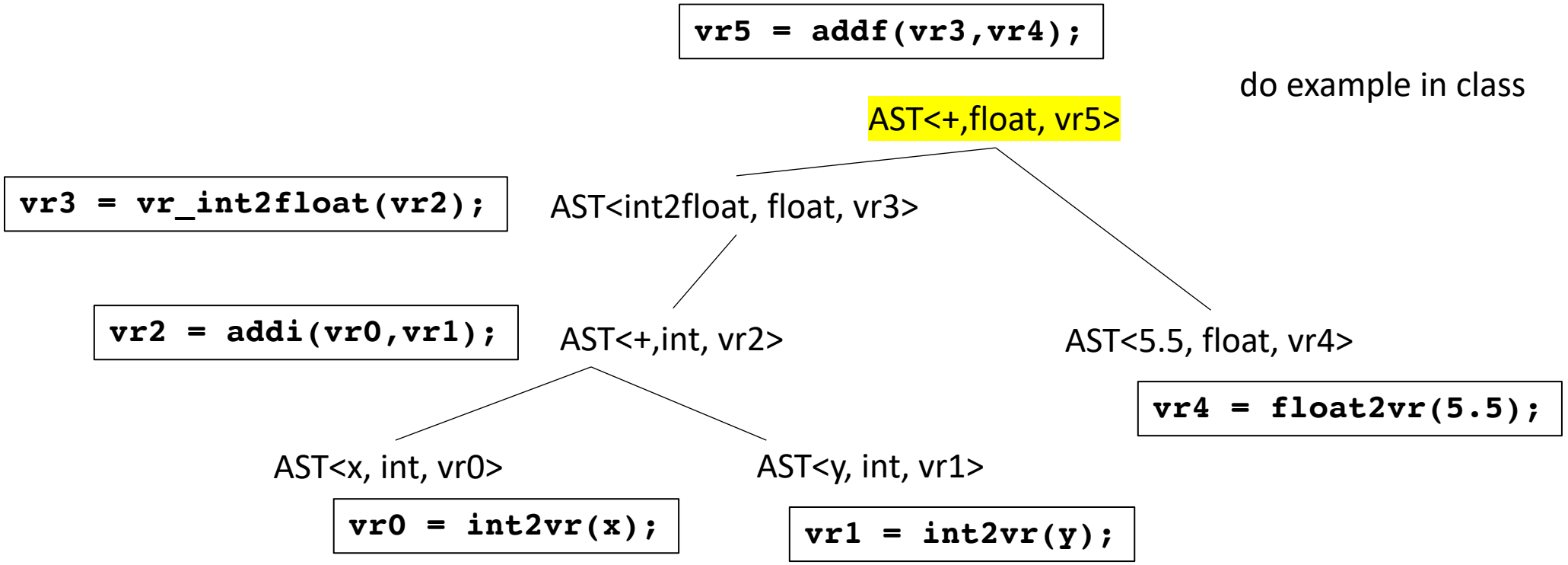


How do you linearize a non leaf node?



How do you linearize a non leaf node?

1. Linearize the children
2. concatenate the lists
3. append your 3 address instruction to the end.



do example in class

How do you linearize a non leaf node?

1. Linearize the children
2. concatenate the lists
3. append your 3 address instruction to the end.

Converting ASTs into 3 address code summary

- Each node gets a virtual register
- Each node needs to implement a function to get a three address instruction
- Each node needs a linearize function

Backing up to an even higher level

- We know how to parse an expression: `parse_expr`
- We know how to create an AST during parsing
- We know how to do type inference on an AST
- We know how to convert a type-safe AST into 3 address code

Backing up to an even higher level

- We can now define what our parser will return: A list of 3 address code

- We can get 3 address code from parsing expressions, now we just need to get it from statements

From our grammar

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

Our top down parser should have a function called `parse_statement`

This should return a list of 3 address code instructions that encode the statment

From our grammar

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

Our top down parser should have a function called `parse_statement`

This should return a list of 3 address code instructions that encode the statment

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

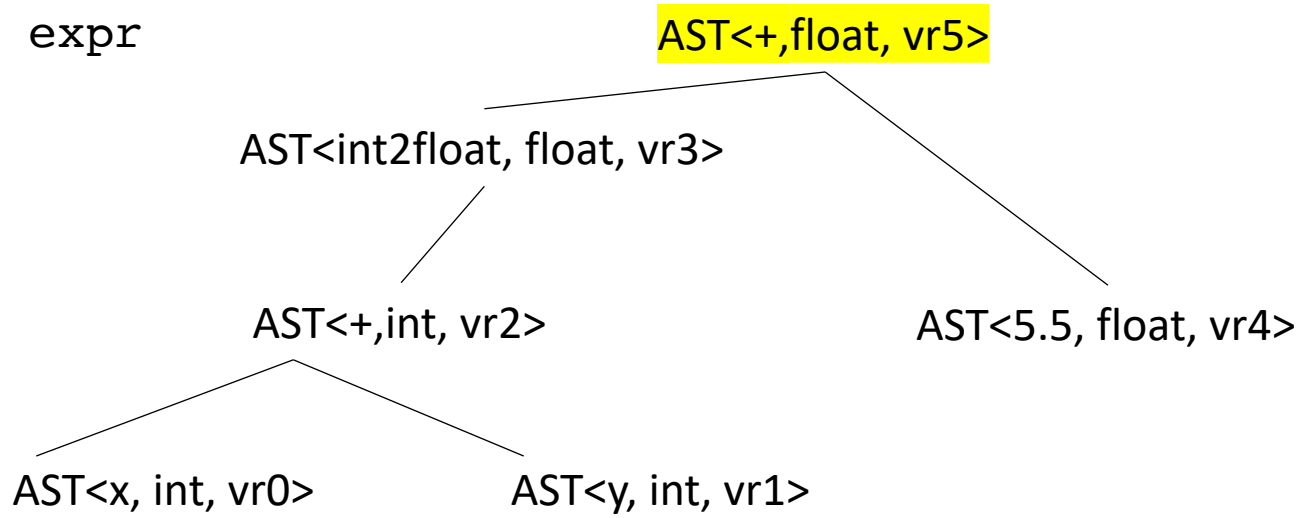
```
assignment_statement_base := ID ASSIGN expr
```

```
{
    id_name = to_match[1]
    eat(“ID”);
    eat(“ASSIGN”);
    ast = parse_expr()
    type_inference(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = “%s = %s” % ?
    return program + [new_inst]
}
```

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

```
assignment_statement_base := ID ASSIGN expr
```

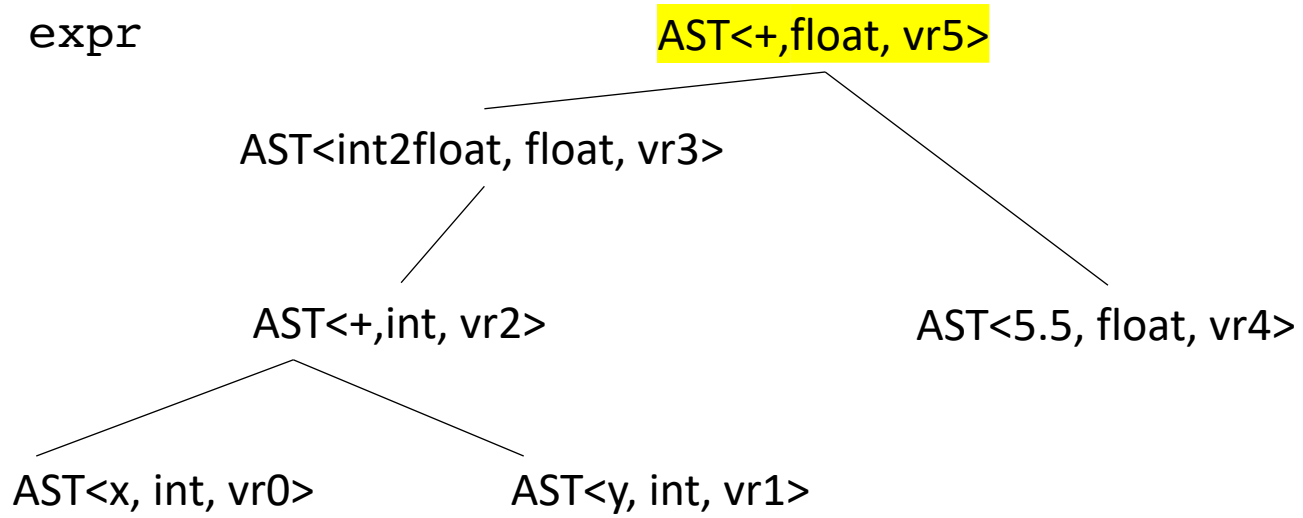
```
{
    id_name = to_match[1]
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % ?
    return program + [new_inst]
}
```



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

```
assignment_statement_base := ID ASSIGN expr
```

```
{
    id_name = to_match[1]
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % (id_name, ast.vr)
    return program + [new_inst]
}
```



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

```
assignment_statement_base := ID ASSIGN expr
```

```
{
    id_name = to_match[1]
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % (id_name, ast.vr)
    return program + [new_inst]
}
```

program

```
vr0 = int2vr(x);
vr1 = int2vr(y);
vr2 = addi(vr0,vr1);
vr3 = vr_int2float(vr2);
vr4 = float2vr(5.5);
vr5 = addf(vr3,vr4);
```

new inst

```
w = vr5
```

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

```
assignment_statement_base := ID ASSIGN expr
```

```
{
    id_name = to_match[1]
    eat(“ID”);
    eat(“ASSIGN”);
    ast = parse_expr()
    type_inference(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = “%s = %s” % (id_name, ast.vr)
    return program + [new_inst]
}
```

What are we missing here?

1. If the type of ID doesn’t match the type of the ast, then the ast needs to be converted.

2. ID should be checked if it is an input/output variable. which means it will need to be handled differently.

3. You need to check the ID in the symbol table

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

```
assignment_statement_base := ID ASSIGN expr
```

```
{
    id_name = to_match[1]
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % (id_name, ast.vr)
    return program + [new_inst]
}
```

What are we missing here?

1. If the type of ID doesn't match the type of the ast, then the ast needs to be converted.

2. ID should be checked if it is an input/output variable. which means it will need to be handled differently.

3. You need to check the ID in the symbol table

It can get a little messy

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

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


{
   eat(“IF”);
   eat(“LPAR”);
   program0 = # Get program from expr
   eat(“RPAR”);
   program1 = # Get program from statement
   eat(“ELSE”)
   program2 = # Get program from statement
   ...

}
```

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


{
   eat(“IF”);
   eat(“LPAR”);
   program0 = # Get program from expr
   eat(“RPAR”);
   program1 = # Get program from statement
   eat(“ELSE”)
   program2 = # Get program from statement
   ...
}
```

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

We need to convert this to 3 address code

```
  beq (program0, 0, else_label)
  program1;
  branch end_label;
else_label:
  program2
end_label:
```

See everyone on Monday

- We’ll discuss more about turning statements into 3 address code