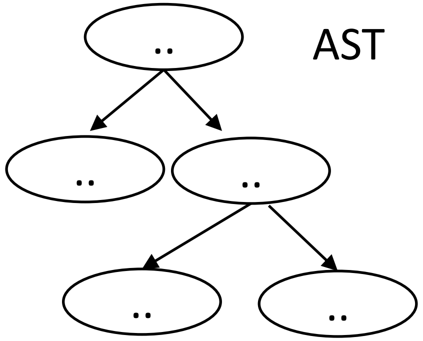
CSE110A: Compilers

May 4, 2022

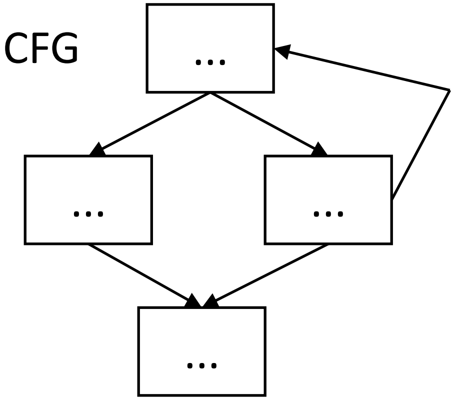


AST

Topics:

- *Finish up type checking*
- *3-address code*

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 1 grades are released
  - Let us know by Monday if there are any issues
  - Please let us know through a private piazza post
- Midterm is out
  - I have started a piazza note with clarifications
  - do not discuss with classmates at all, do not ask (or search for) exact questions online
  - No late midterms will be accepted (please prioritize it)

Announcements

- I released the life preserve for HW 2 on midnight
  - part 1 grammar with the first+ sets
  - this should help with part 2 and 3 if you were stuck on part 1
  - late policy still applies and it won’t be accepted past Thursday
  - regardless of circumstance, the midterm will not be accepted late, so please budget your time accordingly.
- Expect HW 3 on Monday by midnight
  - It will be similar to HW 2 in terms of workload and conceptual depth

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

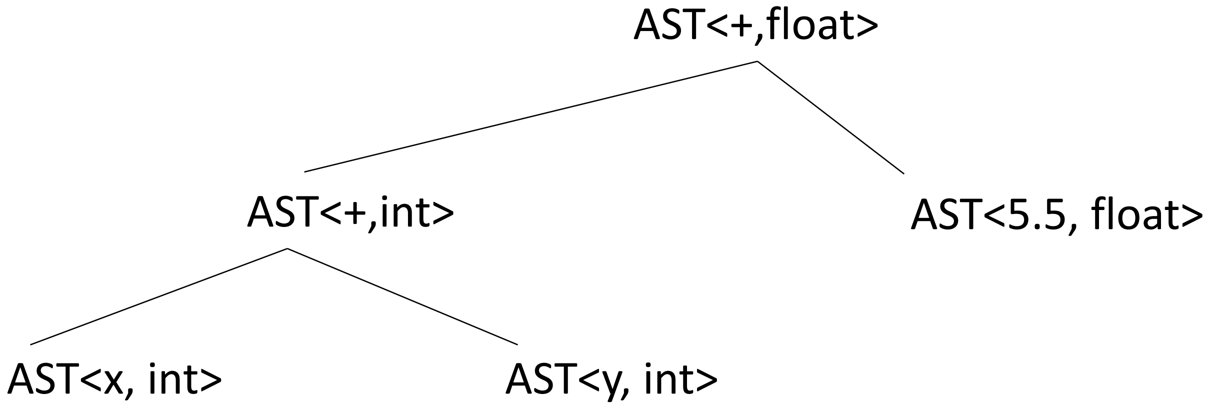
Discussion

- Definition of expression: it returns a value. If it has a value, then it has a type
  - Possible exceptions?
- 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)
```

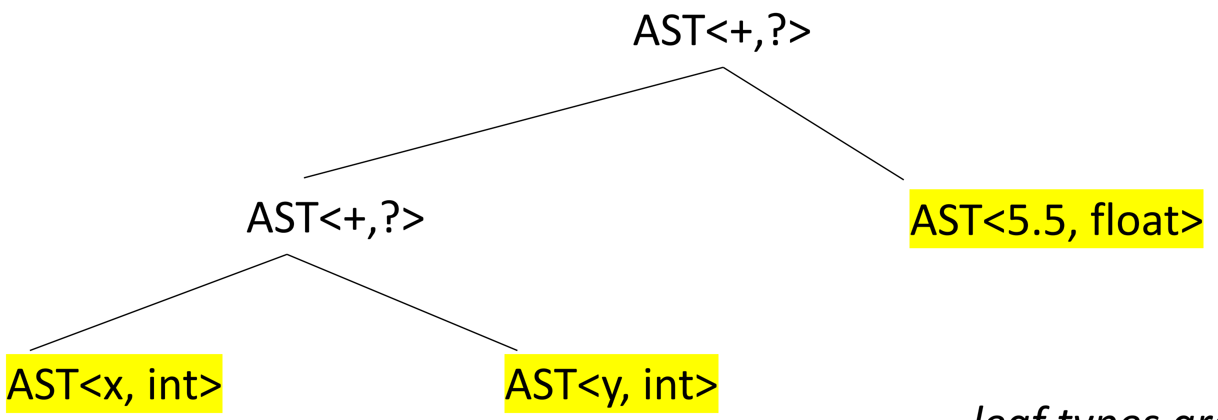
```
class ASTIDNode(ASTLeafNode):
    def __init__(self, value, value_type):
        super().__init__(value)
        self.set_type(value_type)
```

Review

- For the review, we will walk through the type inference algorithm and discuss some new additions to it.

Type inference

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

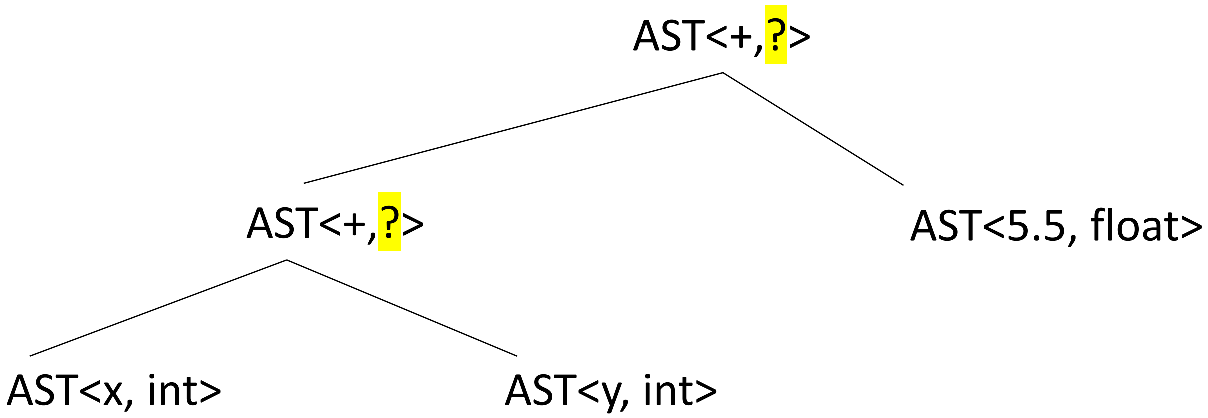


leaf types are provided on construction

Type inference

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

We need to find these types. Why?



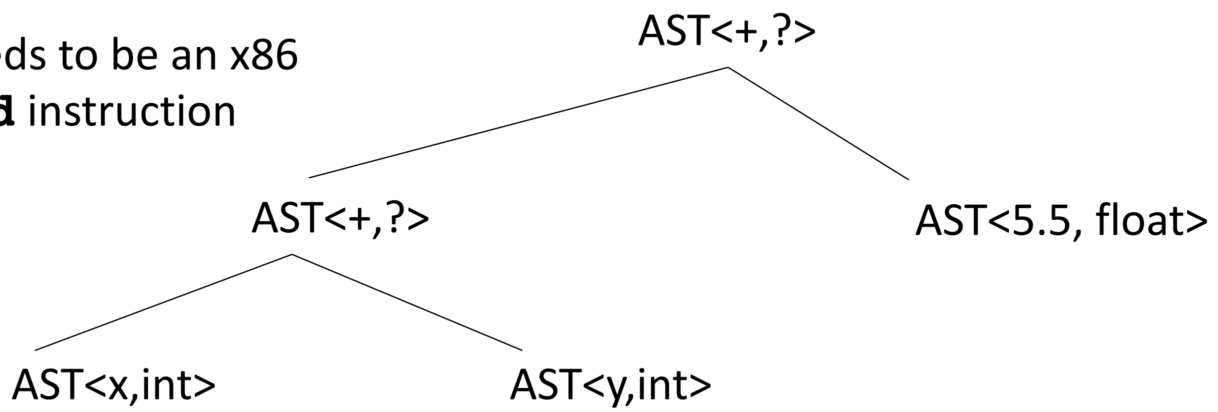
Type inference

Recall the example of: 5 + 5.0

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

needs to be an x86 **addss** instruction

needs to be an x86 **add** instruction

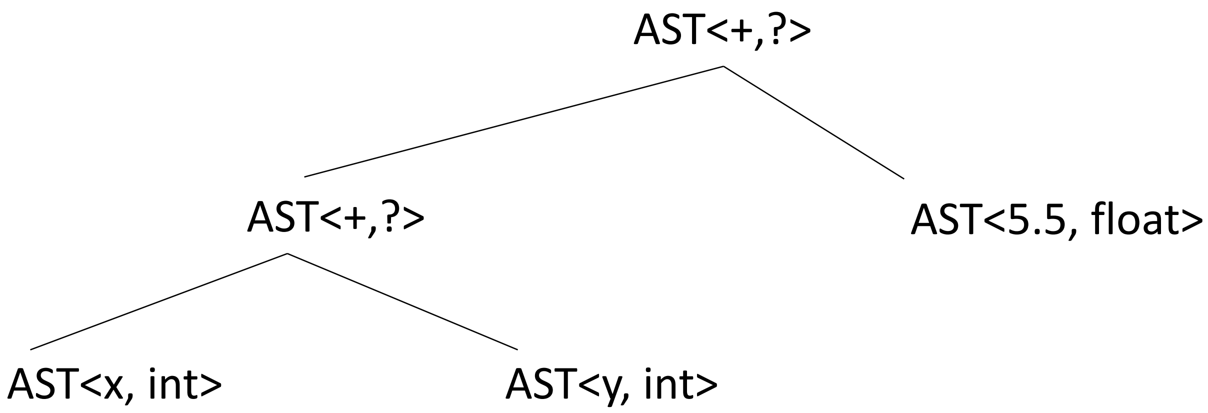


add r0 r1 – interprets the bits in the registers as integers and adds them together

addss r0 r1 – interprets the bits in the registers as floats and adds them together

Type inference

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

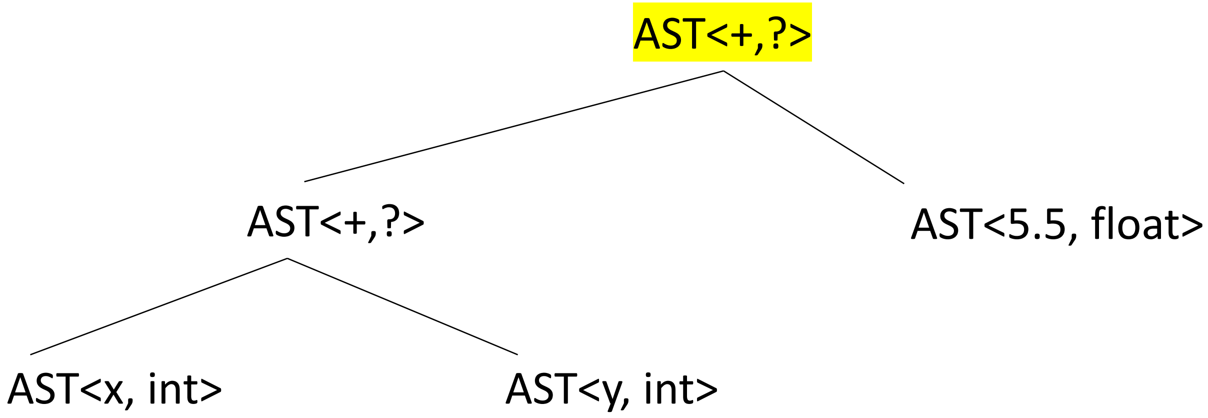



```
def type_inference(n):

    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
       return t
```

Type inference

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

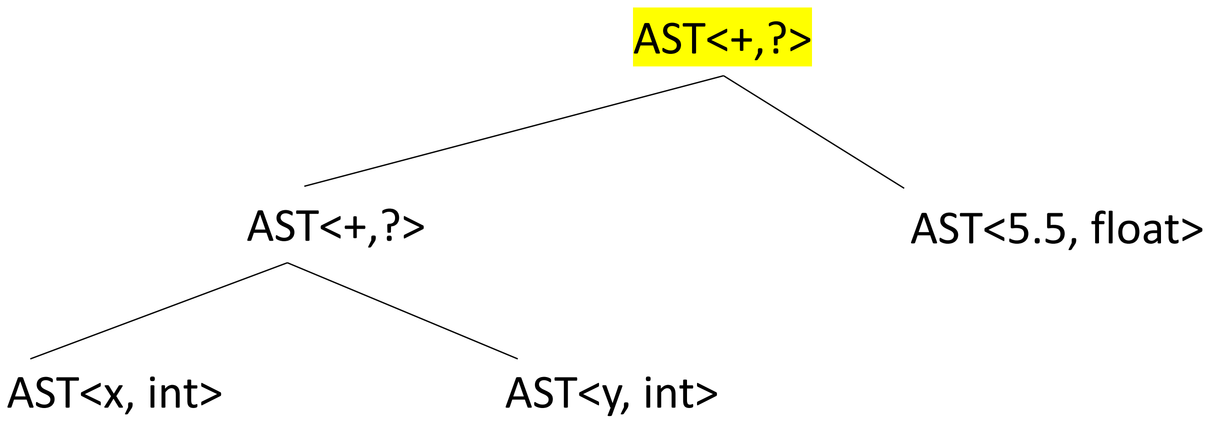
start on top




```
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
       return t
```

Type inference

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

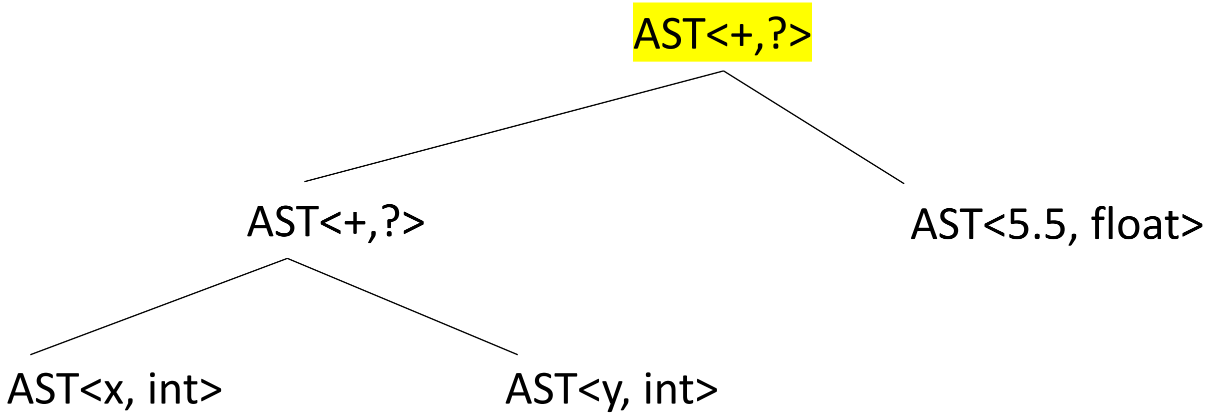


it's a binary op

```
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
     return t
```

Type inference

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

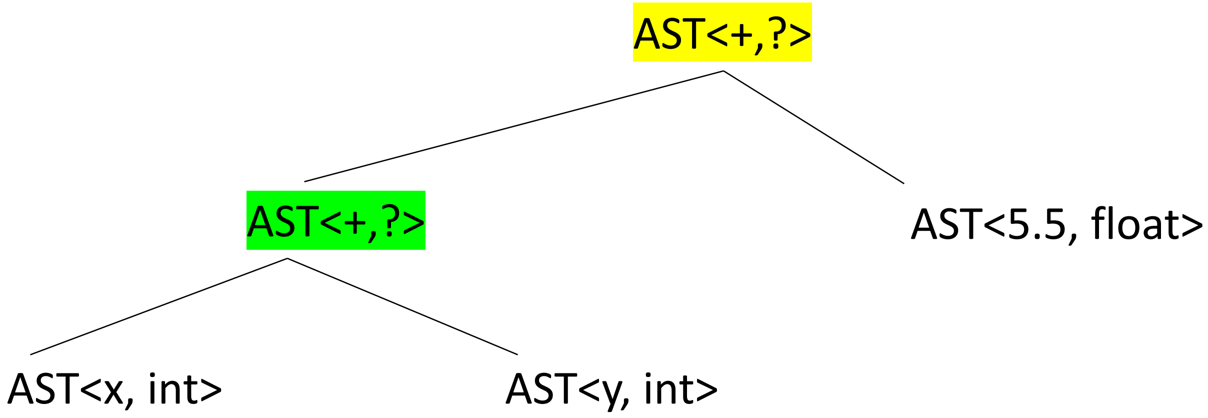


recursion

```
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
     return t
```

Type inference

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



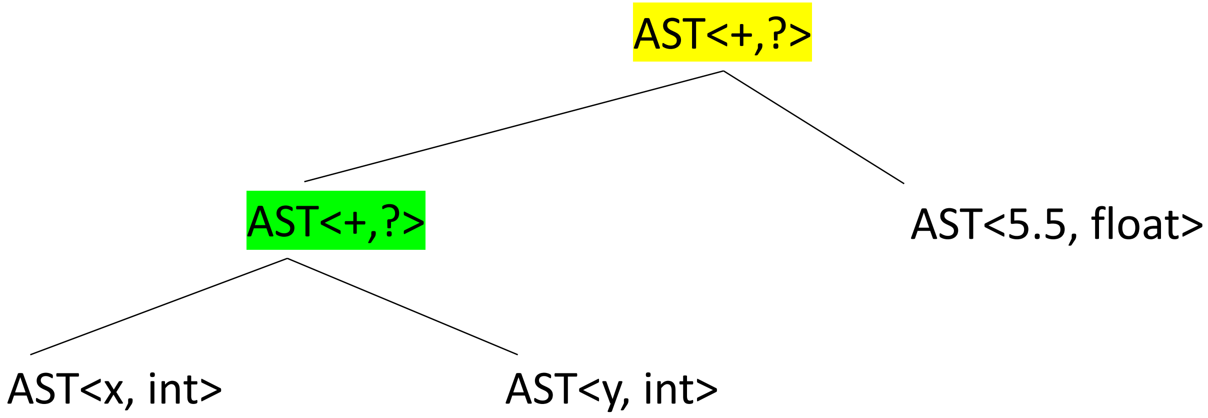

```
def type_inference(n):

    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
       return t
```

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
       return t
```

it's a binary op

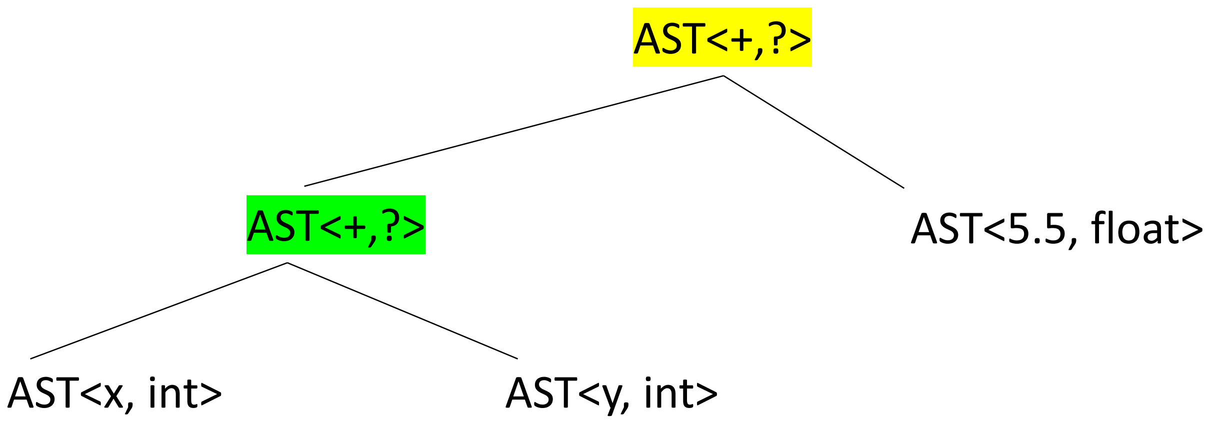


Type inference

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

recursion

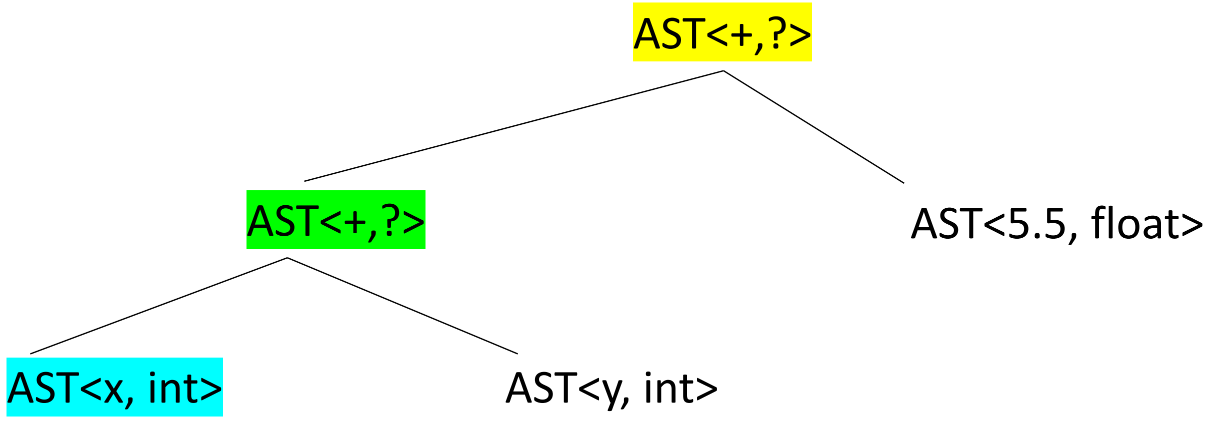
```
def type_inference(n):

  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
     return t
```



Type inference

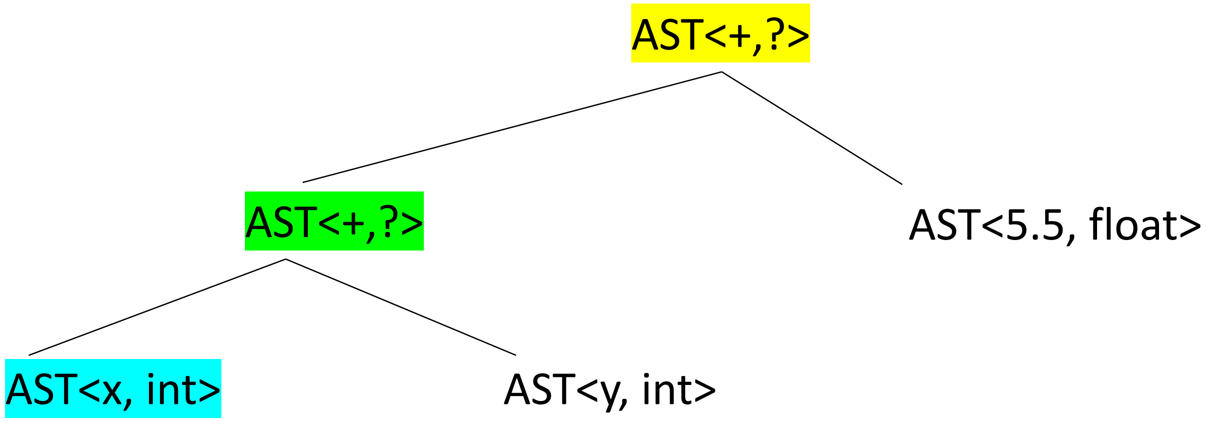
```
int x;
int y;
float w;
w = x + y + 5.5
```

```
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
       return t
```



Type inference

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

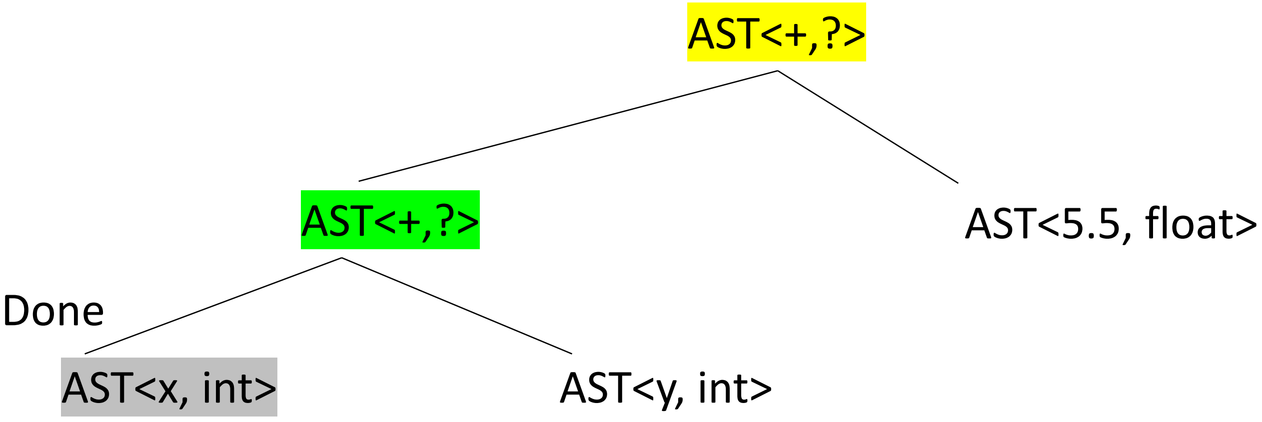



```
def type_inference(n):

    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
       return t
```

Type inference

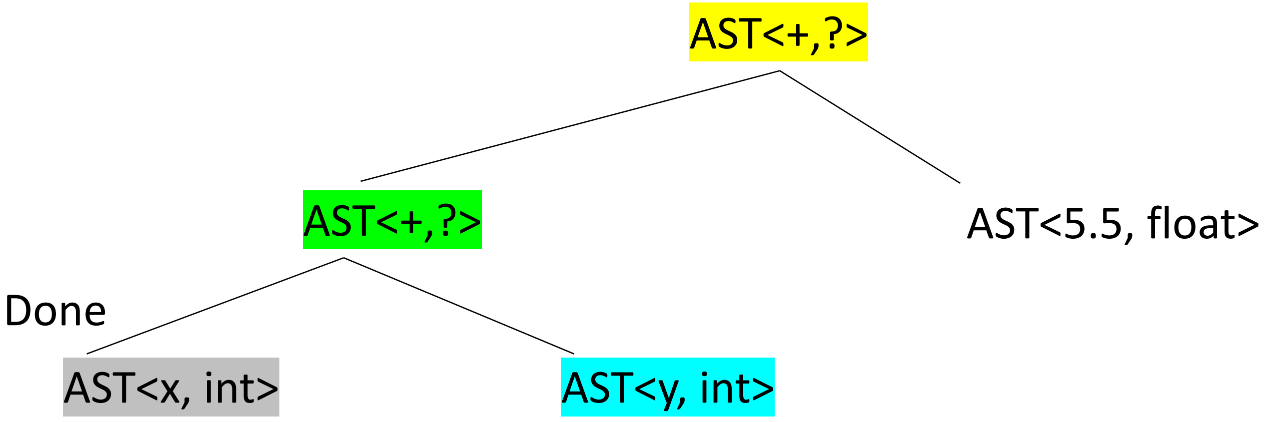
```
def type_inference(n):

    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
       return t
```

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



Type inference

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

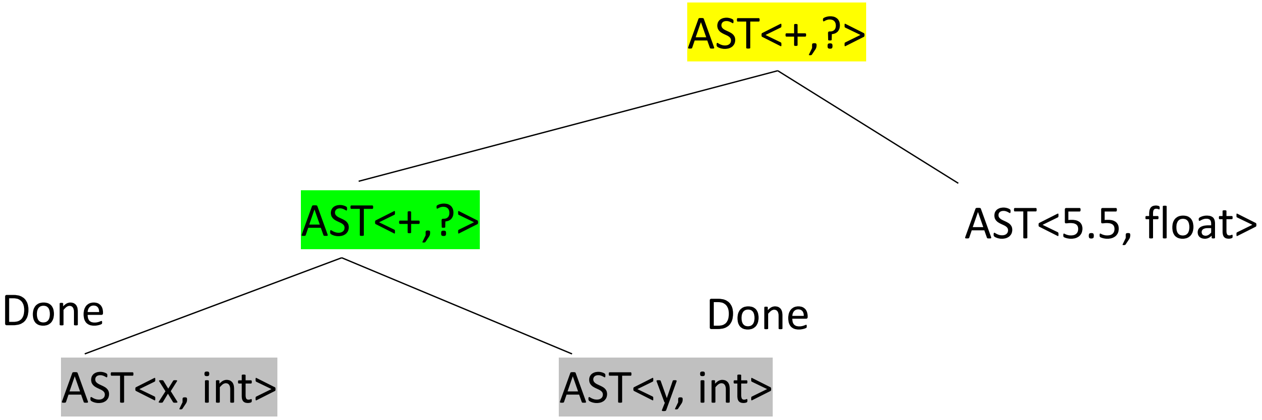



```
def type_inference(n):

    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
       return t
```

Type inference

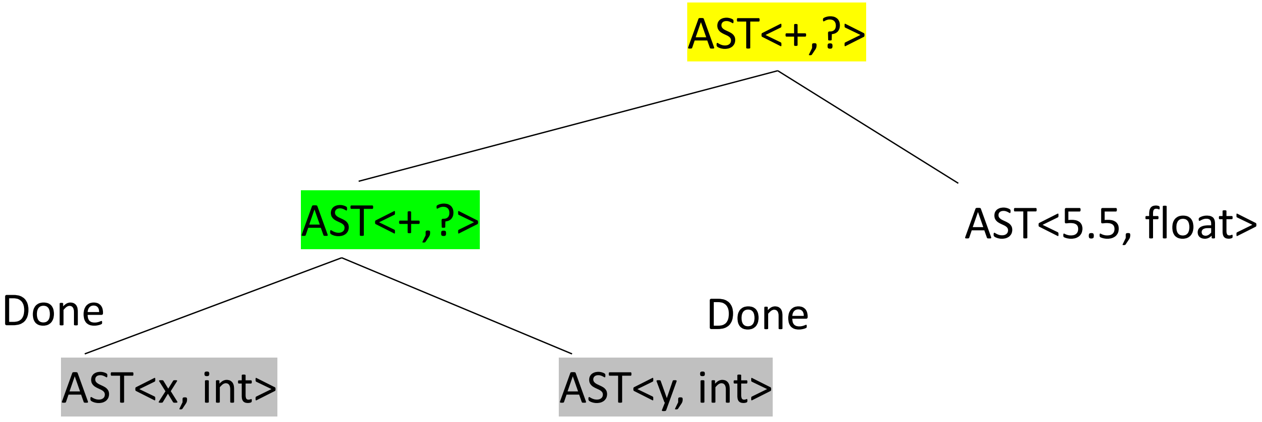
```
int x;
int y;
float w;
w = x + y + 5.5
```

```
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
       return t
```



Type inference

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



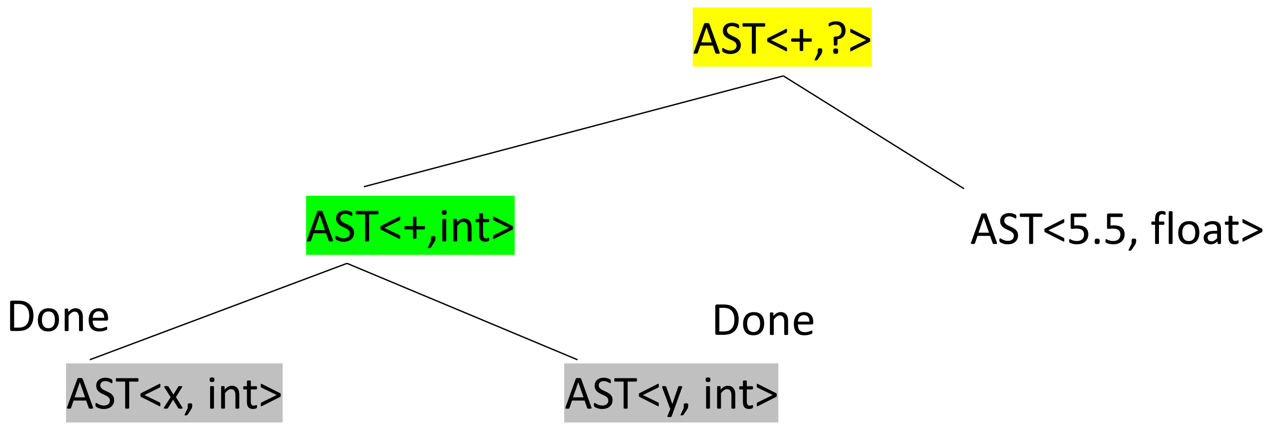

```
def type_inference(n):

    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
       return t
```

Table for most binary ops

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

Type inference

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



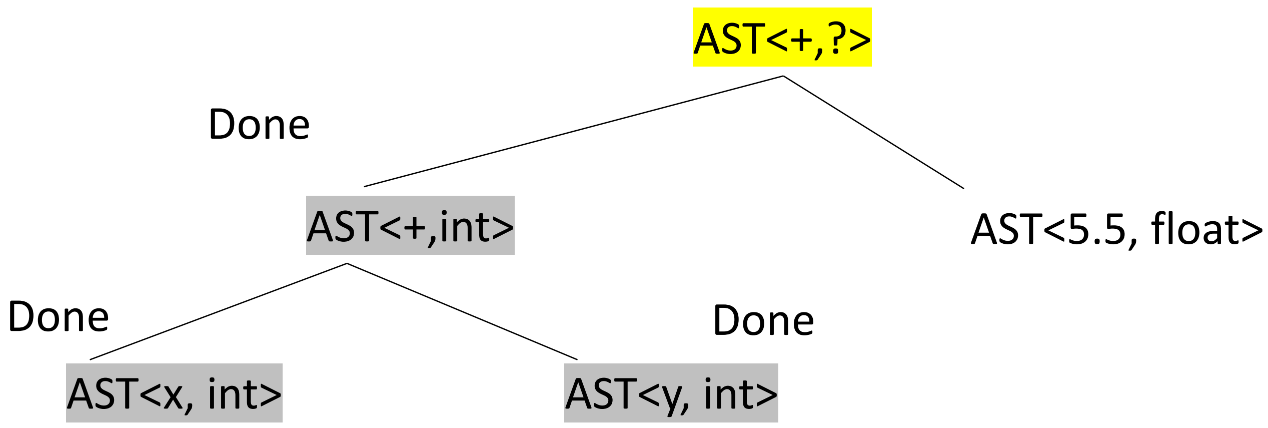

```
def type_inference(n):

    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
       return t
```

Table for most binary ops

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

Type inference

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



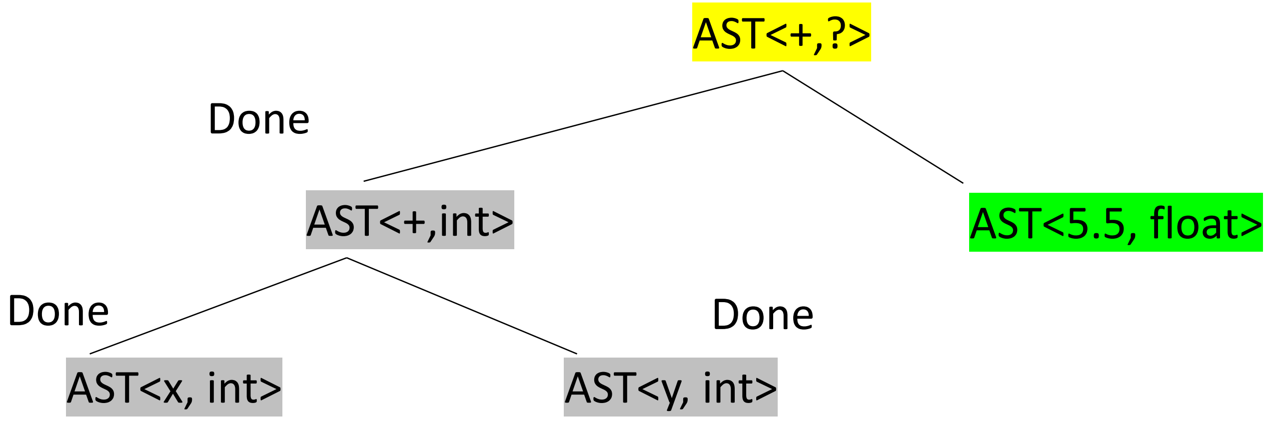

```
def type_inference(n):

    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
       return t
```

Table for most binary ops

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

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
       return t
```

Table for most binary ops

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

Type inference

```
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
       return t
```

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

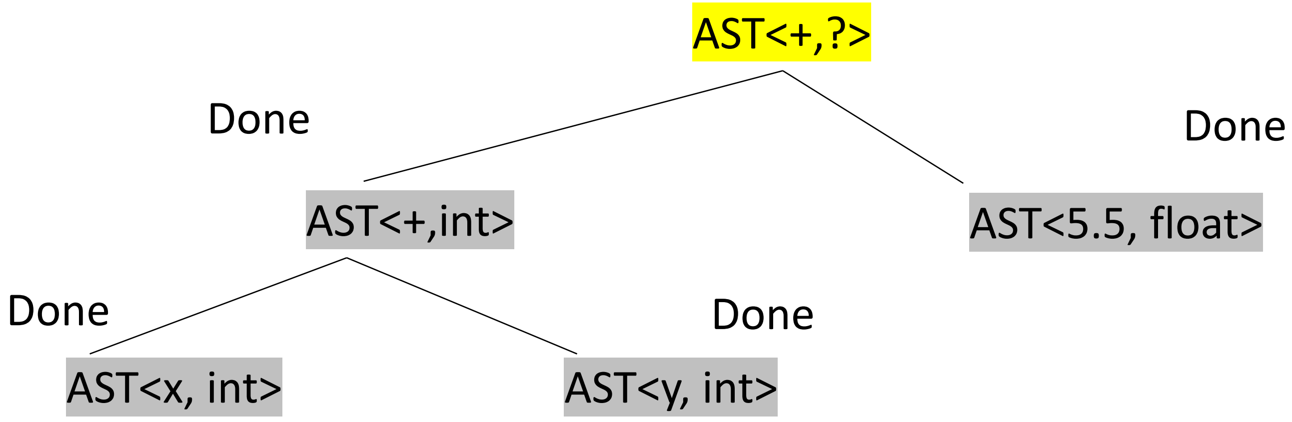
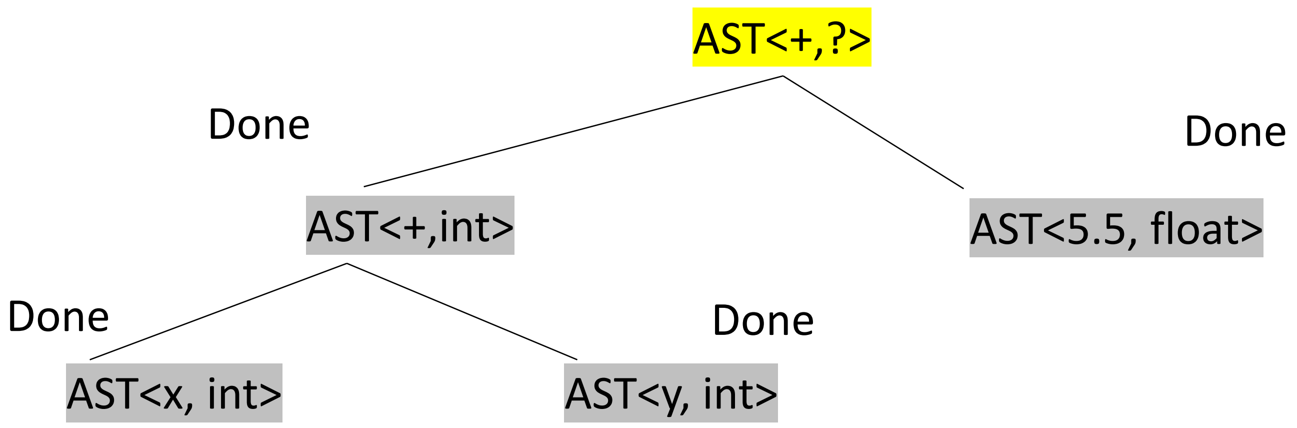


Table for most binary ops

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

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
     return t
```

Table for most binary ops

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

Type inference

```
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
       return t
```

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

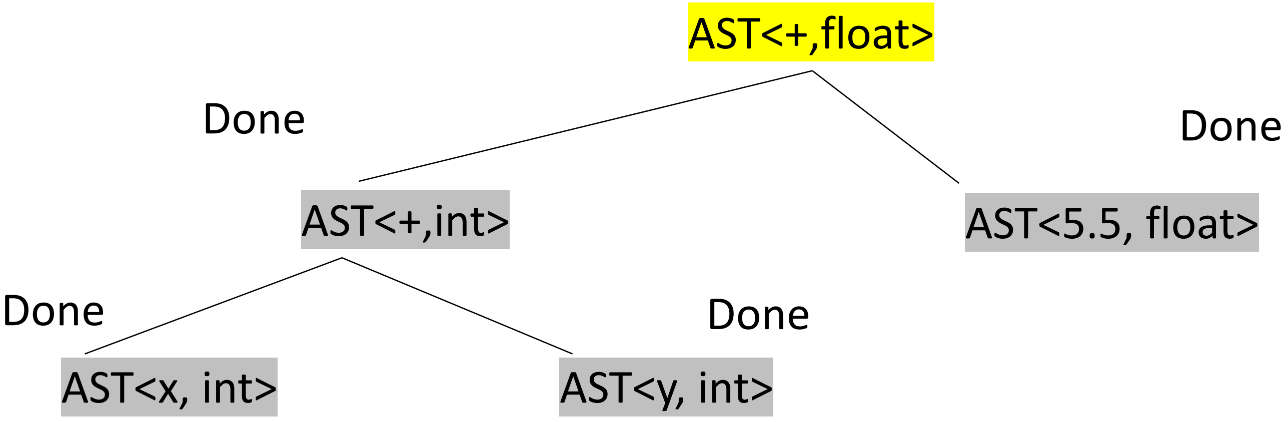


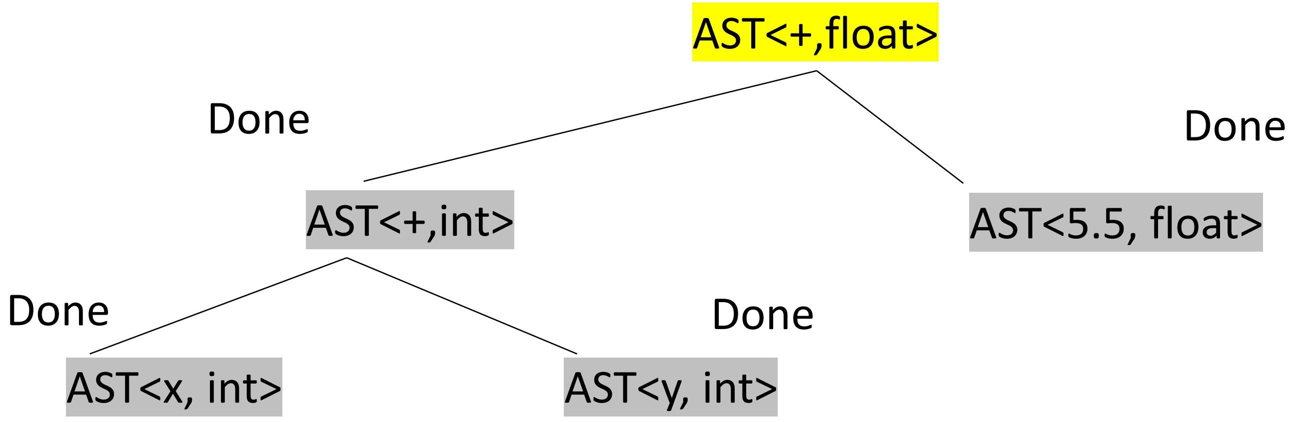
Table for most binary ops

left child	right child	result
`int`	`int`	`int`
`int`	`float`	`float`
`int` → see note		

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
       return t
```

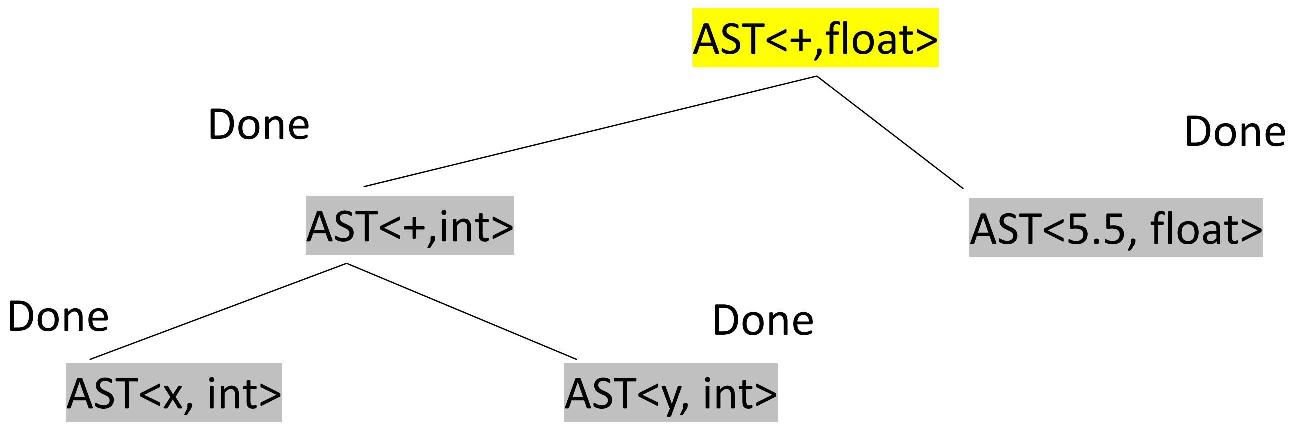


Are we done?

Type inference

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

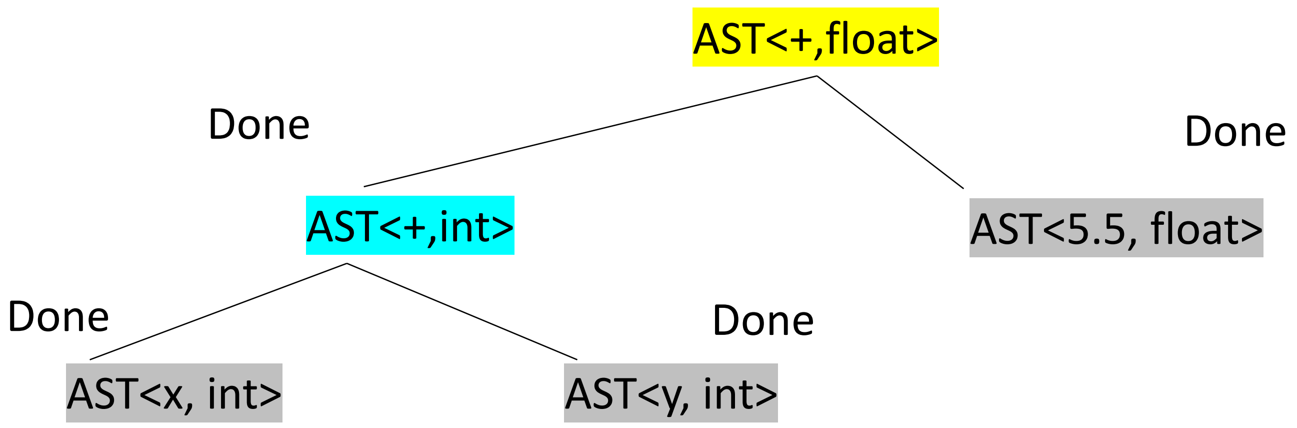
```
def type_inference(n):

    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
```



Are we done?

```
def type_conversion(n):

    if n.left child type is NOT the same as n type:
        conv = get conversion AST node
        conv.child = left child
        set n.left_child to = conv
```

this will need to be done for both children



New type of AST nodes: unary operators

```python
class ASTUnOpNode(ASTNode):
    def __init__(self, child):
        self.child = child


class ASTIntToFloatNode(ASTUnOpNode):
    def __init__(self, child):
        super().__init__(child)


class ASTFloatToIntNode(ASTUnOpNode):
    def __init__(self, child):
        super().__init__(child)
```

```python
from enum import Enum

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

what types are these nodes?

New type of AST nodes: unary operators

```python
class ASTUnOpNode(ASTNode):
    def __init__(self, child):
        self.child = child


class ASTIntToFloatNode(ASTUnOpNode):
    def __init__(self, child):
        super().__init__(child)


class ASTFloatToIntNode(ASTUnOpNode):
    def __init__(self, child):
        super().__init__(child)
```

```python
from enum import Enum

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

what types are these nodes?

New type of AST nodes: unary operators

```python
class ASTUnOpNode(ASTNode):
    def __init__(self, child):
        self.child = child


class ASTIntToFloatNode(ASTBinUnNode):
    def __init__(self, child):
        self.set_type(Types.FLOAT)
        super().__init__(child)


class ASTFloatToIntNode(ASTBinUnNode):
    def __init__(self, child):
        self.set_type(Types.INT)
        super().__init__(child)
```

```python
from enum import Enum

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

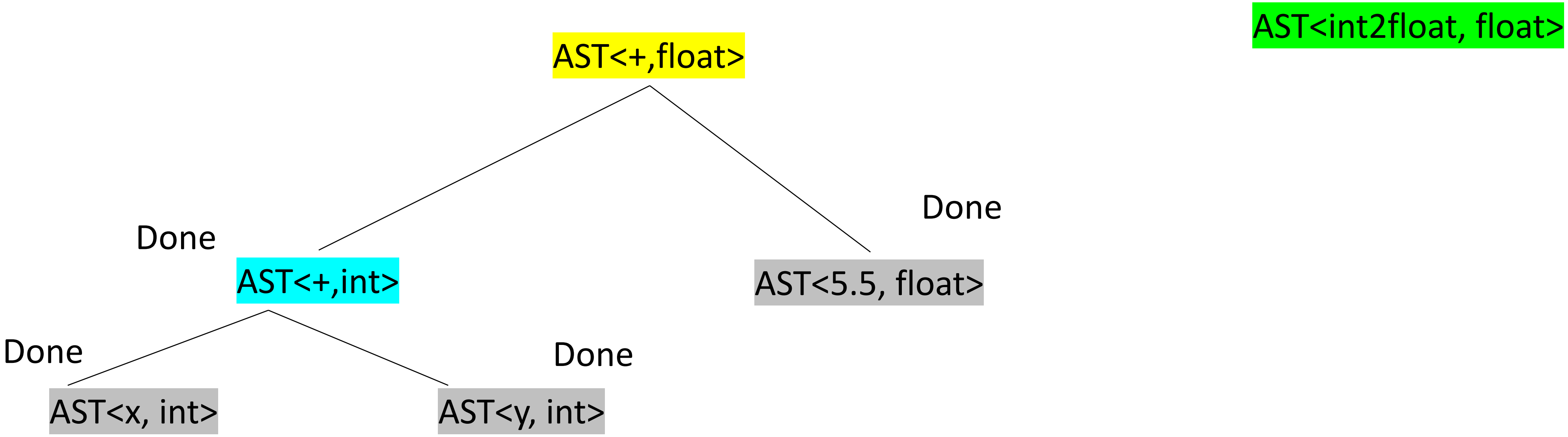
New type of AST nodes: unary operators

what types are these nodes?

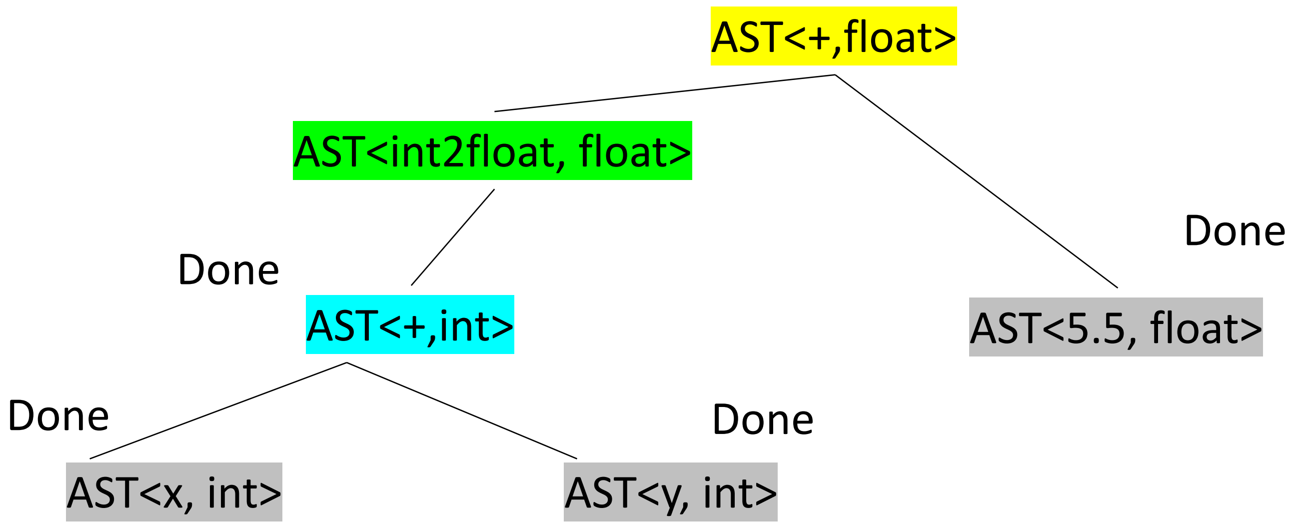
We can go further
and ensure our children
are the right type

```python
class ASTUnOpNode(ASTNode):
    def __init__(self, child):
        self.child = child


class ASTIntToFloatNode(ASTBinUnNode):
    def __init__(self, child):
        self.set_type(Types.FLOAT)
        assert(child.get_type() == Types.INT)
        super().__init__(child)


class ASTFloatToIntNode(ASTBinUnNode):
    def __init__(self, child):
        self.set_type(Types.INT)
        assert(child.get_type() == Types.FLOAT)
        super().__init__(child)
```

```
def type_conversion(n):

    if n.left child type is NOT the same as n type:
       conv = get conversion AST node
       conv.child = left child
       set n.left_child to = conv
```

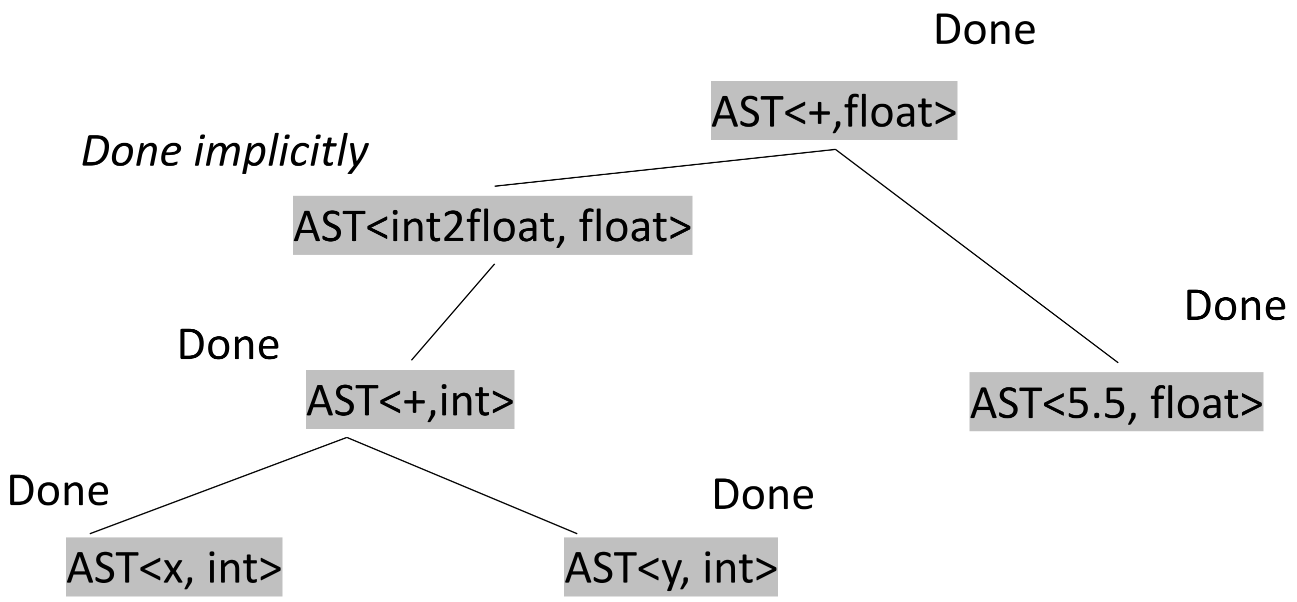


```
def type_conversion(n):

    if n.left child type is NOT the same as n type:
        conv = get conversion AST node
        conv.child = left child
        set n.left_child to = conv
```



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

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**

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?

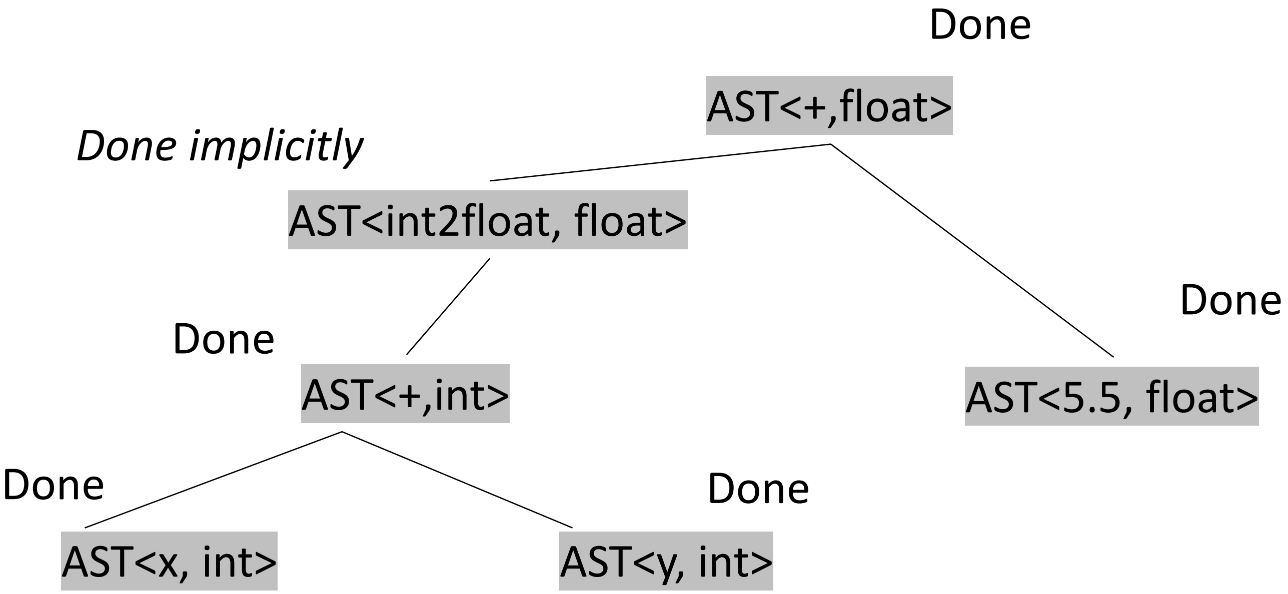
Table for assignment binary ops

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

```
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

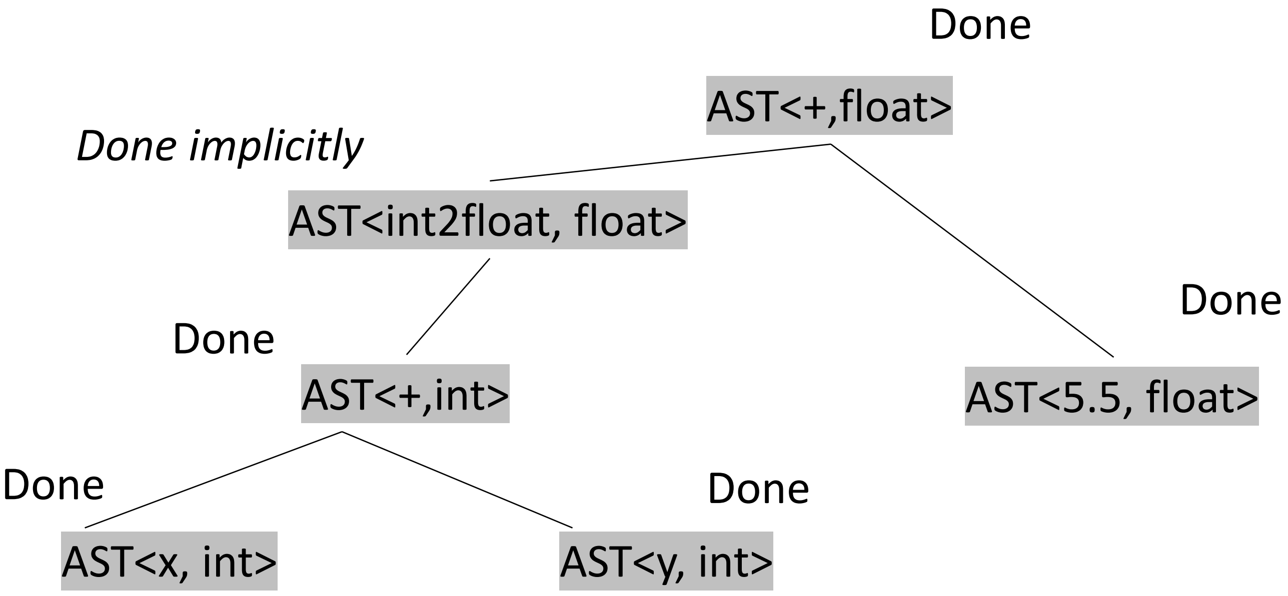
char * in C

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`

Type errors

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

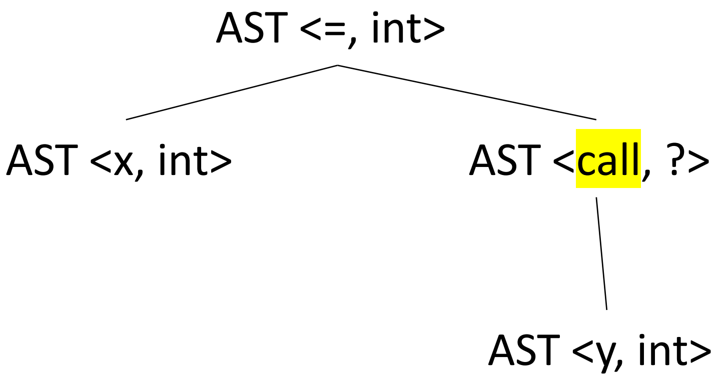



```
def type_inference(n):

    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

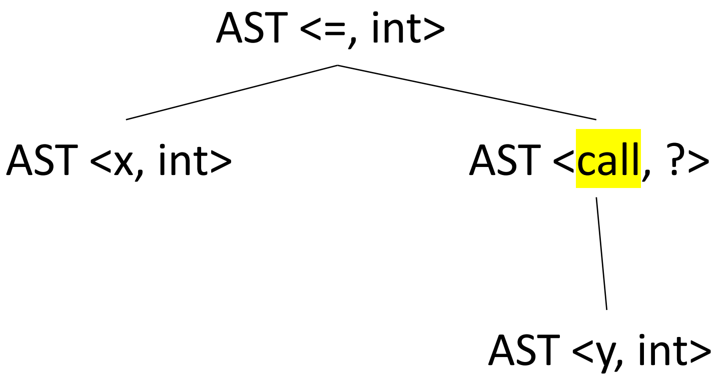
How are functions handled?

```
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

AST <=, int>

AST <x, int>

AST <call, float>

AST <y, int>

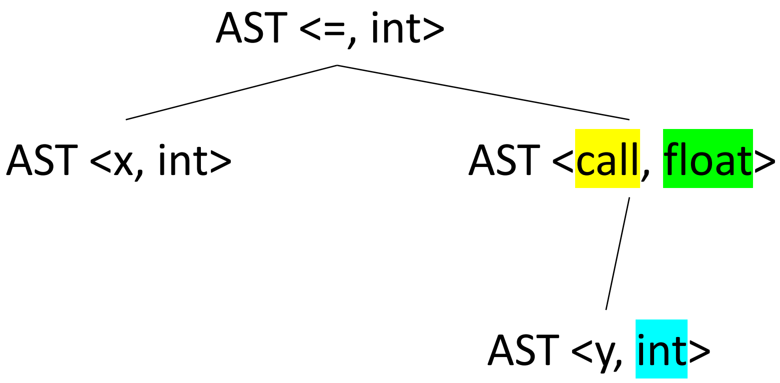
requires a function specification,
using in the .h file:

```
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

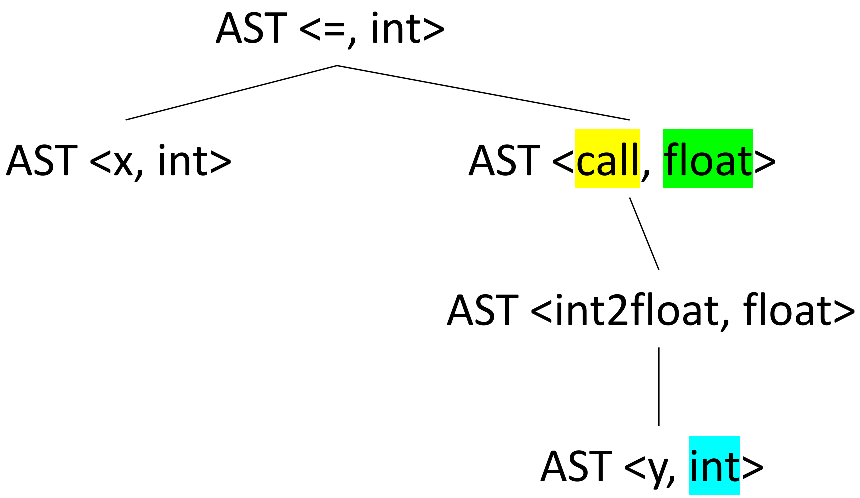
requires a function specification,
using in the .h file:

```
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:

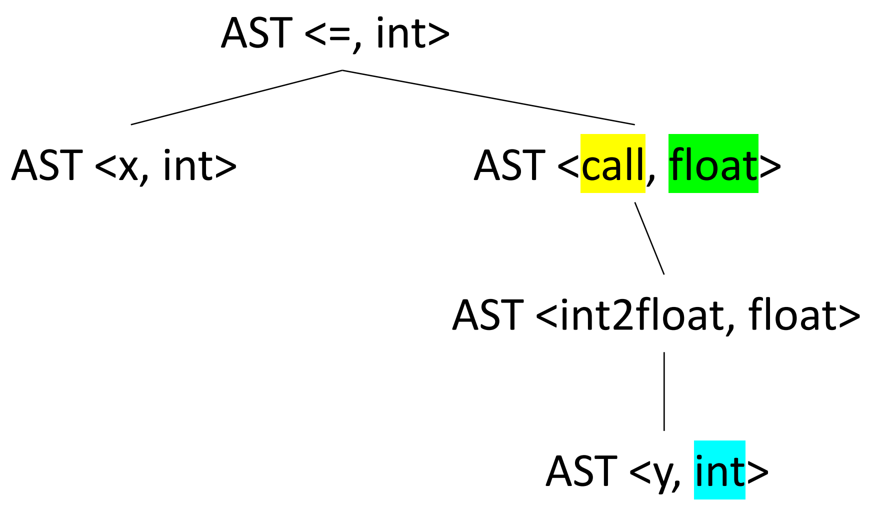
```
float sqrt(float x);
```

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:

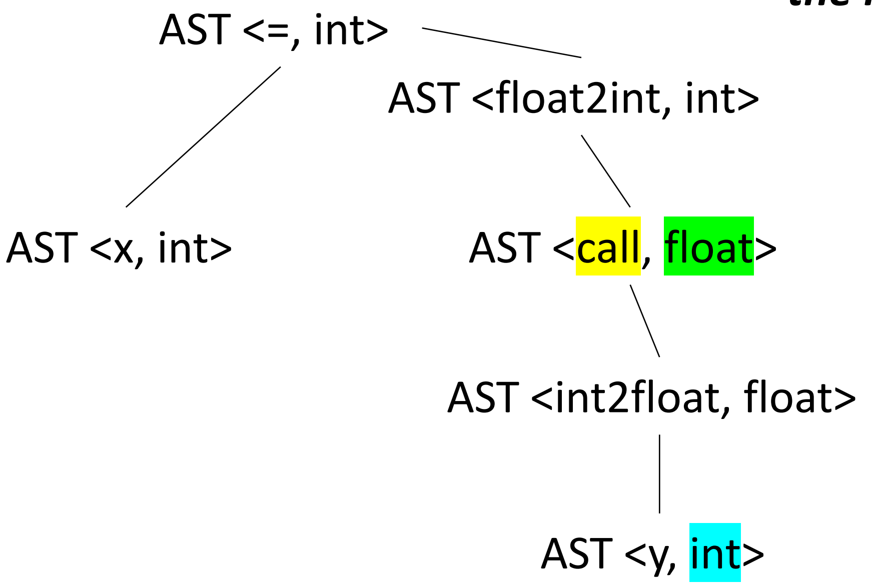
```
float sqrt(float x);
```

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:

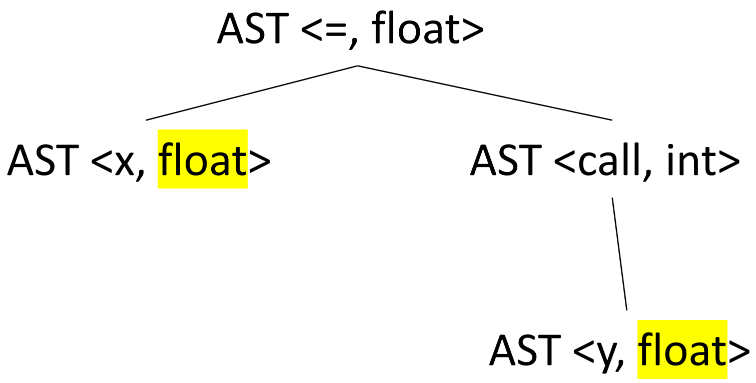
```
float sqrt(float x);
```

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?

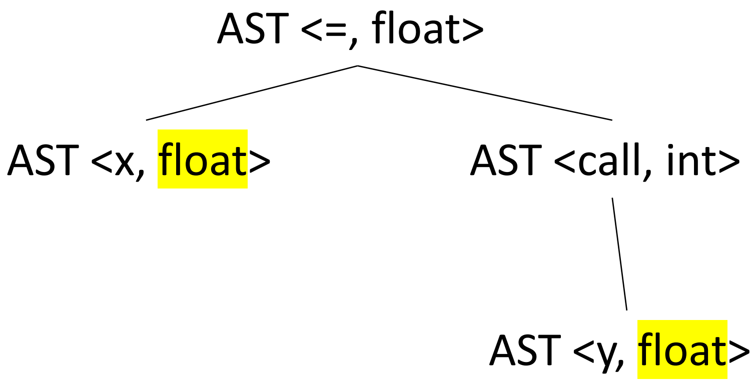


What about floats to ints?

```
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? ...

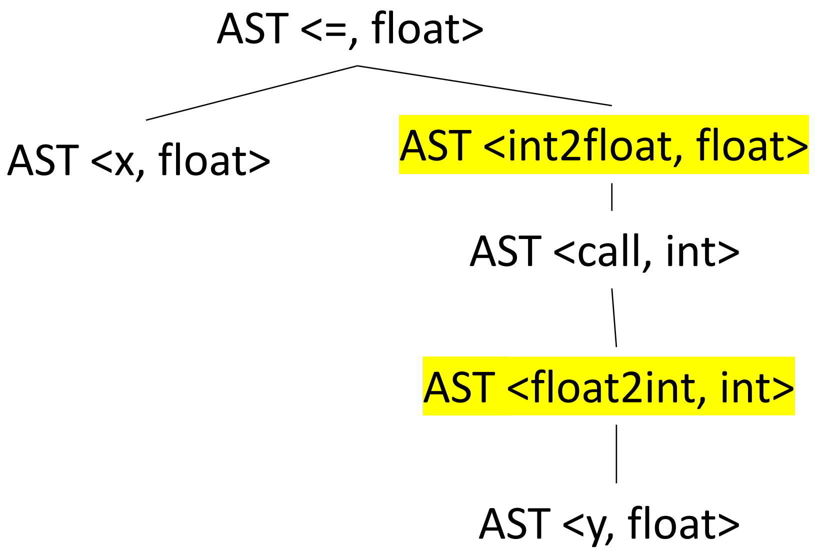


What about floats to ints?

```
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

```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

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
Complex	float	Complex

We can add extra rows and even conversions

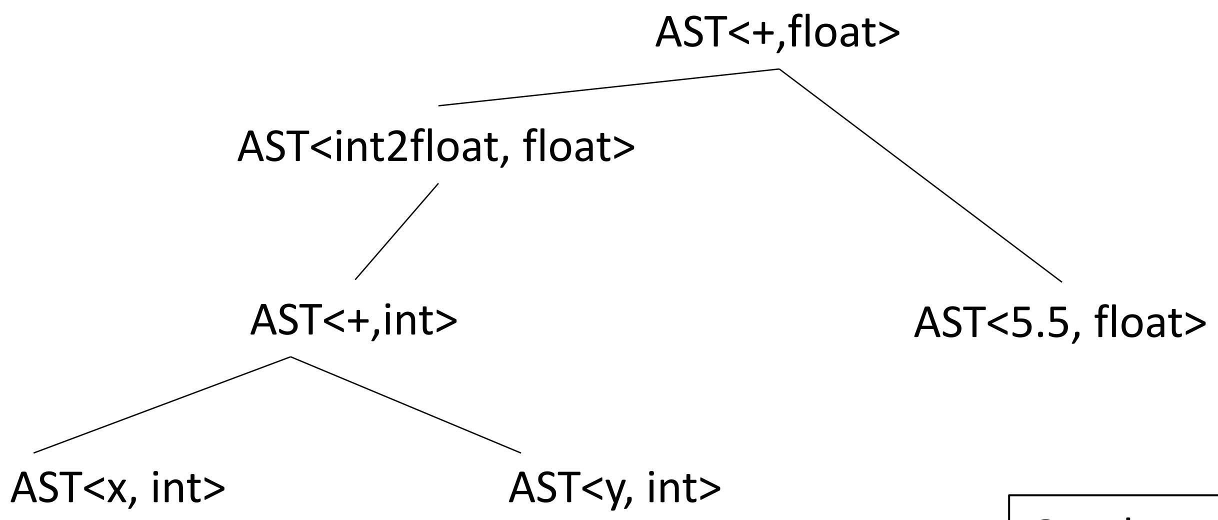
code from: https://www.programiz.com/cpp-programming/operator-overloading

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	this class	LLVM IR
$r_0 \leftarrow x + y;$	`vr0 = addi(x,y);`	`%8 = add nsw i32 %6, %7`
$r_1 \leftarrow 5 * 7;$	`vr1 = multi(5,7);`	`%11 = mul nsw i32 5, 7`
$r_2 \leftarrow r_0 \,/\, r_1$	`vr2 = divi(vr0,vr1);`	`%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

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

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

- 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

Our 3-address code

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

Our 3-address code

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
```

Our 3-address code

binary operators:

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

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

We should have an AST binary operator for each of these.

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

Our 3-address code

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
```

Our 3-address code

unary operators:

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

```
operations can be one of:
[int2float, float2int]
```

```
converts the bits in op0 from one type to
another.
```

Example

`Counting all the values up to 10 in input: int x`

Example

Counting all the values up to 10 in input: int x

```
  r0 = 0;
loop_start:
  r1 = lti(r0,10)
  bne r1, 1, end_label:
  x = addi(x, r0);
  r0 = addi(r0, 1);
  branch loop_start;
end_label:
```

Example

Counting all the values up to 10 in input: float x

```
  r0 = 0;
loop_start:
  r1 = lti(r0,10)
  bne r1, 1, end_label:
  x = addi(x, r0);
  r0 = addi(r0, 1);
  branch loop_start;
end_label:
```

See everyone on Friday

- We will discuss transforming an AST into linear code