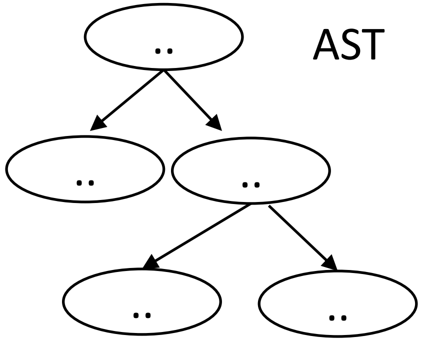
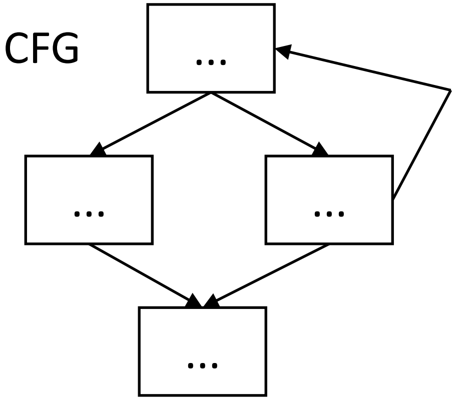
CSE110A: Compilers

May 15, 2023



Topics:

- *Finish up type checking*
- *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 3 is due today!
- We are working on grading HW 3 and midterm and plan to have some grades released by Friday
- HW 4 is planned for release today
  - Big assignment: 2 weeks to do it
  - Type inference and producing 3 address code

Quiz

Quiz

In Python, the type of a function is its return type

- ○ True
- ○ False

Discussion

- The type of a function call ***in an expression*** is the return type

- Type of a function
  - in python it is just called a function
  - in many other languages it is the full type signature
  - Example:
    - `float foo(int x)`
    - is type: *int* → *float*

Quiz

Python is a _____ Language

- ○ Statically Strongly Typed
- ○ Statically Weakly Typed
- ○ Dynamically Strongly Typed
- ○ Dynamically Weakly Typed

Discussion

- static vs. dynamic types?

- strong vs weak types?

https://stackoverflow.com/questions/2690544/what-is-the-difference-between-a-strongly-typed-language-and-a-statically-typed

Discussion

- static vs. dynamic types
  - Static means types are determined at compile time
    - Pros: compiler can emit the exact right ISA instruction, no need to check
  - Dynamic means types are checked at runtime
    - Pros: you can write more generic code

- strong vs weak types
  - Not a clear meaning of strong/weak types
  - might refer to:
    - if types are automatically converted by the compiler or runtime e.g. ints to floats
    - if a variable can change its type during runtime

https://stackoverflow.com/questions/2690544/what-is-the-difference-between-a-strongly-typed-language-and-a-statically-typed

Quiz

Expressions always have a type

- ○ True
- ○ False

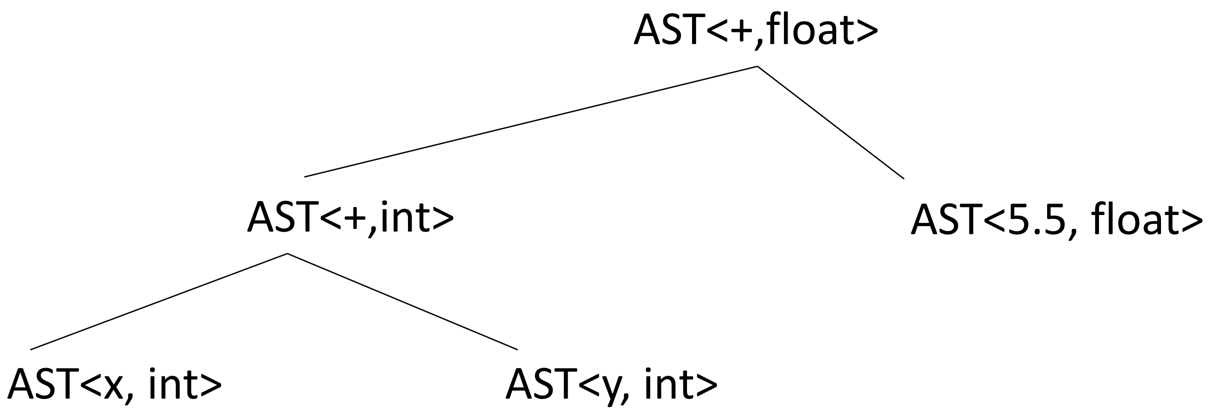
Discussion

- Definition of expression: it returns a value. If it has a value, then it has a type
- In static languages, we can determine the type of the expression at compile time
- Using an AST we can see that any node can be an expression

Discussion

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

All of these nodes have a type!



Quiz

Type of IDs are stored in the Symbol Table during the declaration statement

- ○ True
- ○ False

Symbol Table

Say we are matched the statement:
`int x;`

- `SymbolTable ST;`

(TYPE, ‘int’) (ID, ‘x’)

declare_statement ::= TYPE ID SEMI

```
{
  value_type = self.to_match[1]
  eat(TYPE)
  id_name = self.to_match[1]
  eat(ID)
  ST.insert(id_name, value_type)
  eat(SEMI)
}
```

get the type from the TYPE lexeme

record the type in the symbol table

add the type at parse time

```
Unit ::= ID
      |   NUM
```

```
def parse_unit(self, lhs_node):
    # ... for applying the first production rule (ID)
    value = self.next_word[1]
    # ... Check that value is in the symbol table
    node = ASTIDNode(value, ST[value])
    return node
```

when we create the ID
node, provide the type

A reminder on where we are with our code

Enum for types

```
from enum import Enum

class Types(Enum):
    INT = 1
    FLOAT = 2
```

Our base AST Node needs a type

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

    def set_type(self, t):
        self.node_type = t

    def get_type(self):
        return self.node_type
```

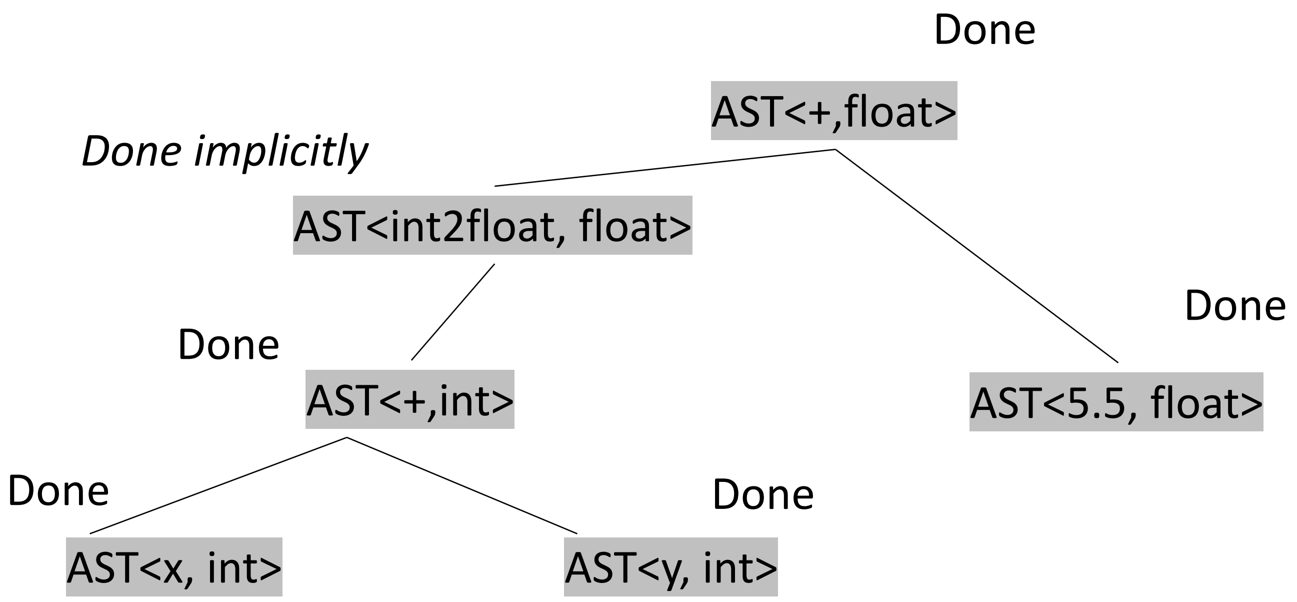
Now we need to set the types for the leaf nodes

```
class ASTNumNode(ASTLeafNode):
    def __init__(self, value):
        super().__init__(value)
        if is_int(value):
            self.set_type(Types.INT)
        else:
            self.set_type(Types.FLOAT)
```

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

Review type inference

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




```
def type_inference(n):

  case split on type of n:

  if n is a leaf node:
    return n.get_type()

  if n is a bin op node:
     do type inference on children
     t = lookup type from table
     set n type to t
     do any required type conversions
     return t
```

Done

Type inference

Table for most binary ops

left child	right child	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

what are binary ops that don't fit this?

Type inference

Table for most binary ops

left child	right child	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

what are binary ops that don't fit this?

Table for assignment binary ops

left child	right child	result
`int`	`int`	`int`
`int`	`float`	`int`
`float`	`int`	`float`
`float`	`float`	`float`

Result is what is being assigned too

Type inference

Table for most binary ops

left child	right child	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

It is up to the language designer to create these tables! Most follow a natural progression: **bool to int to float** and size promotion: **short to int to long**

what are binary ops that don't fit this?

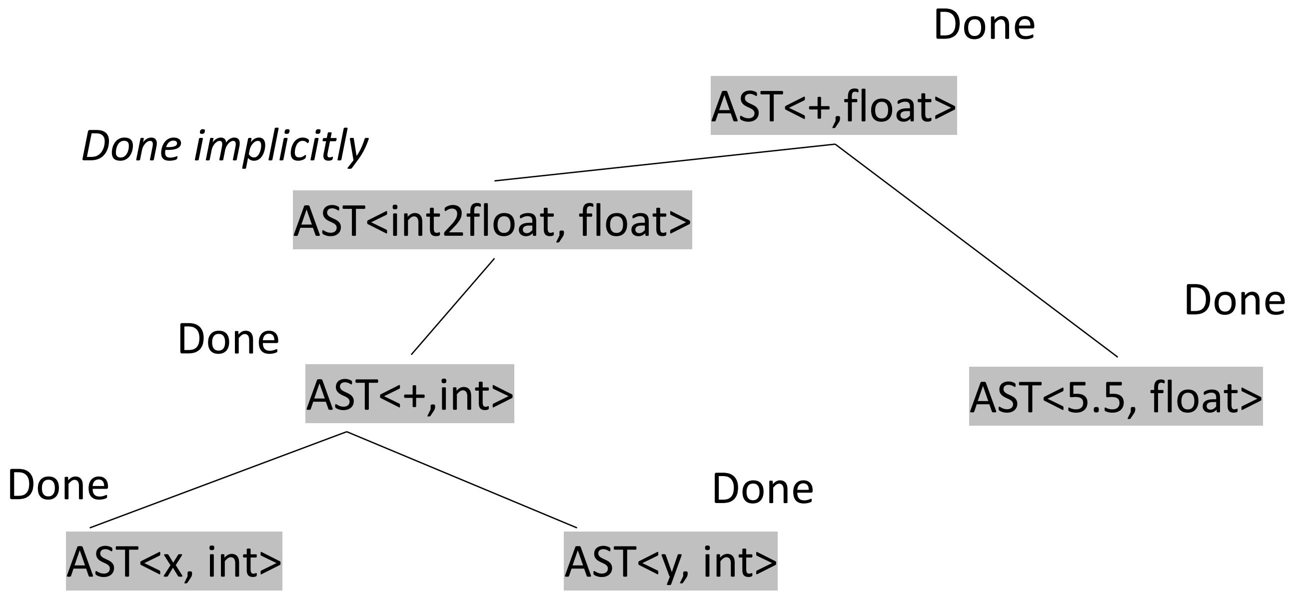
Table for assignment binary ops

left child	right child	result
`int`	`int`	`int`
`int`	`float`	`int`
`float`	`int`	`float`
`float`	`float`	`float`

Result is what is being assigned too

Type inference

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




```
def type_inference(n):

  case split on type of n:

  if n is a leaf node:
    return n.get_type()

  if n is a bin op node:
     do type inference on children
     t = lookup type from table
     set n type to t
     do any required type conversions
     return t
```

Make sure to check for special cases, like assignment!

Type errors

Table for most binary ops

left child	right child	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`
`string`	`int`	`?`
`string`	`float`	`?`

what about these?

Type errors

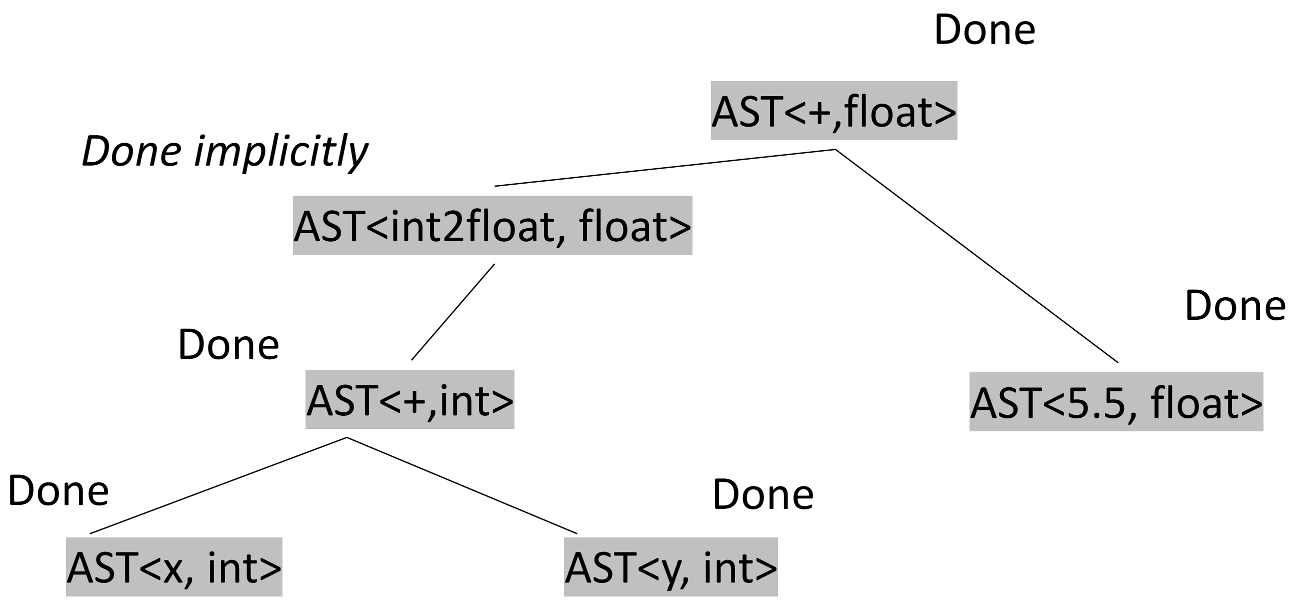
Table for most binary ops

left child	right child	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`
`string`	`int`	`ERROR (in python)` `string (in C)`
`string`	`float`	`ERROR`

char * in C

Type errors

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

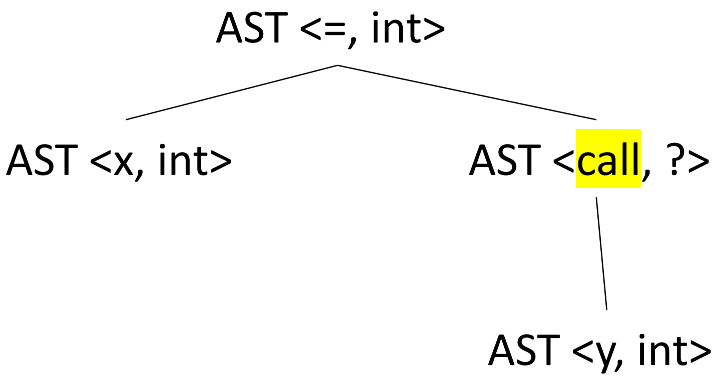



```
def type_inference(n):

    case split on type of n:

    if n is a leaf node:
      return n.get_type()

    if n is a bin op node:
       do type inference on children
       t = lookup type from table
       if t is None:
          raise typeExcpetion()
       set n type to t
       do any required type conversions
       return t
```

Table should return a flag (e.g. None) if it cannot do the conversion. We can then raise an exception

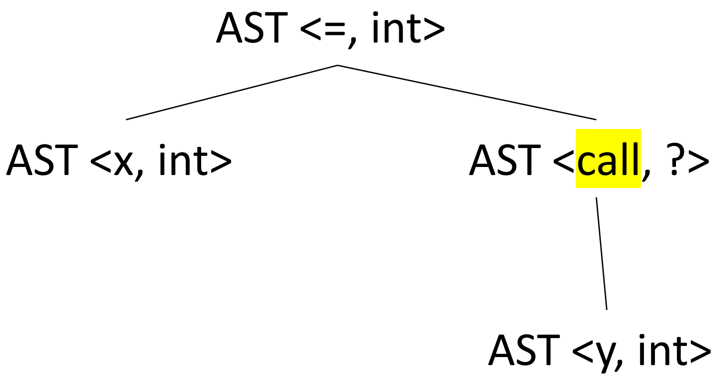
How are functions handled?

```
int x;
int y;
x = sqrt(y)
```



How are functions handled?

```
int x;
int y;
x = sqrt(y)
```



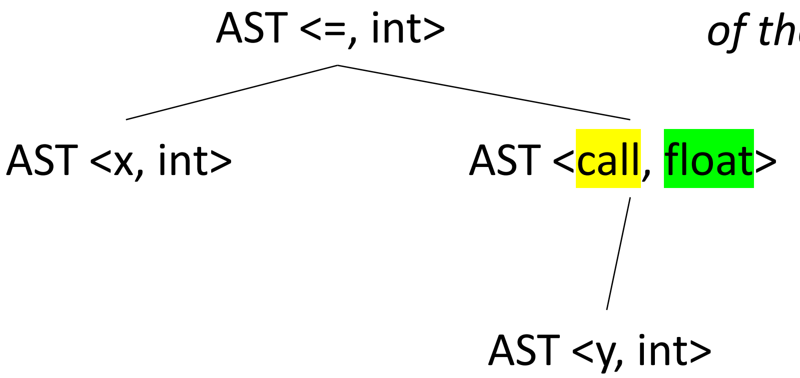
requires a function specification,
using in the .h file:

```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```



type of the AST node becomes the return type of the function

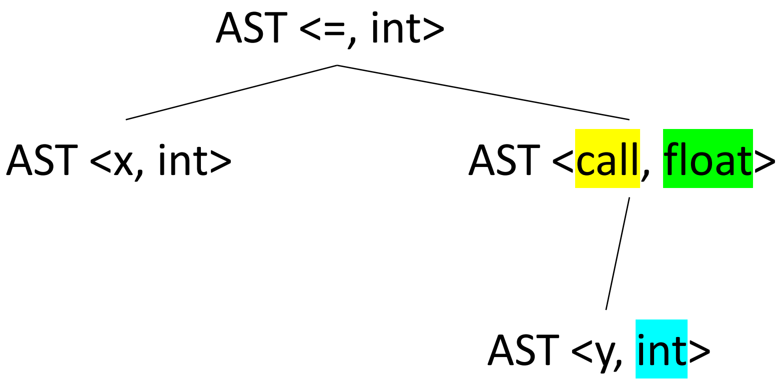
requires a function specification,
using in the .h file:

```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```



type inference must make sure arguments match types

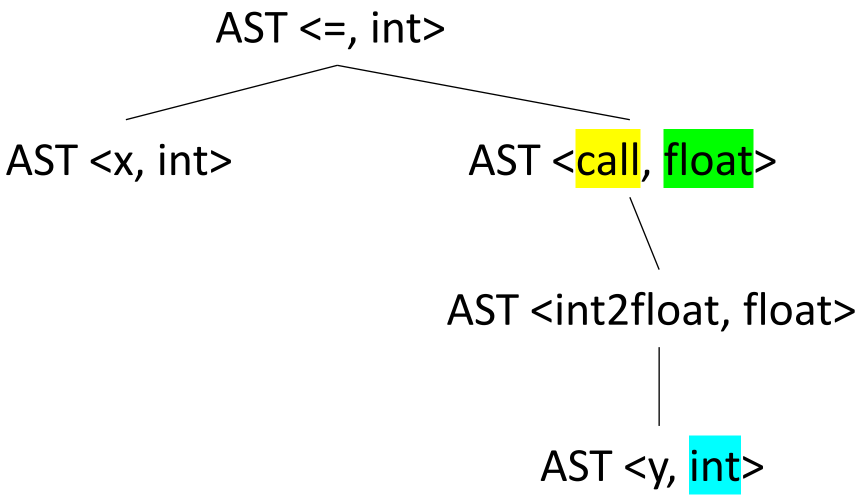
requires a function specification,
using in the .h file:

```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```



type inference must make sure arguments match types

requires a function specification,
using in the .h file:

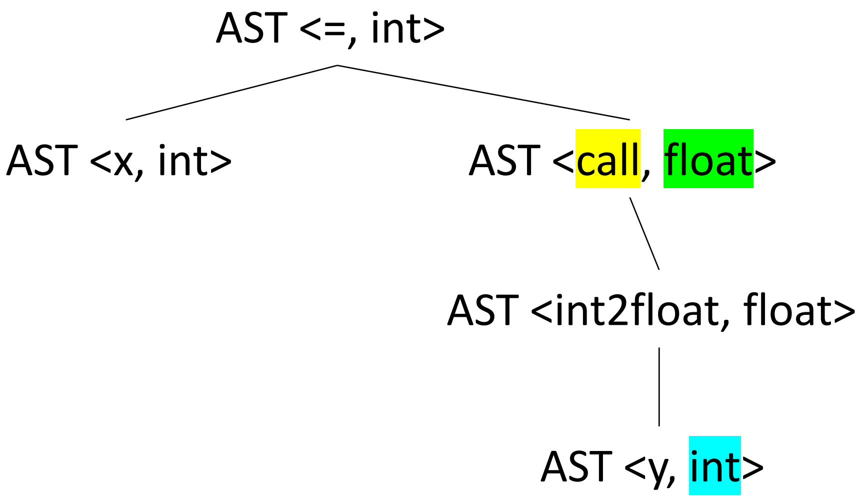
```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```

How would type inference finish this?



requires a function specification,
using in the .h file:

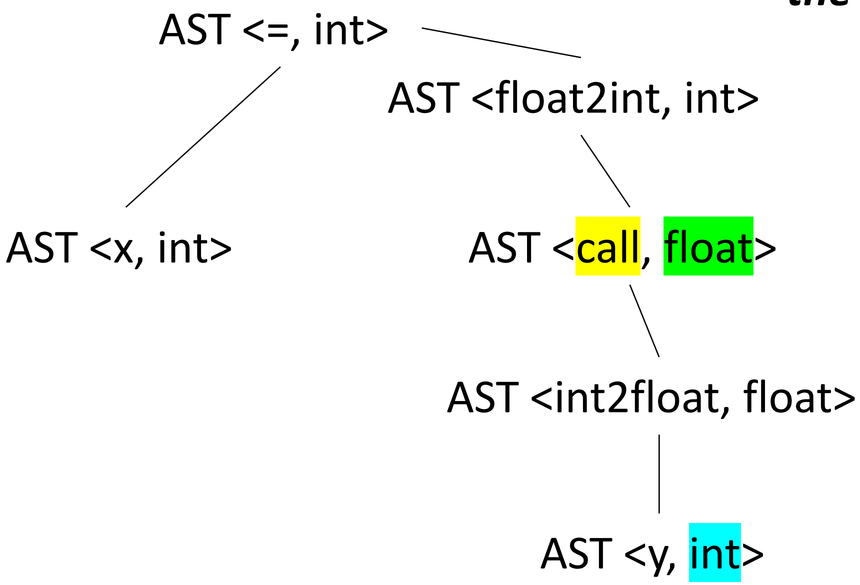
```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

How are functions handled?

```
int x;
int y;
x = sqrt(y)
```

How would type inference finish this?
remember that assignment converts to the lhs type



requires a function specification, using in the .h file:

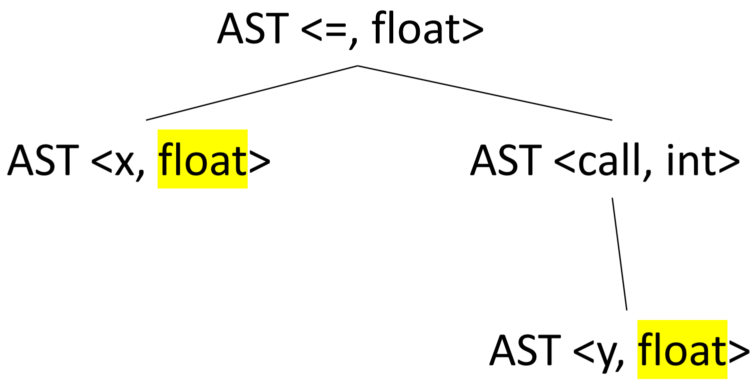
```
float sqrt(float x);
```

stored in the symbol table before type checking - think about C. you have to declare a function before you use it

What about floats to ints?

```
int int_sqrt(int input);

float x;
float y;
x = int_sqrt(y)
```

Does this compile?

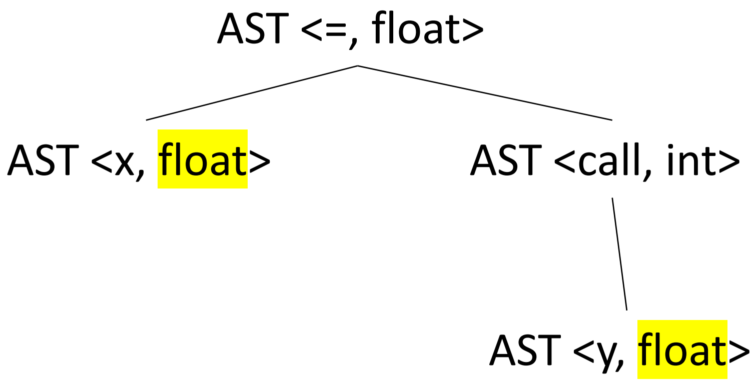


What about floats to ints?

```
int int_sqrt(int input);

float x;
float y;
x = int_sqrt(y)
```

Does this compile? Yes!

In this case the compiler will convert floats to an int.
Is that the right choice? ...

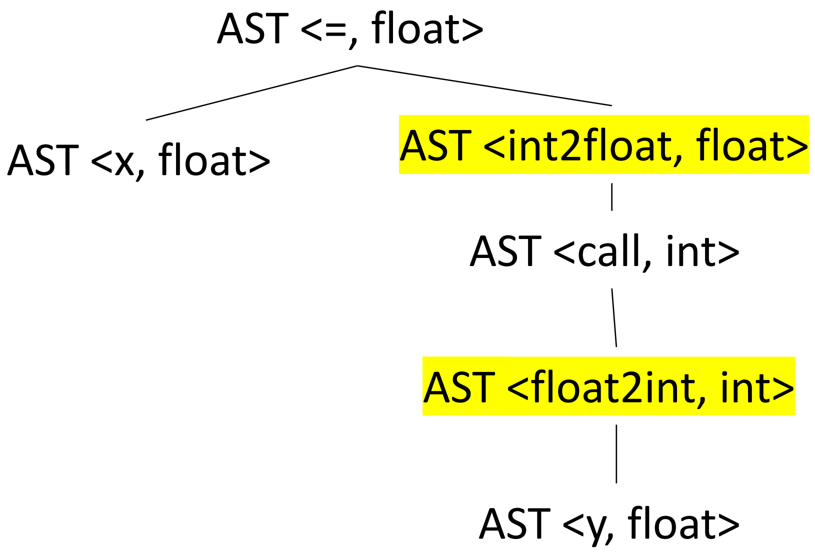


What about floats to ints?

```
int int_sqrt(int input);

float x;
float y;
x = int_sqrt(y)
```

Does this compile? Yes!

In this case the compiler will convert floats to an int.
Is that the right choice? ...



Discussion

- Many languages (and styles) state that the programmer extends the type system through functions

- Other languages allow operator overloading
  - Controversial design pattern
  - But it can be really nice (e.g. it is used extensively in LLVM internals)

```
class Complex {
  private:
   float real;
   float imag;
  public:
   // Constructor to initialize real and imag to 0
   Complex() : real(0), imag(0) {}

   // Overload the + operator
   Complex operator + (const Complex& obj) {
     Complex temp;
     temp.real = real + obj.real;
     temp.imag = imag + obj.imag;
     return temp;
   }
```

Table for ***plus*** binary ops

left child	right child	result
int	int	int
int	float	float
float	int	float
float	float	float
Complex	Complex	Complex

code from: https://www.programiz.com/cpp-programming/operator-overloading

```
class Complex {
   private:
    float real;
    float imag;
   public:
    // Constructor to initialize real and imag to 0
    Complex() : real(0), imag(0) {}

    // Overload the + operator
    Complex operator + (const Complex& obj) {
      Complex temp;
      temp.real = real + obj.real;
      temp.imag = imag + obj.imag;
      return temp;
    }

   Complex operator + (const float& i) {
       Complex temp;
       temp.real = real + i;
       temp.imag = imag;
       return temp;
     }
```

Table for ***plus*** binary ops

left child	right child	result
int	int	int
int	float	float
float	int	float
float	float	float
Complex	Complex	Complex

code from: https://www.programiz.com/cpp-programming/operator-overloading

```cpp
class Complex {
  private:
   float real;
   float imag;
  public:
   // Constructor to initialize real and imag to 0
   Complex() : real(0), imag(0) {}

   // Overload the + operator
   Complex operator + (const Complex& obj) {
     Complex temp;
     temp.real = real + obj.real;
     temp.imag = imag + obj.imag;
     return temp;
   }

  Complex operator + (const float& i) {
      Complex temp;
      temp.real = real + i;
      temp.imag = imag;
      return temp;
    }
```

Table for ***plus*** binary ops

left child	right child	result
int	int	int
int	float	float
float	int	float
float	float	float
Complex	Complex	Complex
Complex	float	Complex

We can add extra rows and even conversions

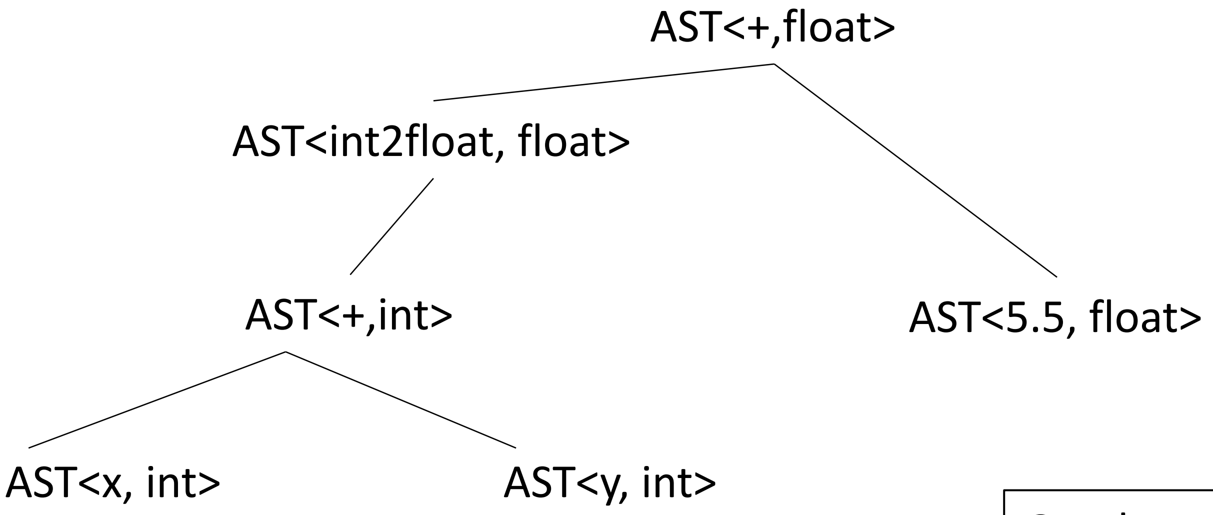
code from: https://www.programiz.com/cpp-programming/operator-overloading

Type systems finished

- Defined what a type system is and discussed various different design decisions
  - static vs. dynamic, choice of primitive types, size of primitive types
- Implemented type inference parameterized by type conversion tables on an AST.
  - identified common conversions (int to float) and when the opposite can happen
- Discussed how programmers can extend the type system
  - function calls
  - operator overloading

Linear intermediate representations

- So far, we've been looking at graph representations
- Linear IRs are a linear sequence of instructions, similar to assembly



Graph representation

```
vr0 = addi(x,y);
vr1 = int2float(vr0);
vr2 = addf(vr1,5.5);
```

Linear representation

3-address code

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

In this class we will focus on 3 address code

- By address, we don’t mean “memory address”. We mean virtual registers. Several formats

3-address code

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

- By address, we don’t mean “memory address”. We mean virtual registers. Several formats

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
```

3-address code

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

- By address, we don't mean "memory address". We mean virtual registers. Several formats

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
```

Conceptually it should be clear what each one is doing and we may switch depending on the example

3-address code

- Several types of linear code:
  - 1 address code
  - 2 address code
  - 3 address code
- By address, we don’t mean “memory address”. We mean virtual registers. Several formats

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
```

three address as each instruction has roughly 3 addresses: 1 destination and 2 operands

3-address code

Different designs have different trade offs and different information carried with it

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

- By address, we don't mean "memory address". We mean virtual registers. Several formats

book

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

no types

this class

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

type: i = integer, f = float

LLVM IR

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

type in instruction

3-address code

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

Unlimited virtual registers

- By address, we don't mean "memory address". We mean virtual registers. Several formats

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
```

3-address code

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

What about these others?

- By address, we don’t mean “memory address”. We mean virtual registers. Several formats

3-address code

- 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

- By address, we don’t mean “memory address”. We mean virtual registers. Several formats

```
push 2
push b
multiply
push a
subtract
```

3-address code

- 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

- By address, we don't mean "memory address". We mean virtual registers. Several formats

```
push 2
push b
multiply
push a
subtract
```

Execute this code as an exercise

3-address code

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

Not really used these days

- By address, we don’t mean “memory address”. We mean virtual registers. Several formats

3-address code

- Several exceptions to the 3 in the 3-address code

```
// memory loads
vr0 = load(x)

// memory stores
store(x,5);

// function calls
vr2 = foo(x,y,z,w)
```

but it is a best-effort attempt to capture the code in a semi-readable form close to an ISA

3-address code

What does this code do?

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

```
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(x,y);
  vr1 = multi(5,7);
  vr2 = divi(vr0,vr1);
  bne vr2 0 label0;
  vr3 = ...
  vr4 = ...
```

Our 3-address code

The 3 address code we will be targeting with our homework and using for optimizations in the next module

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);
```

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!
```

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

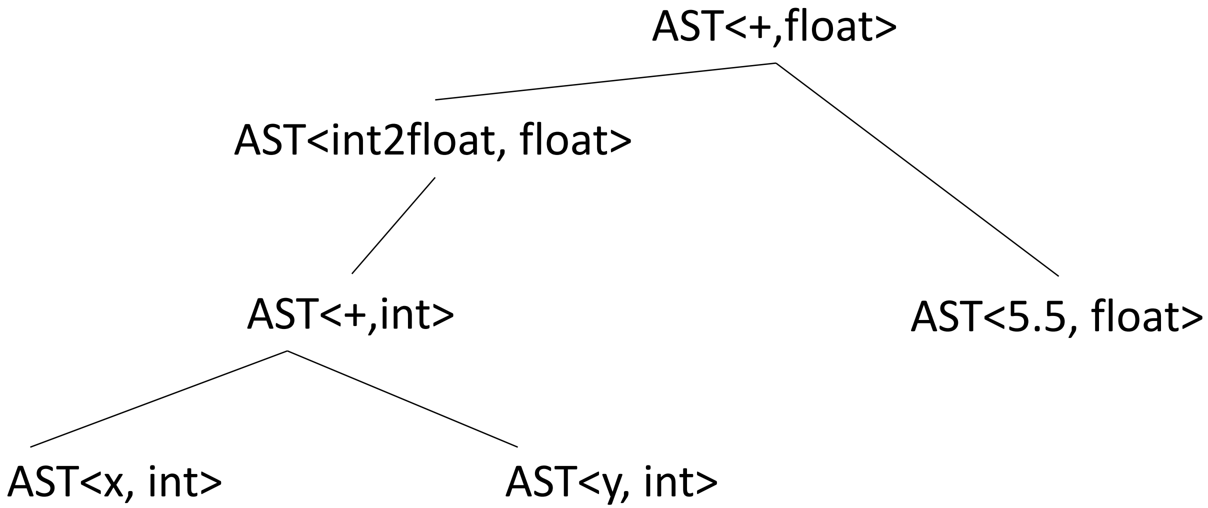
```
  vr0 = int2vr(1);
  vr1 = int2vr(1);
  vr2 = int2vr(10);
loop_start:
  vr3 = lti(vr0, vr2);
  bne(vr3, vr1, end_label);
  vr4 = int2vr(x);
  vr5 = addi(vr4,vr0);
  x = vr2int(vr5);
  vr0 = addi(vr0, vr1);
  branch(loop_start);
end_label:
```

Converting AST into Class-IR

Converting AST into Class-IR

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

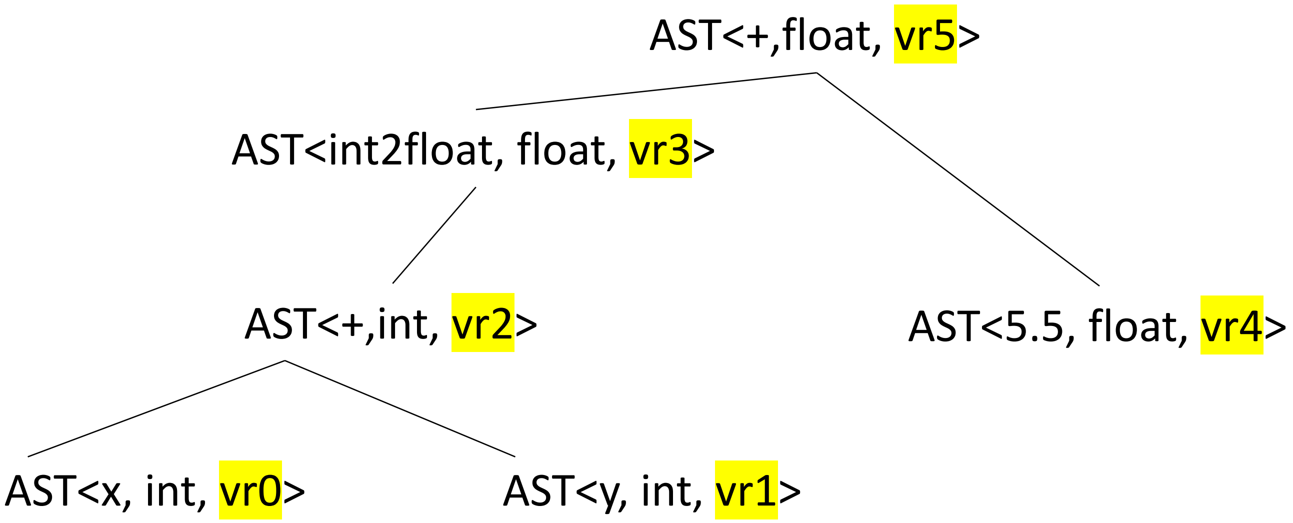
After type inference



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

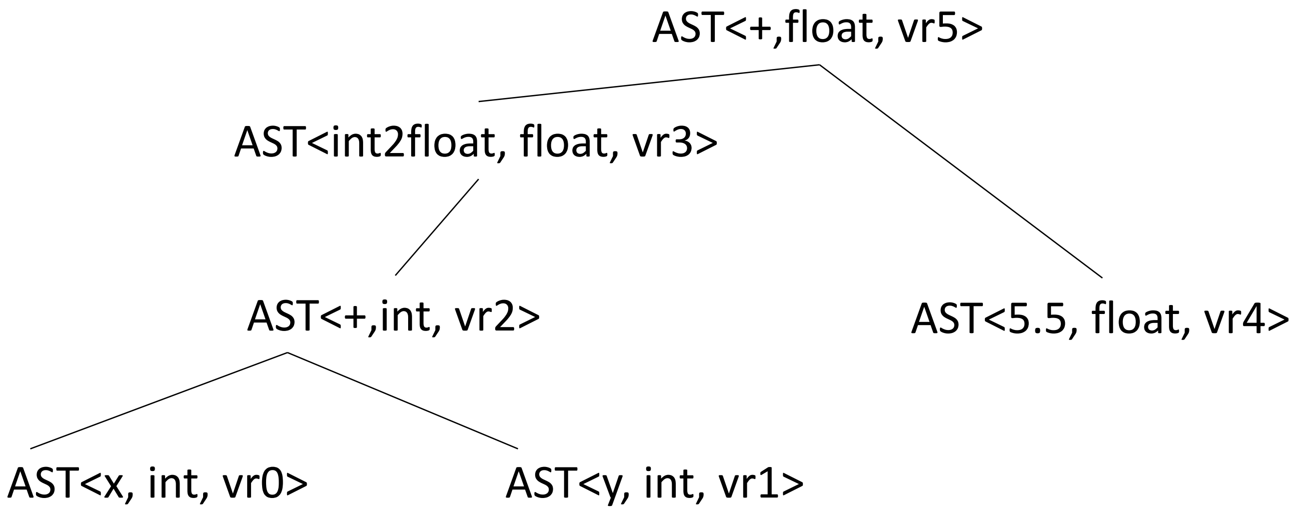
Each node needs a distinct virtual register

Converting AST into Class-IR

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

After type inference

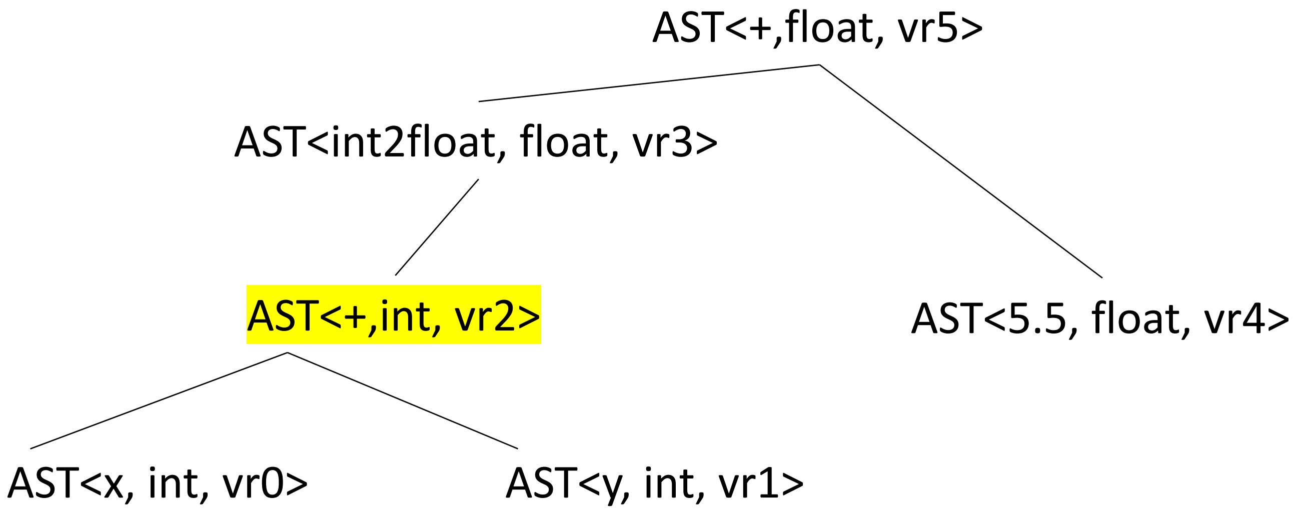
Next each AST node needs to know how to print a 3 address instruction



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

```python
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):
       ??
```

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

```python
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):
      ??
```

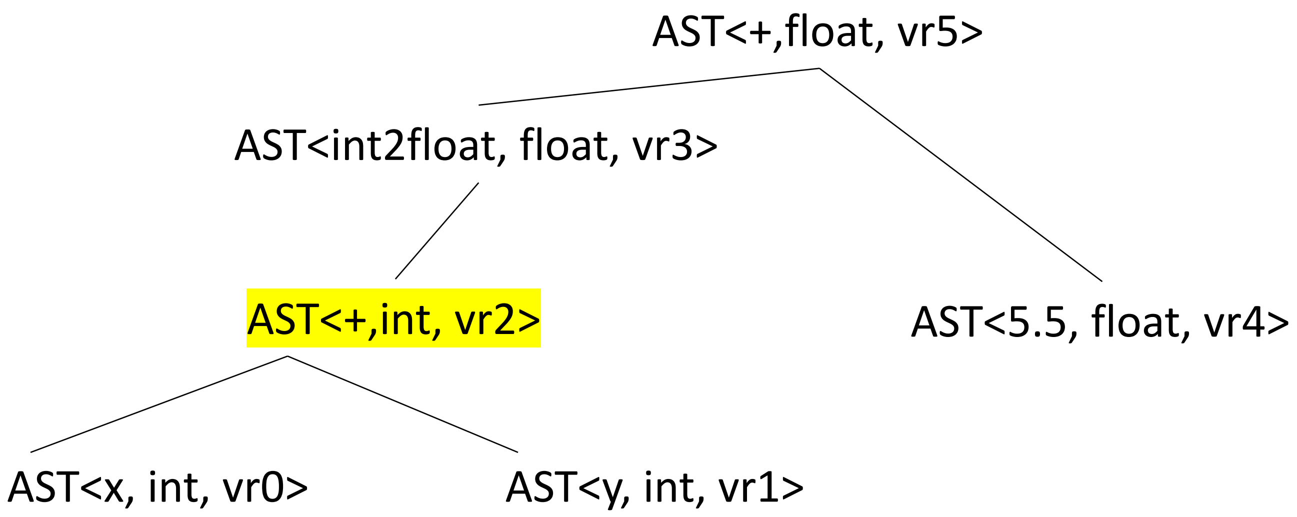
```python
return "%s = %s(%s,%s);" %
                  (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

What is this one?

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

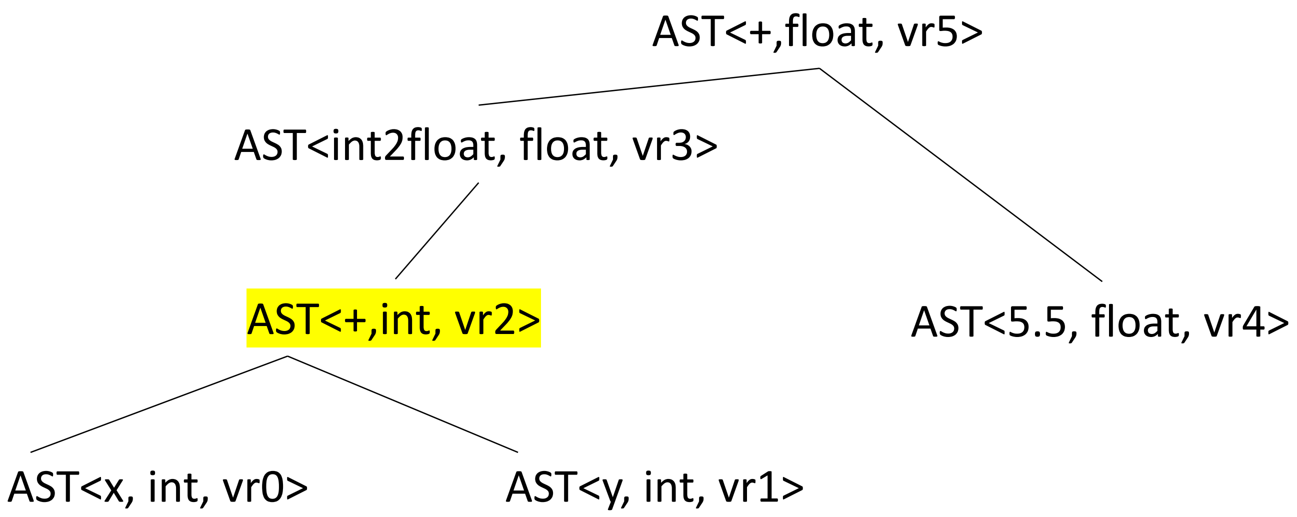
```python
return "%s = %s(%s,%s);" %
        (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

What is this one?




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

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

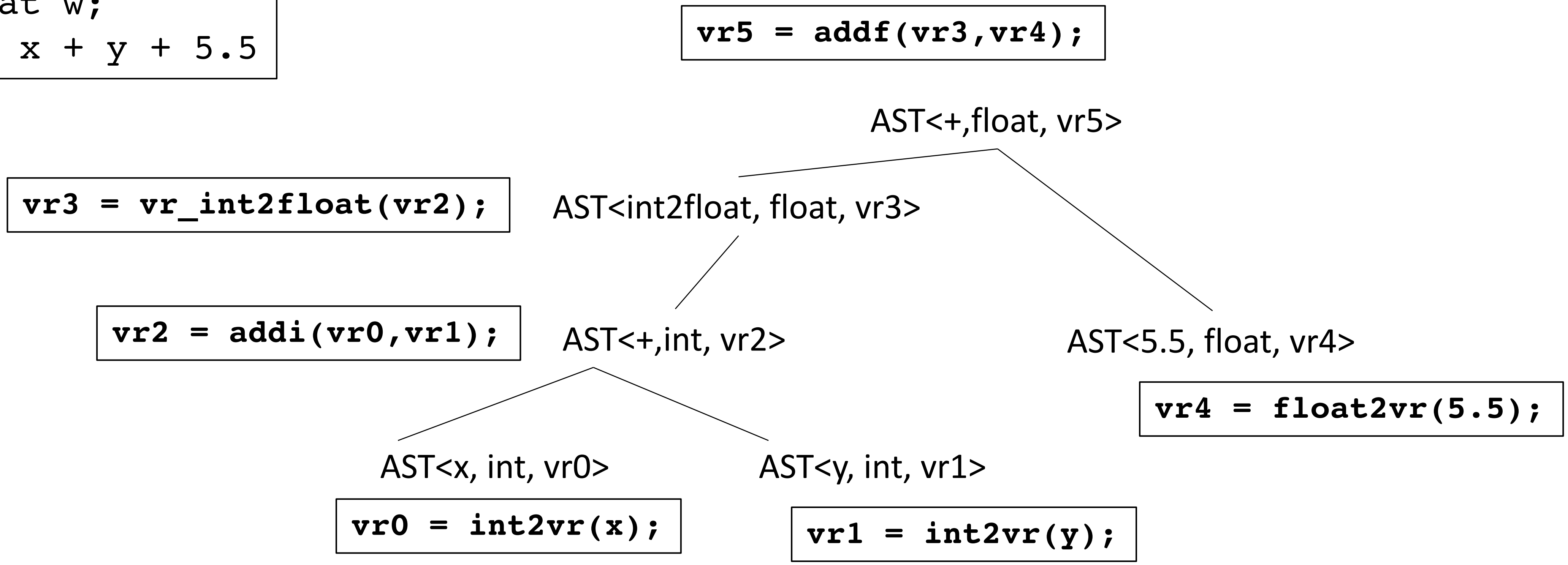



```python
def get_op(self):
        if self.node_type is Types.INT:
            return "addi"
        else:
            return "addf"
```

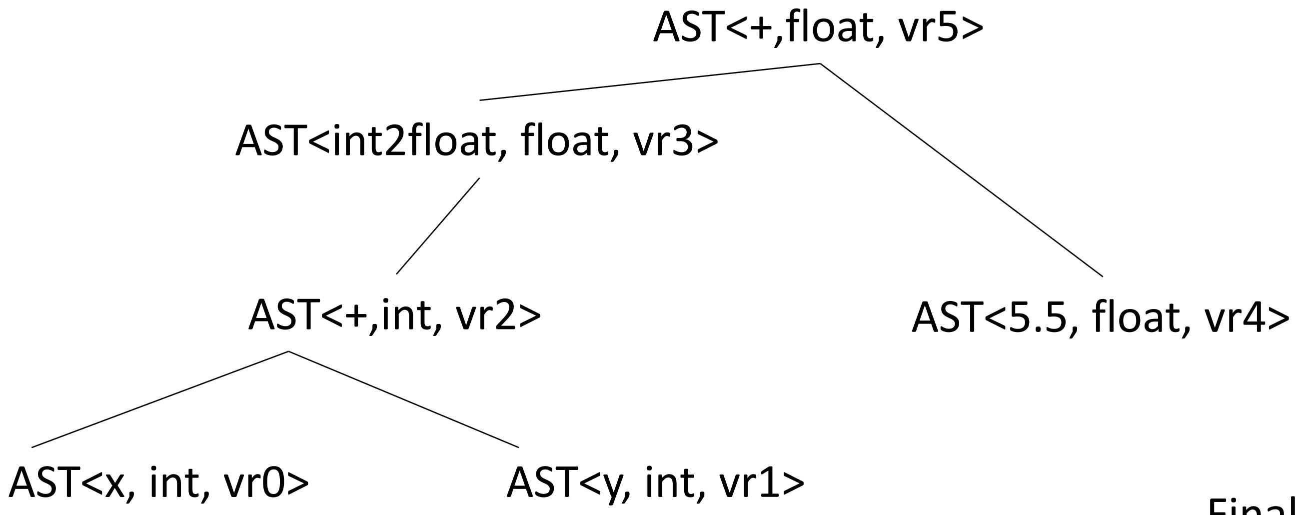
```python
return "%s = %s(%s,%s);" %
                (self.vr, self.get_op(), self.l_child.vr, self.r_child.vr)
```

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

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



```
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);
```

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 statement

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 statement

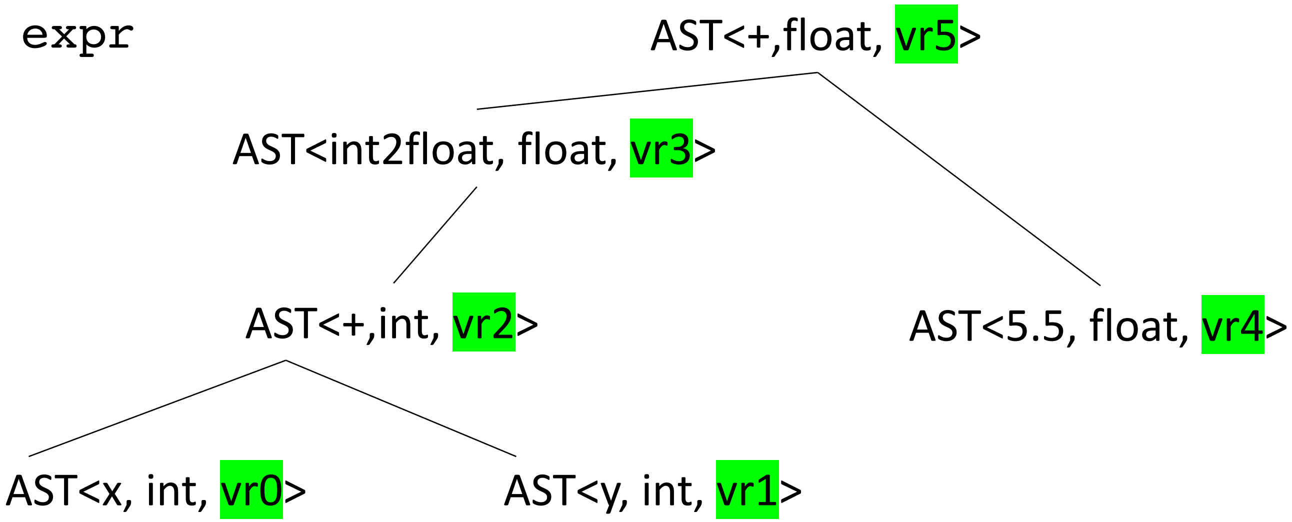
```
int x;
int y;
float w;
w = x + y + 5.5
```

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

```
{
   id_name = to_match.value
   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
```

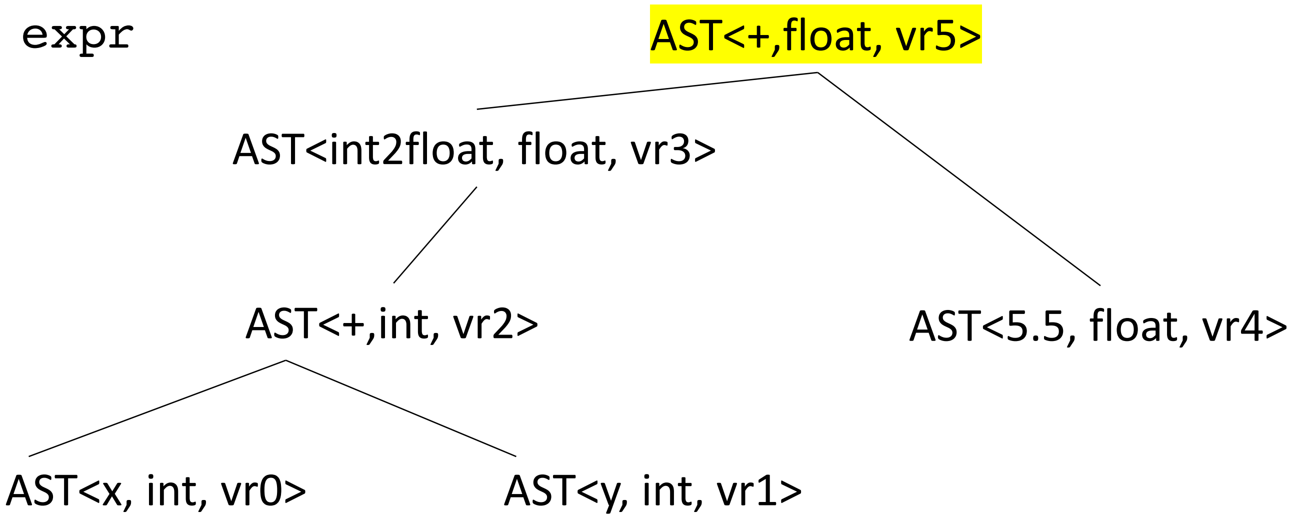



```
{
   id_name = to_match.value
   eat("ID");
   eat("ASSIGN");
   ast = parse_expr()
   type_inference(ast)
   assign_registers(ast)
   program = ast.linearize()
   new_inst = "%s = %s" % ?
   return program + [new_inst]
}
```

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

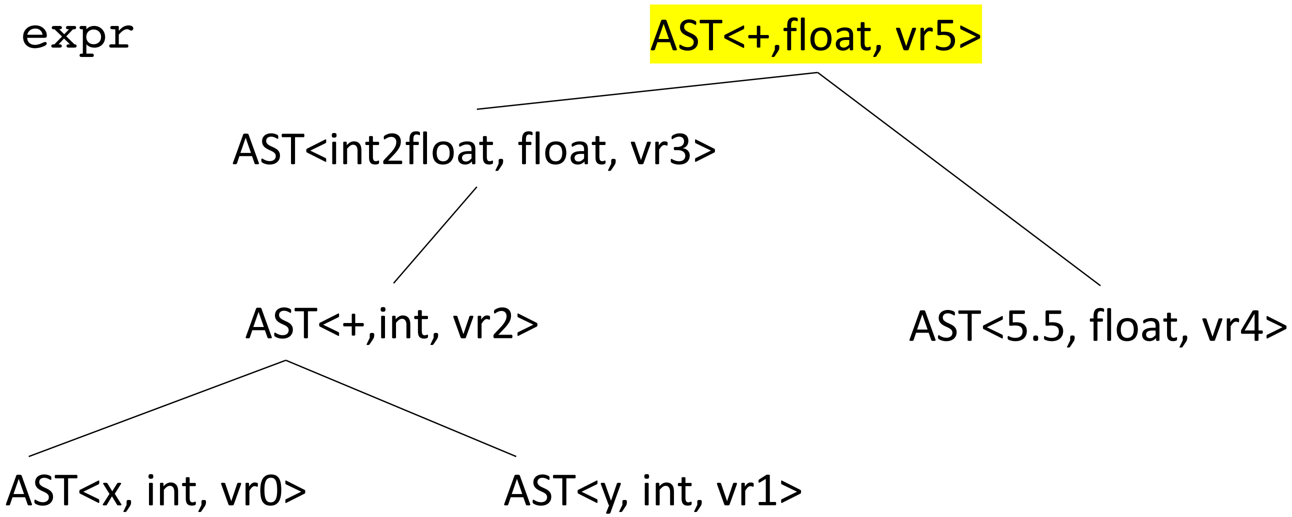
```
assignment_statement_base := ID ASSIGN expr
```

```
{
    id_name = to_match.value
    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.value
    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.value
    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
```

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

```
assignment_statement_base := ID ASSIGN expr



 {
    id_name = to_match.value
    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]
 }
```

it can get a little messy

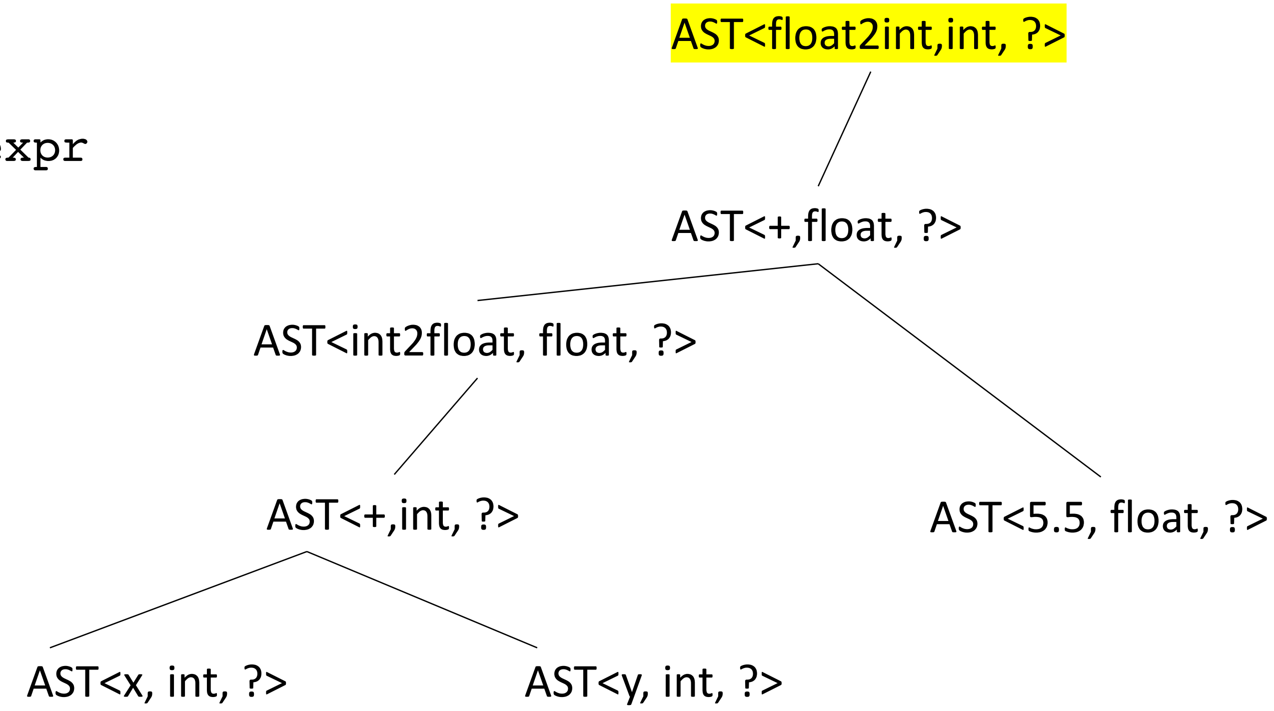
```
int x;
int y;
int w;
w = x + y + 5.5
```

```
assignment_statement_base := ID ASSIGN expr

 {
    id_name = to_match.value
    id_data_type = # get ID data type
    eat(“ID”);
    eat(“ASSIGN”);
    ast = parse_expr()
    type_inference(ast)
    if id_data_type == INT and
               ast.node_type == FLOAT:
       ast = ASTFloatToInt(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = “%s = %s” % (id_name, ast.vr)
    return program + [new_inst]
 }
```

one possible case

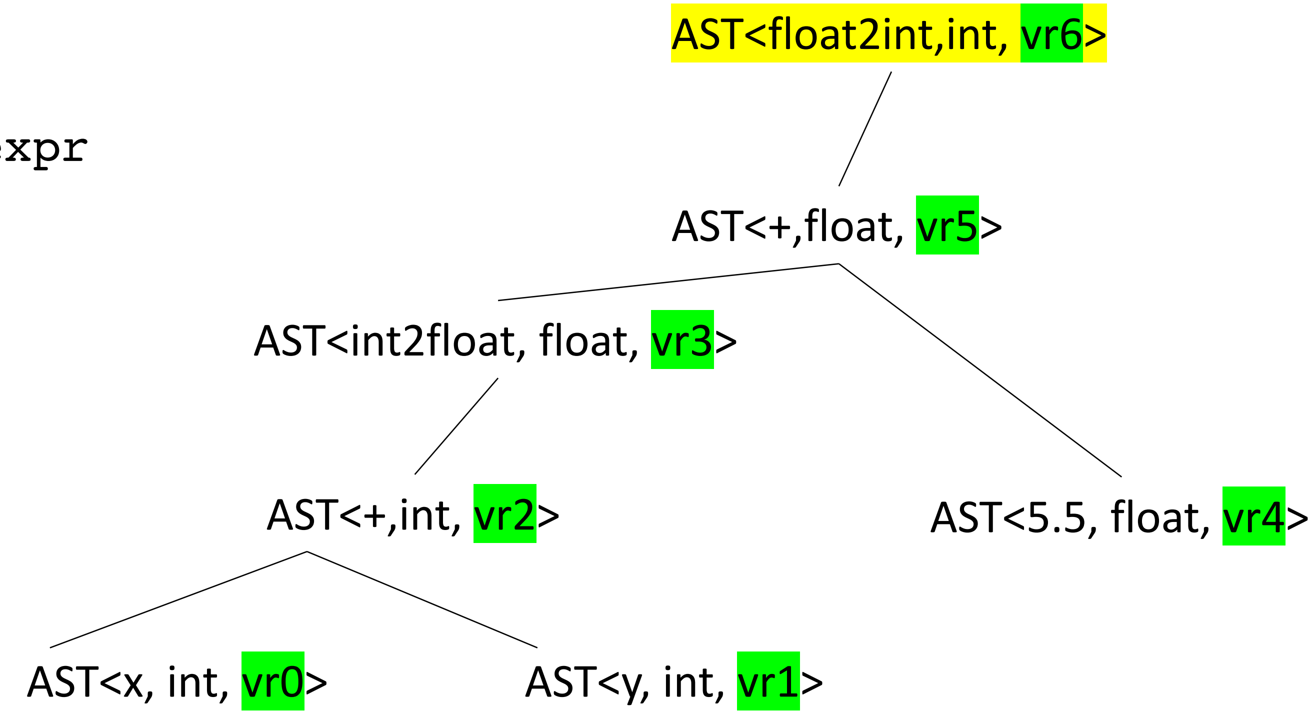
```
int x;
int y;
int w;
w = x + y + 5.5
```

```
assignment_statement_base := ID ASSIGN expr

 {
    id_name = to_match.value
    id_data_type = # get ID data type
    eat(“ID”);
    eat(“ASSIGN”);
    ast = parse_expr()
    type_inference(ast)
    if id_data_type == INT and
              ast.node_type == FLOAT:
         ast = ASTFloatToInt(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = “%s = %s” % (id_name, ast.vr)
    return program + [new_inst]
 }
```



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

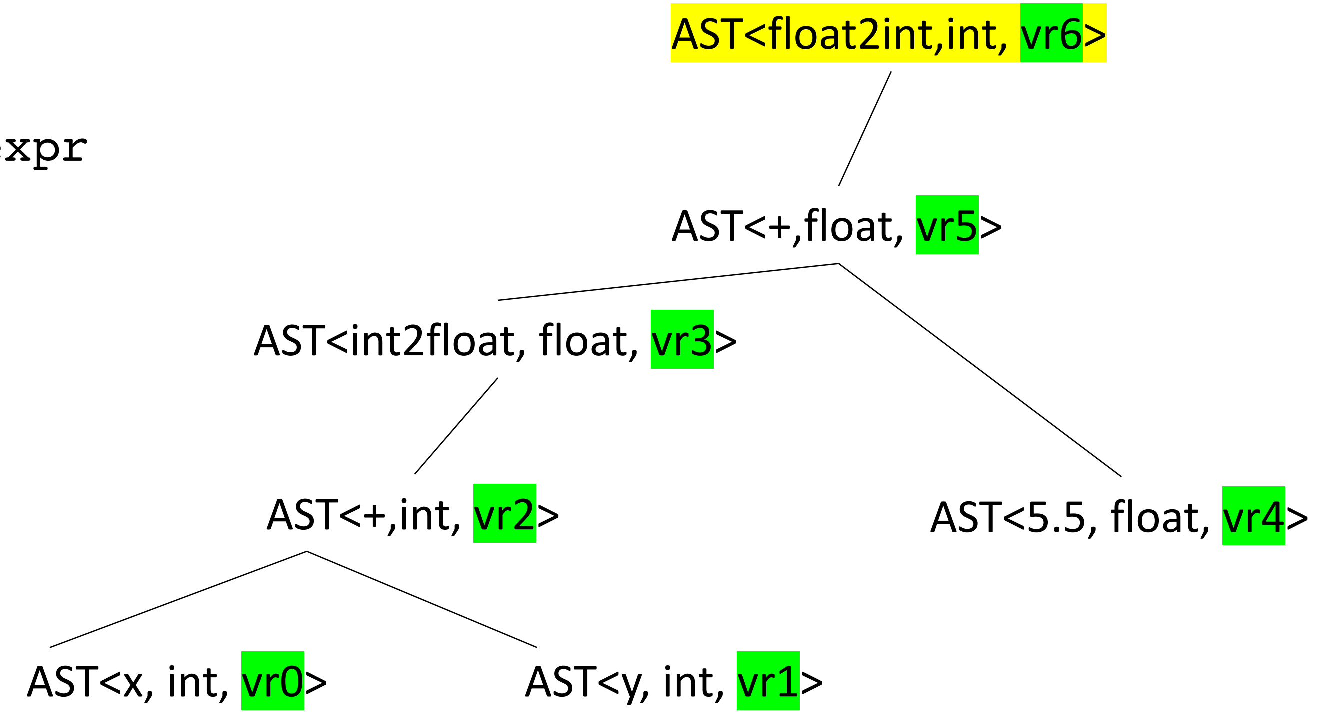
```
assignment_statement_base := ID ASSIGN expr

{
    id_name = to_match.value
    id_data_type = # get ID data type
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    if id_data_type == INT and
                   ast.node_type == FLOAT:
          ast = ASTFloatToInt(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = %s" % (id_name, ast.vr)
    return program + [new_inst]
}
```



```
(IO: int w)
int x;
int y;
w = x + y + 5.5
```

How would we deal with w as an IO variable?

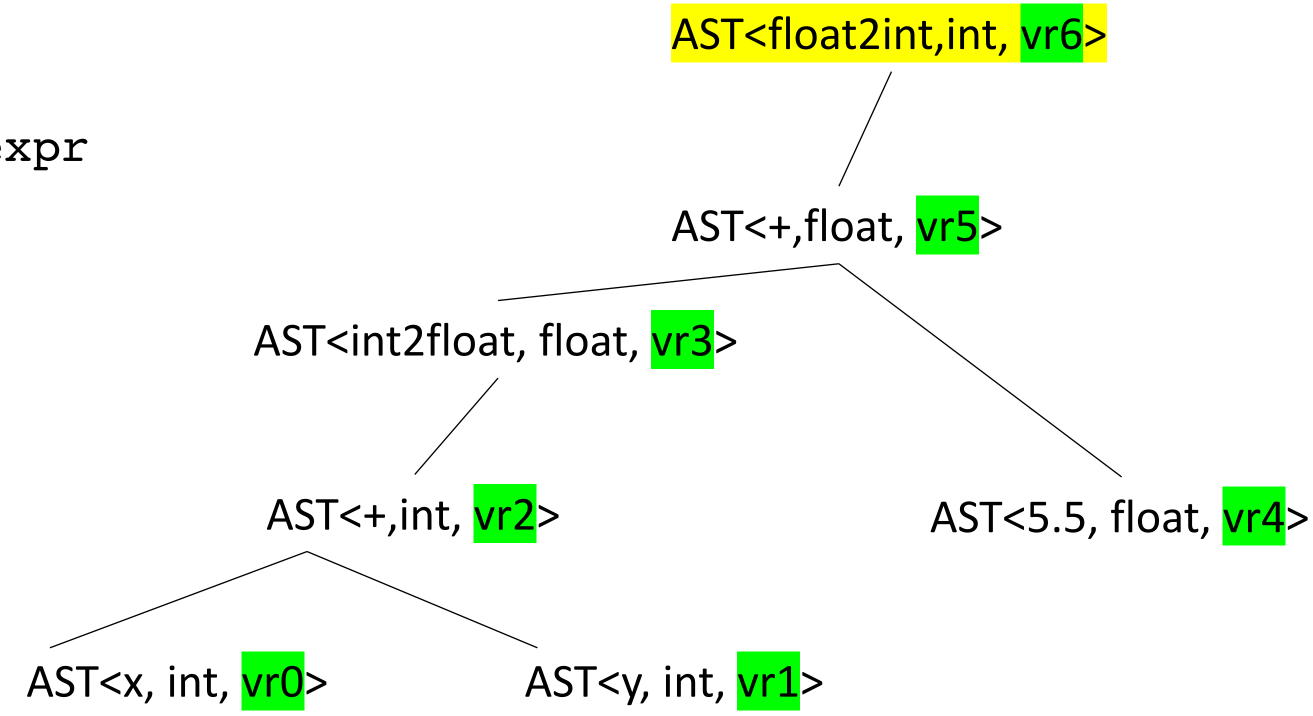
```
assignment_statement_base := ID ASSIGN expr

 {
    id_name = to_match.value
    id_data_type = # get ID data type
    eat(“ID”);
    eat(“ASSIGN”);
    ast = parse_expr()
    type_inference(ast)
    if id_data_type == INT and
                   ast.node_type == FLOAT:
           ast = ASTFloatToInt(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = “%s = %s” % (id_name, ast.vr)
    return program + [new_inst]
 }
```



```
(IO: int w)
int x;
int y;
w = x + y + 5.5
```

How would we deal with w as an IO variable?

```
assignment_statement_base := ID ASSIGN expr

 {
    id_name = to_match.value
    id_data_type = # get ID data type
    eat("ID");
    eat("ASSIGN");
    ast = parse_expr()
    type_inference(ast)
    if id_data_type == INT and
                ast.node_type == FLOAT:
        ast = ASTFloatToInt(ast)
    assign_registers(ast)
    program = ast.linearize()
    new_inst = "%s = vr2int(%s)" % (id_name, ast.vr)
    return program + [new_inst]
 }
```



Only if it is an IO variable!

It gets a little messy

Let’s do another one

```
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");
    expr_ast = parse_expr()
    ...
    program0 = # type safe and linearized ast
    eat("RPAR");
    program1 = parse_statement()
    eat("ELSE")
    program2 = parse_statement()
    ...
}
```

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

We need to convert this to 3 address code

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

{
    eat("IF");
    eat("LPAR");
    expr_ast = parse_expr()
    ...
    program0 = # type safe and linearized ast
    eat("RPAR");
    program1 = parse_statement()
    eat("ELSE")
    program2 = parse_statement()
    ...
}
```

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

We need to convert this to 3 address code

```
program0
program1
program2
```

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

{
    eat("IF");
    eat("LPAR");
    expr_ast = parse_expr()
    ...
    program0 = # type safe and linearized ast
    eat("RPAR");
    program1 = parse_statement()
    eat("ELSE")
    program2 = parse_statement()
    ...
}
```

```
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:
```

```
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:
```

```
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)]
}
```

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


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

```
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)]
}
```

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


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

```
 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)]
}
```

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

Need a :

```
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()
                                                        def label_code(l): return l + ":"
   # 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)]
}
```

```
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!

See everyone on Wednesday

- Continue will discussing transforming an AST into linear code