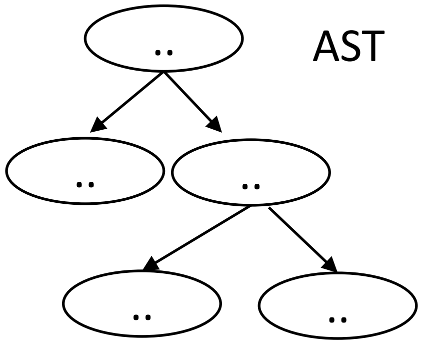
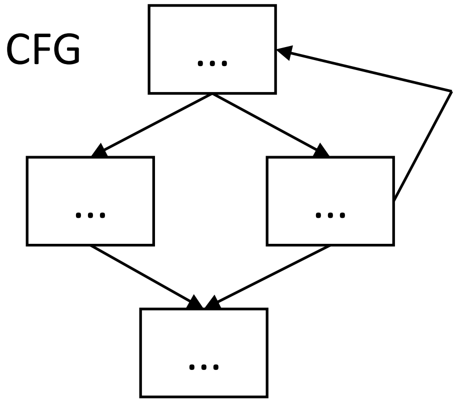
CSE110A: Compilers

May 3, 2024

Topics:

- *Module 8: Intermediate representations*
  - *Type checking*





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

- Glad to see everyone survived the test!

- Homework 1 grades are out
  - Planning on homework 2 grades out by next Friday
  - Grading midterm this Friday and hoping to get grades by Monday

- Homework 3 is due by Friday (one extra day)
  - Delay is due to prepping HW 3, not due to the poll

- Homework 4 will be released on Friday

Announcements

- Next week:
  - I will be gone Wednesday and Friday
  - One day will be a midterm review led by the Tas
  - The other day we are figuring out, likely either:
    - A special topics lecture by the TAs
    - Canceled

Announcements

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

- Grading questions are private questions on piazza or office hours. Not public on piazza.

Announcements

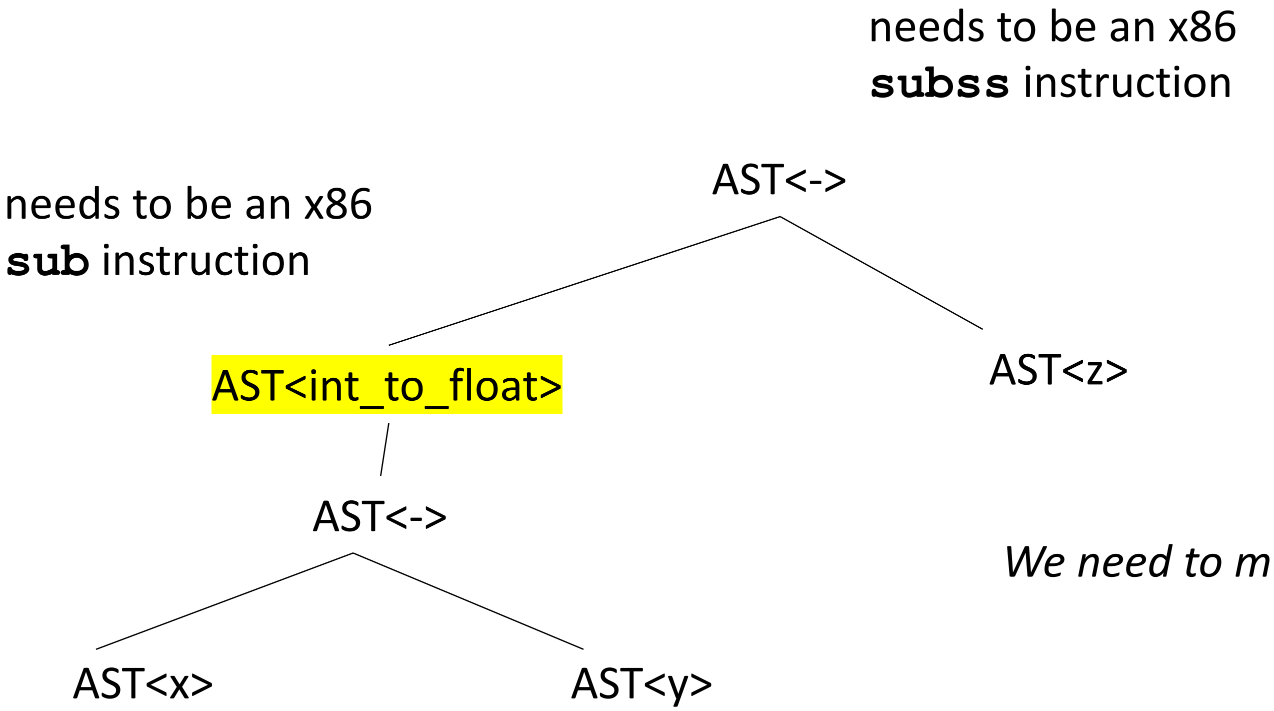
- Quick announcement from Elliot

Back into type systems!

Evaluate an AST by doing a post order traversal

```
Expr  ::= NUM Expr2
Expr2 ::= MINUS NUM Expr2
       |   “”
```

```
int x;
int y;
float z;
float w;
w = x - y - z
```



needs to be an x86 **subss** instruction

needs to be an x86 **sub** instruction

AST<->

AST<int_to_float>

AST<z>

AST<->

AST<x>

AST<y>

We need to make sure our operands are in the right format!

Type systems

- Given a language a type system defines:
  - The primitive (base) types in the language
  - How the types can be converted to other types
    - implicitly or explicitly
  - How the user can define new types

Type checking and inference

- Check a program to ensure that it adheres to the type system

Especially interesting for compilers as a program given in the type system for the input language must be translated to a type system for lower-level program

Type systems

Considerations:

- Base types:
  - ints
  - chars
  - strings
  - floats
  - bool

- How to combine types in expressions:
  - int and float?
  - int and char?
  - int and bool?

Type checking

Two components

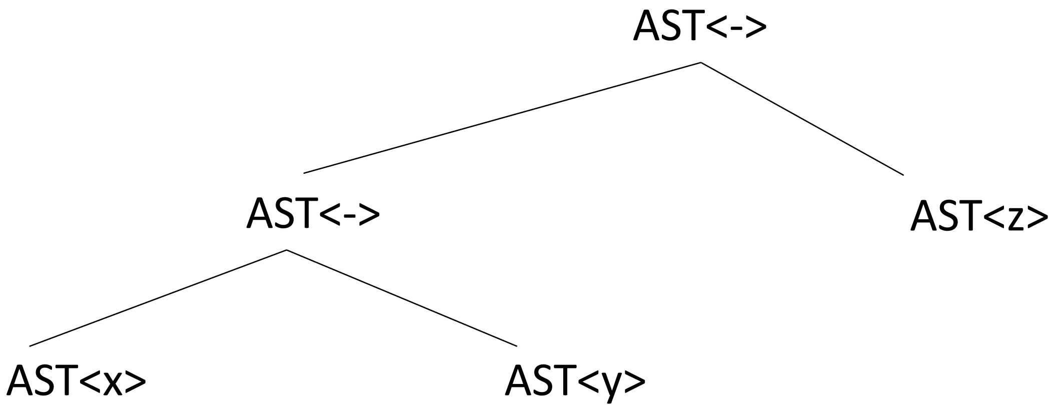
- Type inference
  - Determines a type for each AST node
  - Modifies the AST into a type-safe form

- Catches type-related errors

Type checking on an AST

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

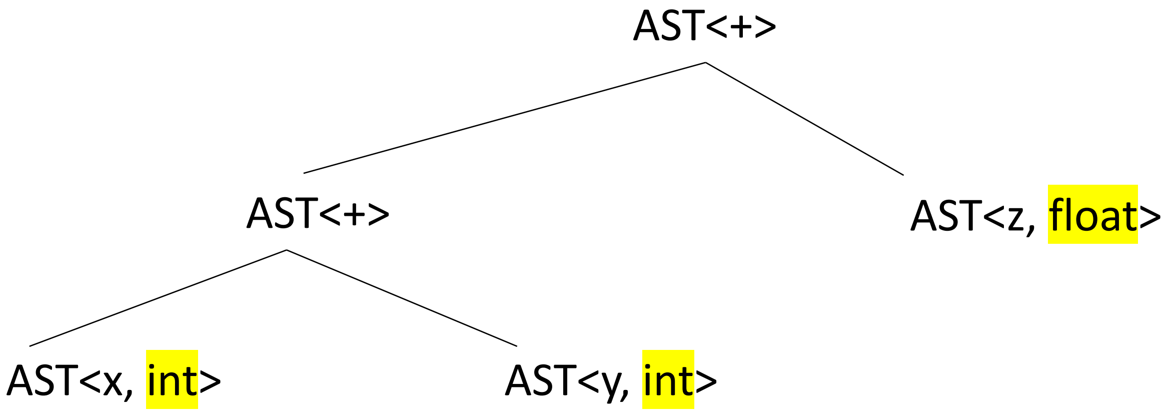
each node additionally gets a type



Type checking on an AST

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

each node additionally gets a type
we can get this from the symbol table for the leaves or based on the input (e.g. 5 vs 5.0)



AST<+>

AST<+>

AST<z, float>

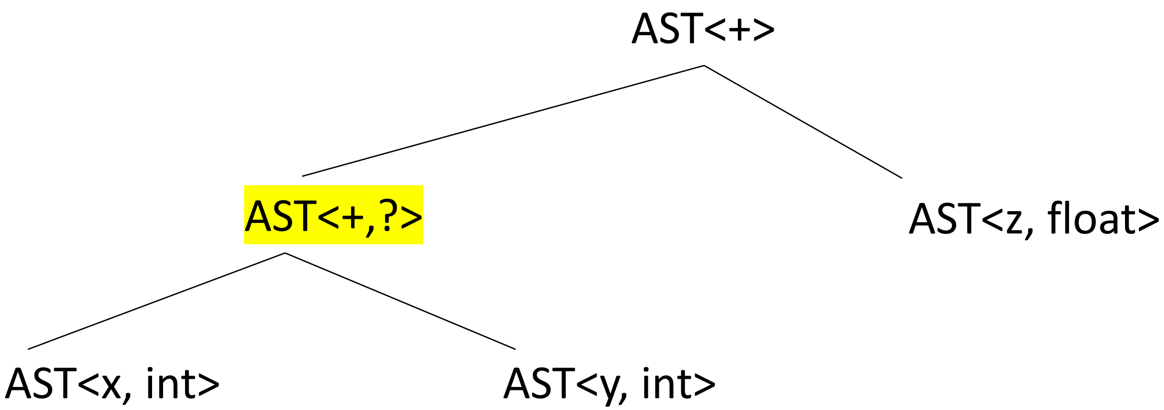
AST<x, int>

AST<y, int>

Type checking on an AST

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

How do we get the type for this one?



AST<+>

AST<+,?>

AST<z, float>

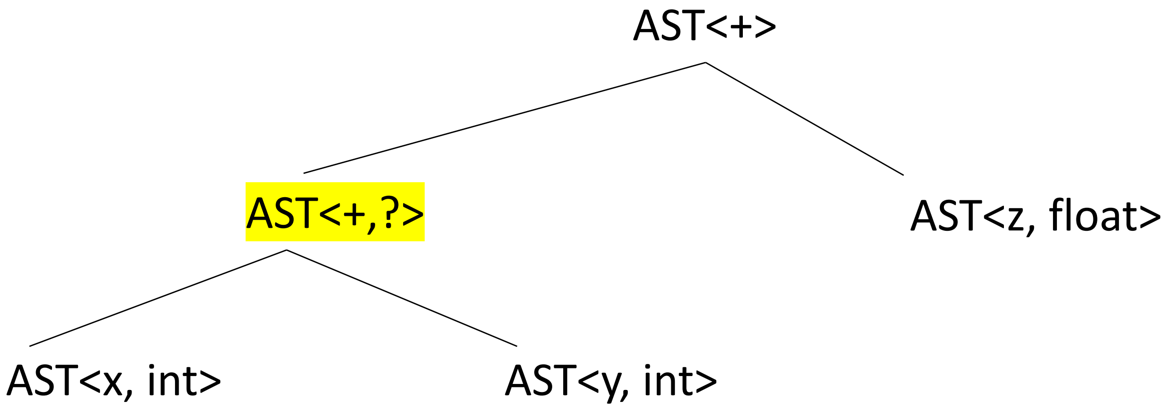
AST<x, int>

AST<y, int>

Type checking on an AST

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

How do we get the type for this one?



AST<+>

AST<+,?>

AST<z, float>

AST<x, int>

AST<y, int>

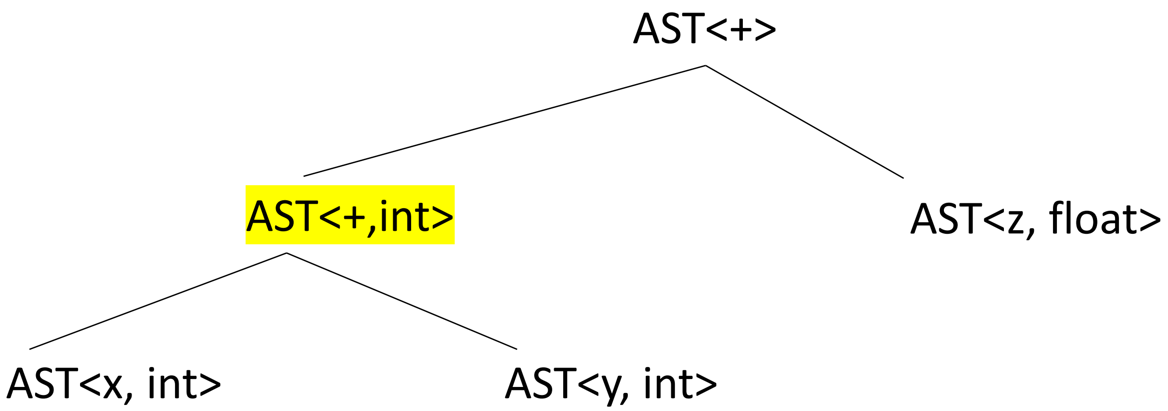
inference rules for addition:

first	second	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

Type checking on an AST

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

How do we get the type for this one?



AST<+>

AST<+,int>

AST<z, float>

AST<x, int>

AST<y, int>

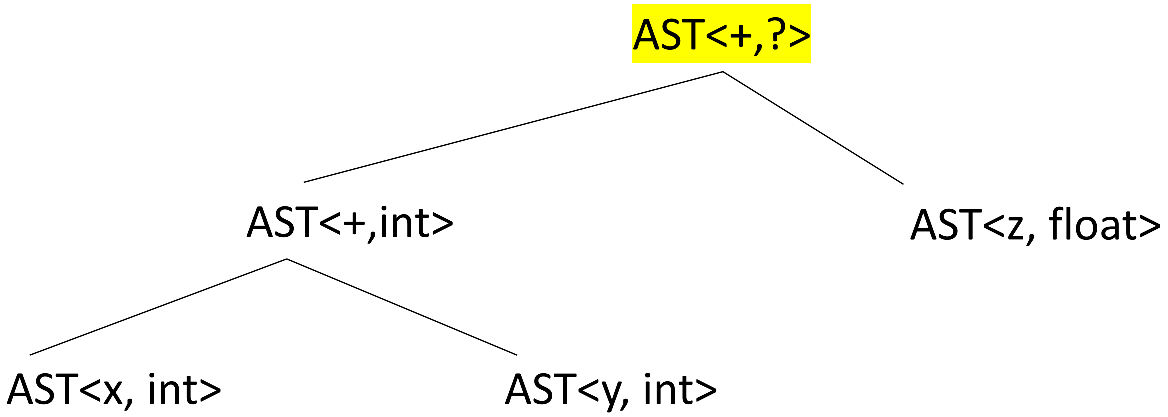
inference rules for addition:

first	second	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

Type checking on an AST

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

How do we get the type for this one?



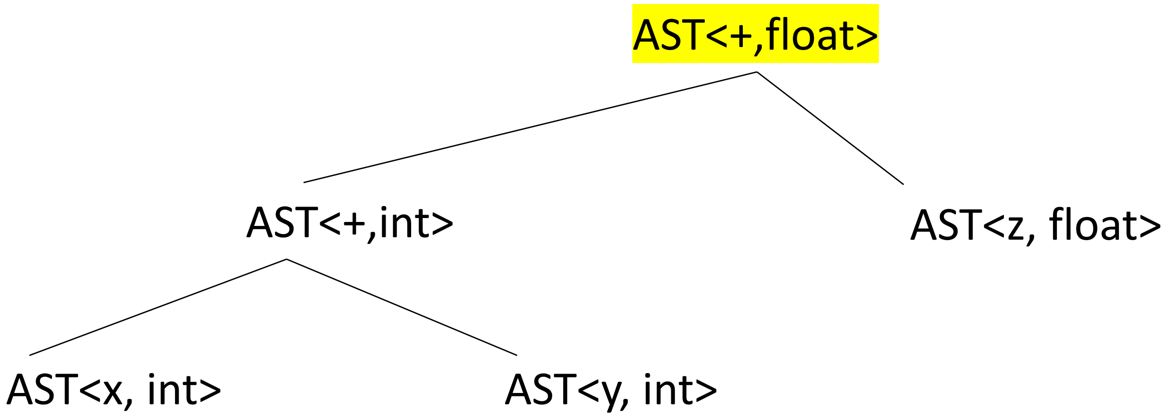
inference rules for addition:

first	second	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

Type checking on an AST

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

How do we get the type for this one?



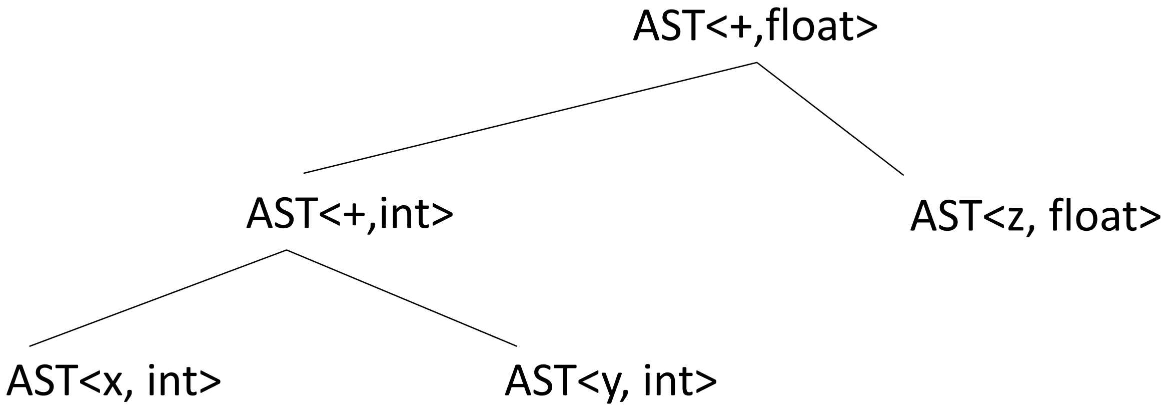
inference rules for addition:

first	second	result
int	int	int
int	float	float
float	int	float
float	float	float

Type checking on an AST

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

How do we get the type for this one?



inference rules for addition:

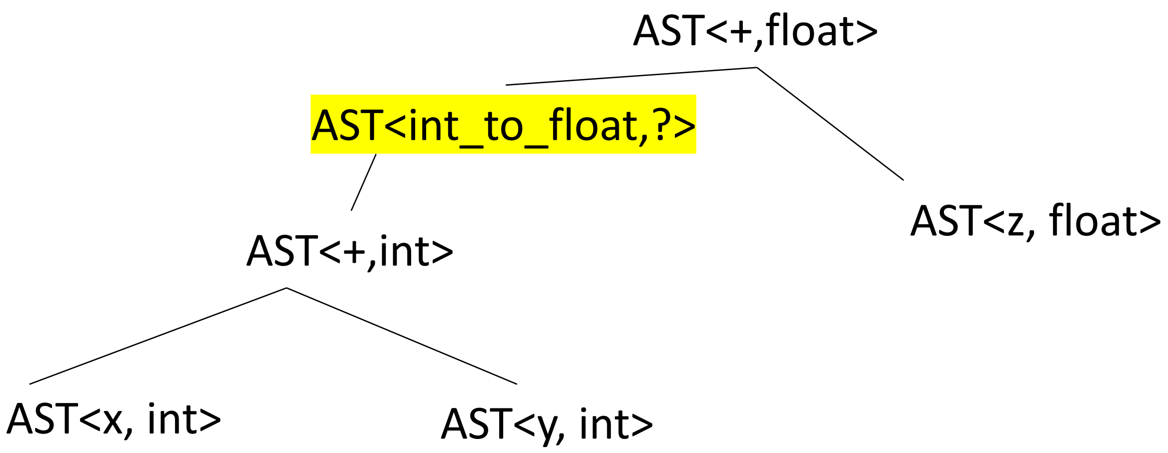
first	second	result
int	int	int
int	float	float
float	int	float
float	float	float

what else?

Type checking on an AST

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

How do we get the type for this one?



AST<+,float>

AST<int_to_float,?>

AST<z, float>

AST<+,int>

AST<x, int>

AST<y, int>

inference rules for addition:

first	second	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

what else? need to convert the int to a float

```python
class ASTNode():
    def __init__(self):
        pass
```

```python
class ASTLeafNode(ASTNode):
    def __init__(self, value):
        self.value = value


class ASTNumNode(ASTLeafNode):
    def __init__(self, value):
        super().__init__(value)


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

```python
class ASTBinOpNode(ASTNode):
    def __init__(self, l_child, r_child):
        self.l_child = l_child
        self.r_child = r_child


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


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

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

Enum for types

```python
from enum import Enum

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

Our base AST Node needs a type

```python
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

```python
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)
```

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)
```

Where can we get the value type for an ID?

Symbol Table

Say we are matched the statement:

```
int x;
```

- `SymbolTable ST;`

(TYPE, 'int') (ID, 'x')

declare_statement ::= TYPE ID SEMI

```
{
  eat(TYPE)
  id_name = self.to_match.value
  eat(ID)
  ST.insert(id_name, None)
  eat(SEMI)
}
```

in homework 2 and 3 we didn't record any information in the symbol table

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.value
  eat(TYPE)
  id_name = self.to_match.value
  eat(ID)
  ST.insert(id_name, value_type)
  eat(SEMI)
}
```

previously we weren’t saving any information about the ID

record the type in the symbol table

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)
```

Where can we get the value type for an ID?

But that doesn’t get us here yet...

add the type at parse time

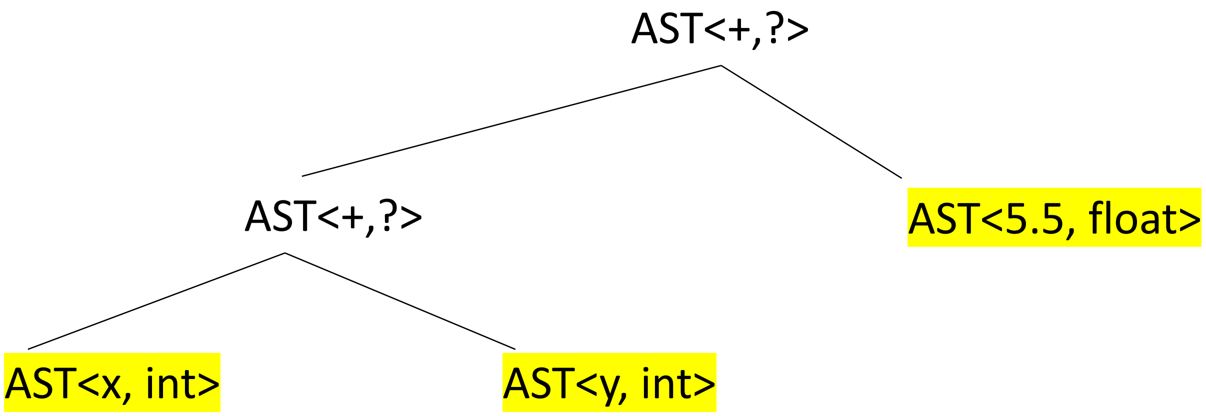
```
Unit ::= ID
     |   NUM
```

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

Type inference

- We now have the types for the leaf nodes

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



AST<+,?>

AST<+,?>

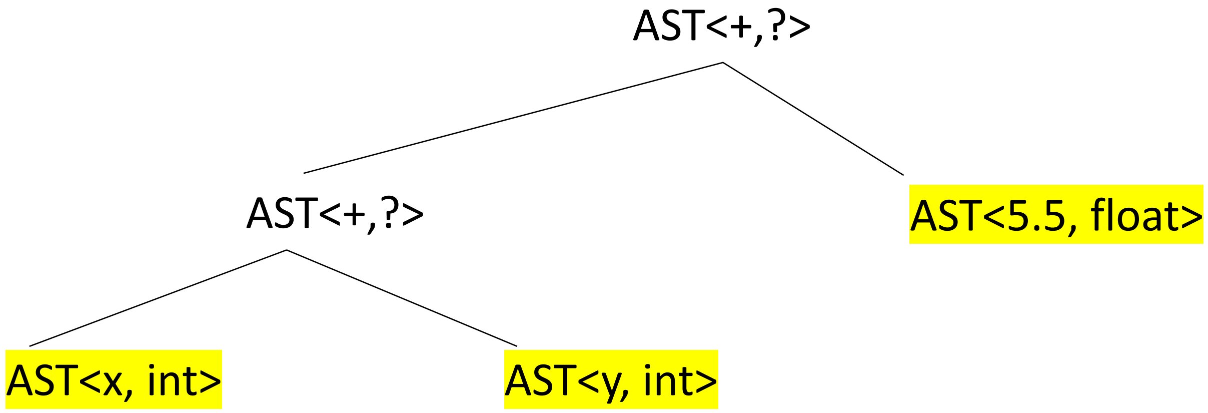
AST<5.5, float>

AST<x, int>

AST<y, int>

Type inference

- We now have the types for the leaf nodes



Next steps:

we do a post order traversal
on the AST and do a type inference

Type inference

```
def type_inference(n):
```

Given a node n: find its type and the types of any of its children

Type inference

```
def type_inference(n):
```

Given a node n: find its type and the types of any of its children

```
 case split on n:

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

base case

Type inference

```
def type_inference(n):

 case split on n:

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

 if n is a plus node:
    ...
```

Given a node n: find its type and the types of any of its children

Type inference

```
def type_inference(n):
```

Given a node n: find its type and the types of any of its children

```
 case split on n:

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

 if n is a plus node:
    return lookup type from table
```

lookup the rule for plus

inference rules for plus

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

Type inference

```
def type_inference(n):

 case split on n:

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

 if n is a plus node:
    return lookup type from table
```

Given a node n: find its type and the types of any of its children

lookup the rule for plus

inference rules for plus

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

but we’re missing a few things

Type inference

```
def type_inference(n):
```

Given a node n: find its type and the types of any of its children

```
 case split on n:

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

 if n is a plus node:
     do type inference on children
     return lookup type from table
```

we need to make sure the children have types!

inference rules for plus

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

Type inference

```
def type_inference(n):

 case split on n:

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

 if n is a plus node:
    do type inference on children
    t = lookup type from table
    set n type to t
    return t
```

Given a node n: find its type and the types of any of its children

we should record our type

inference rules for plus

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

Type inference

```
def type_inference(n):

 case split on n:

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

 if n is a plus node:
    do type inference on children
    t = lookup type from table
    set n type to t
    return t
```

Given a node n: find its type and the types of any of its children

is this just for plus?

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

Type inference

```
def type_inference(n):
```

Given a node n: find its type and the types of any of its children

```
case split on n:

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

if n is a plus node:
   do type inference on children
   t = lookup type from table
   set n type to t
   return t
```

is this just for plus?

most language promote types, e.g. ints to float for expression operators

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

Type inference

```
def type_inference(n):
```

Given a node n: find its type and the types of any of its children

```
 case split on 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
```

is this just for plus?

most language promote types, e.g. ints to float for expression operators

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

Type inference

```
def type_inference(n):

 case split on 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
```

What about for assignments?

```
int x;
cout << (x = 5.5) << endl;
```

What does this return?

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

Type inference

```
def type_inference(n):

 case split on 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
```

What about for assignments?

```
int x;
cout << (x = 5.5) << endl;
```

What does this return?

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

whatever the left is

Type checking

- Checking for errors

Type inference

```
def type_inference(n):
```

Given a node n: find its type and the types of any of its children

```
 case split on n:

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

 if n is a plus node:
    do type inference on children
    t = lookup type from table
    if t is None:
       throw type exception
    set n type to t
    return t
```

we should record our type

inference rules for plus

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

Type inference

```
def type_inference(n):
```

Given a node n: find its type and the types of any of its children

```
case split on n:

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

if n is a plus node:
   do type inference on children
   t = lookup type from table
   if t is None:
      throw type exception
   set n type to t
   return t
```

we should record our type

inference rules for plus

left	right	result
int	int	int
int	float	float
float	int	float
float	float	float
string	int	None

like in Python

Type inference

What other examples would throw an error?

```
def type_inference(n):
```

Given a node n: find its type and the types of any of its children

```
case split on n:

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

if n is a plus node:
   do type inference on children
   t = lookup type from table
   if t is None:
      throw type exception
   set n type to t
   return t
```

we should record our type

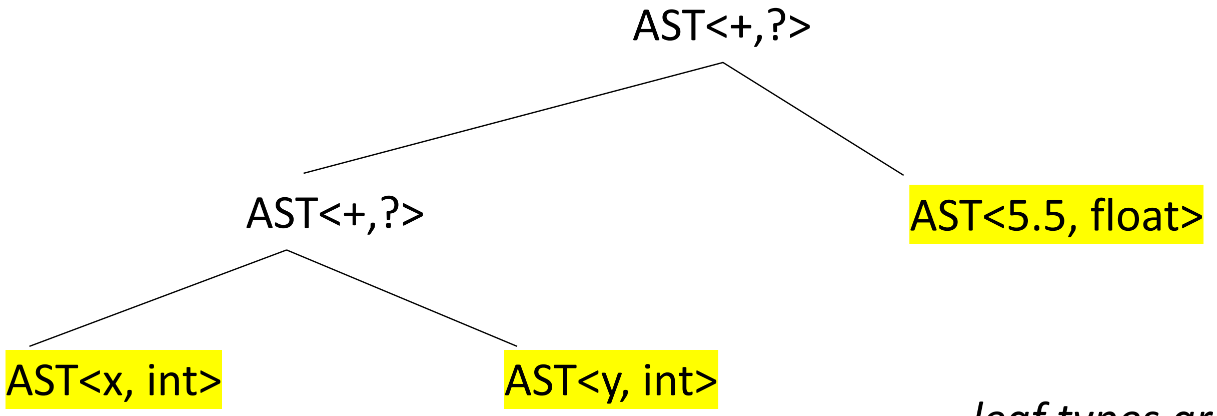
inference rules for plus

left	right	result
int	int	int
int	float	float
float	int	float
float	float	float
string	int	None

like in Python

Type inference

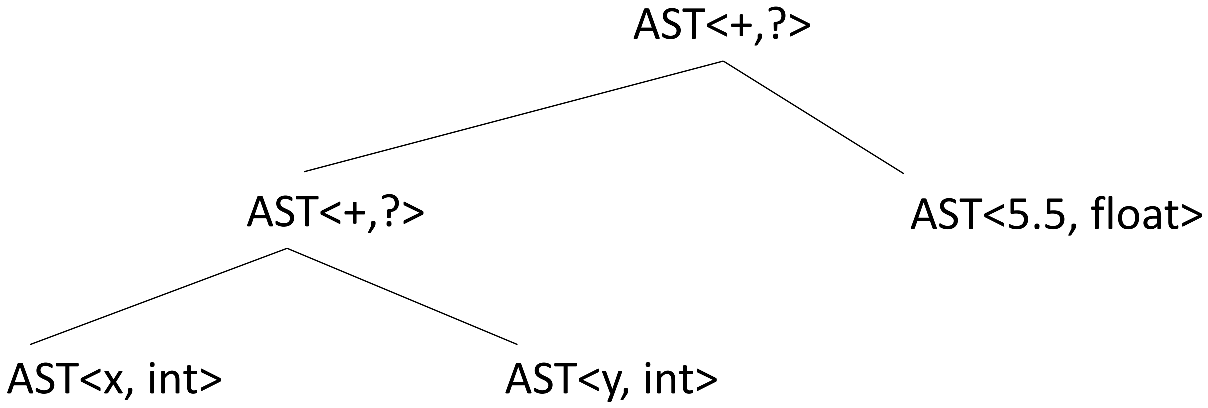
```
int x;
int y;
float w;
w = x + y + 5.5
```



leaf types are provided on construction

Type inference

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

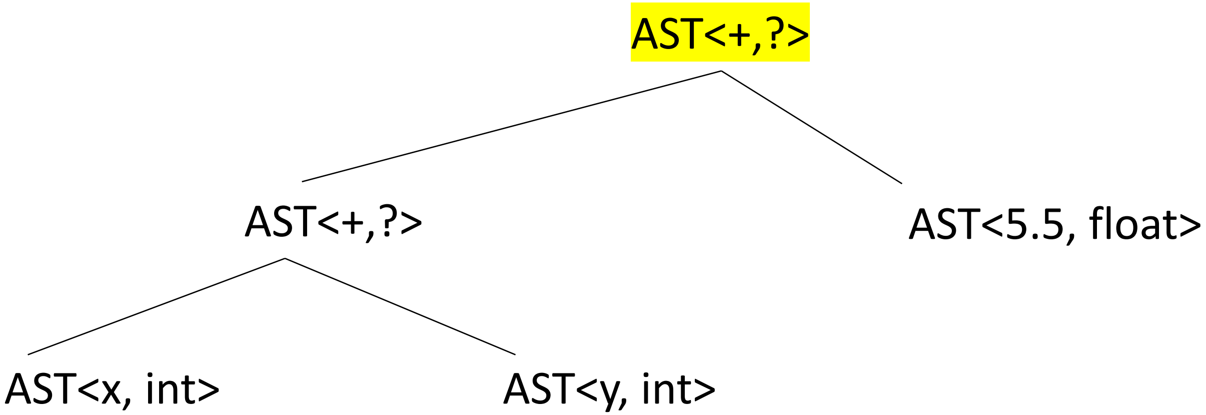


```
def type_inference(n):

     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



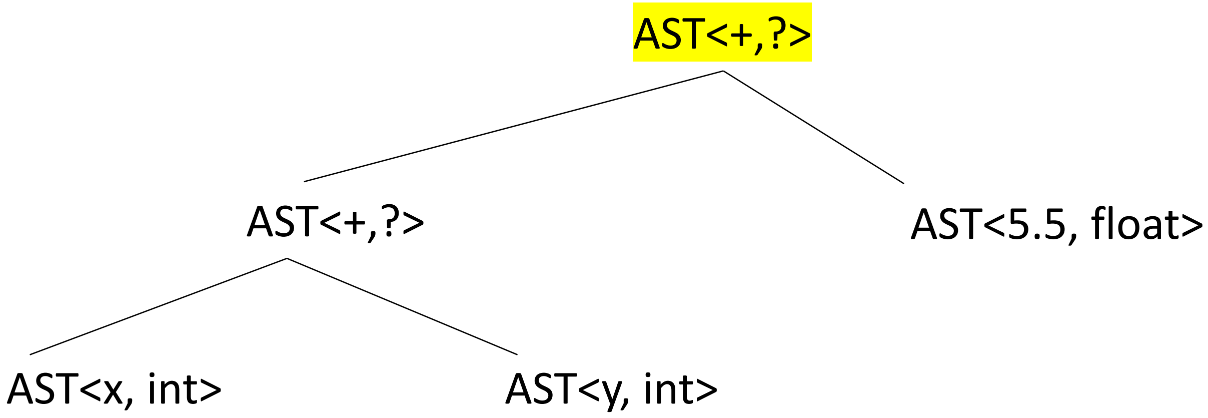
```
def type_inference(n):

    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
```

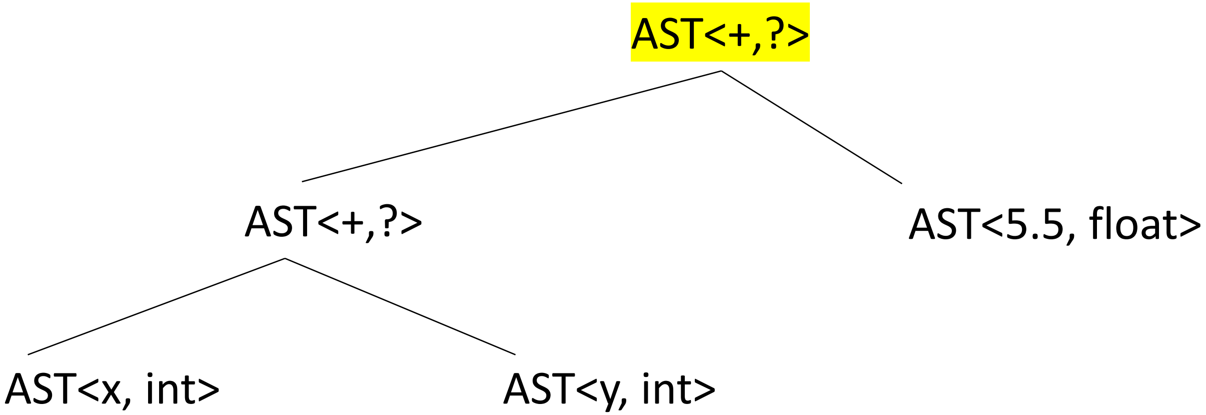


Type inference

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

recursion

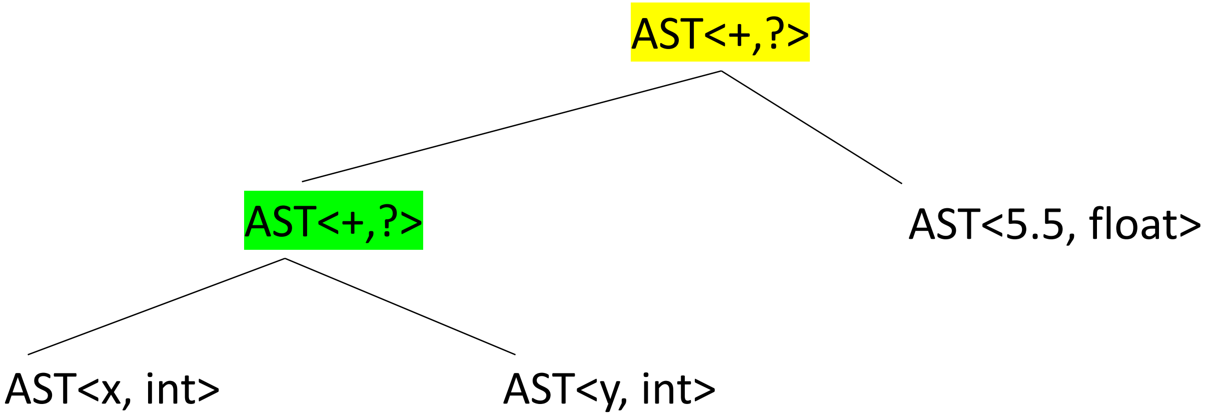
```
def type_inference(n):

    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
```

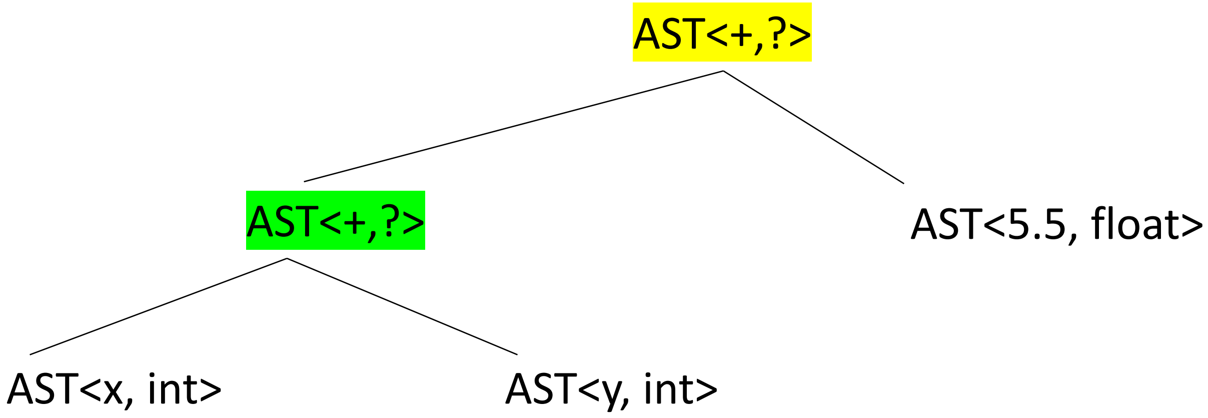


Type inference

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

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
```

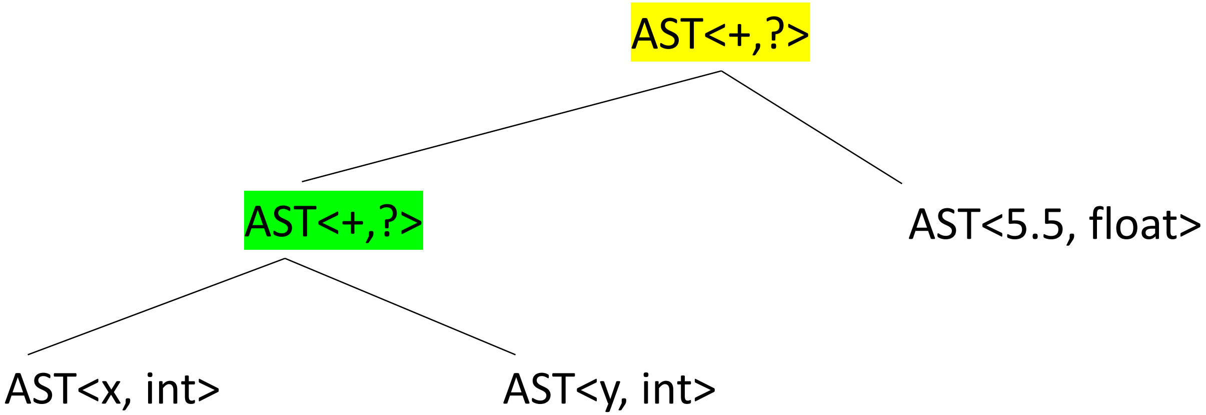


Type inference

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

```
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
```

recursion



AST<+,?>

AST<+,?>

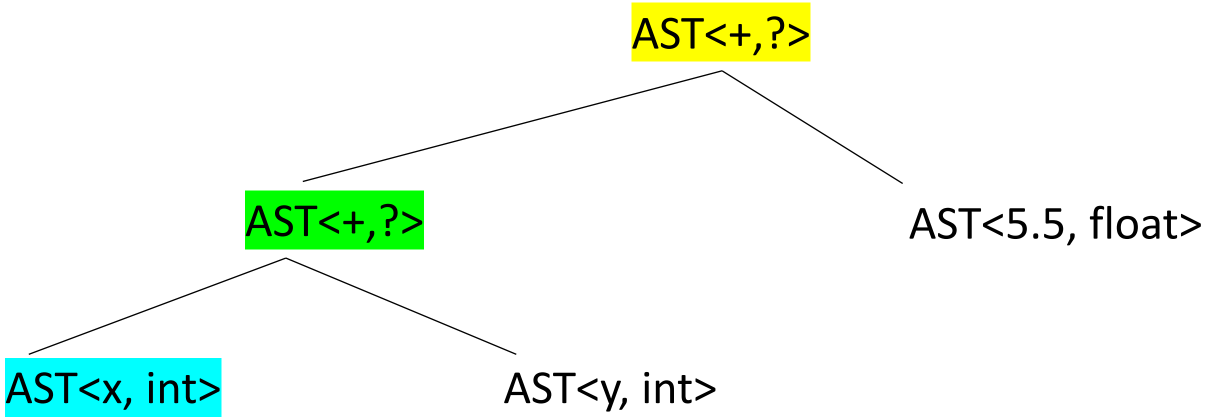
AST<5.5, float>

AST<x, int>

AST<y, int>

Type inference

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



AST<+,?>

AST<+,?>

AST<5.5, float>

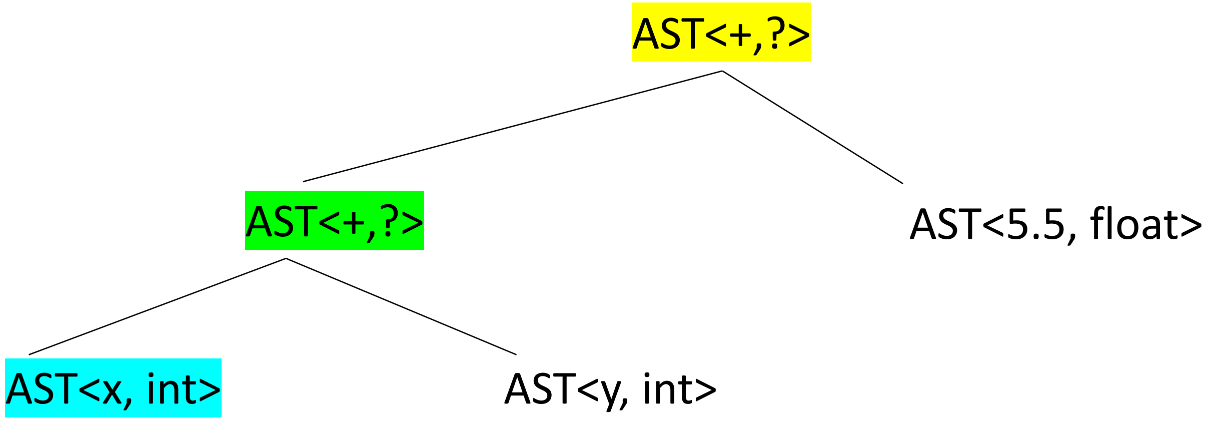
AST<x, int>

AST<y, int>

```
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

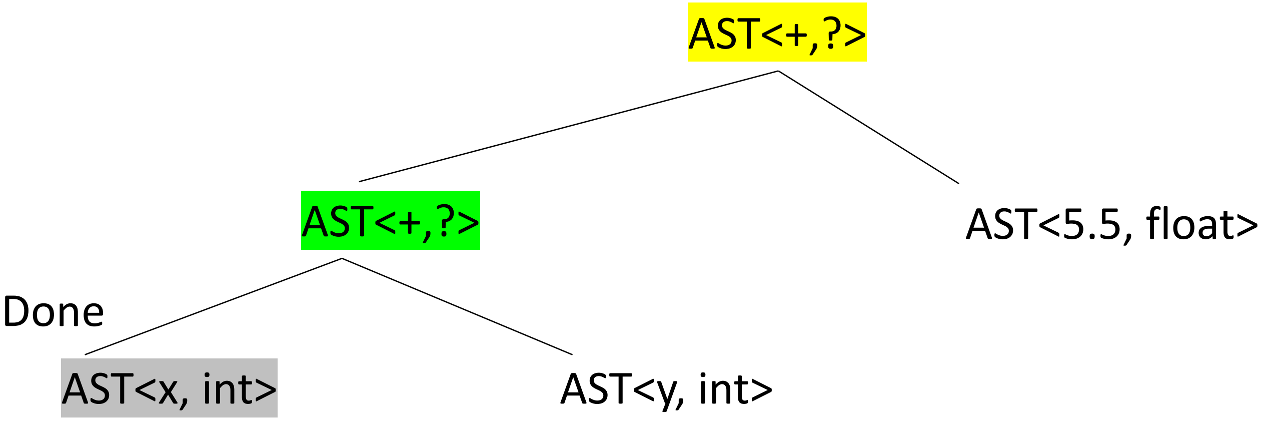
```
int x;
int y;
float w;
w = x + y + 5.5
```



```
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

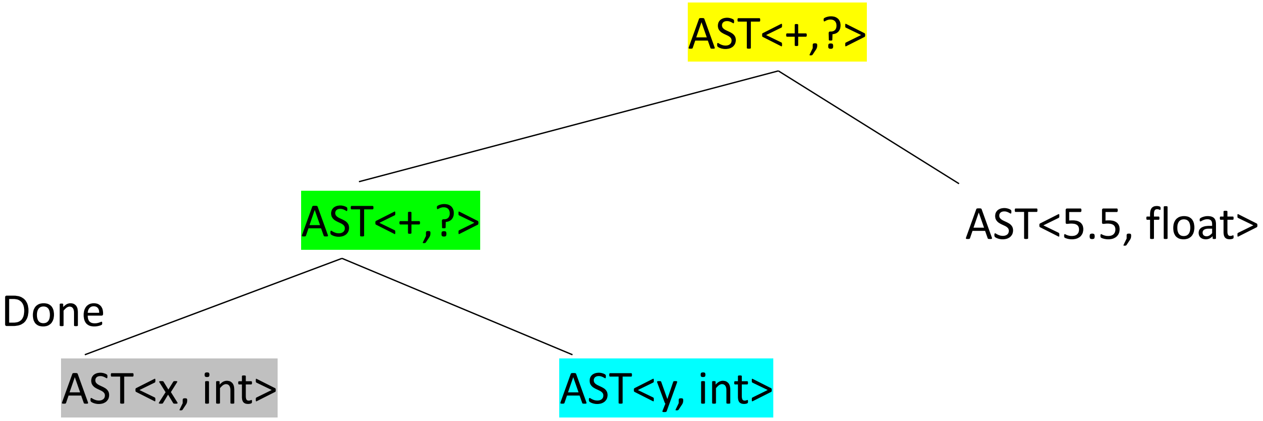
```
int x;
int y;
float w;
w = x + y + 5.5
```



```
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

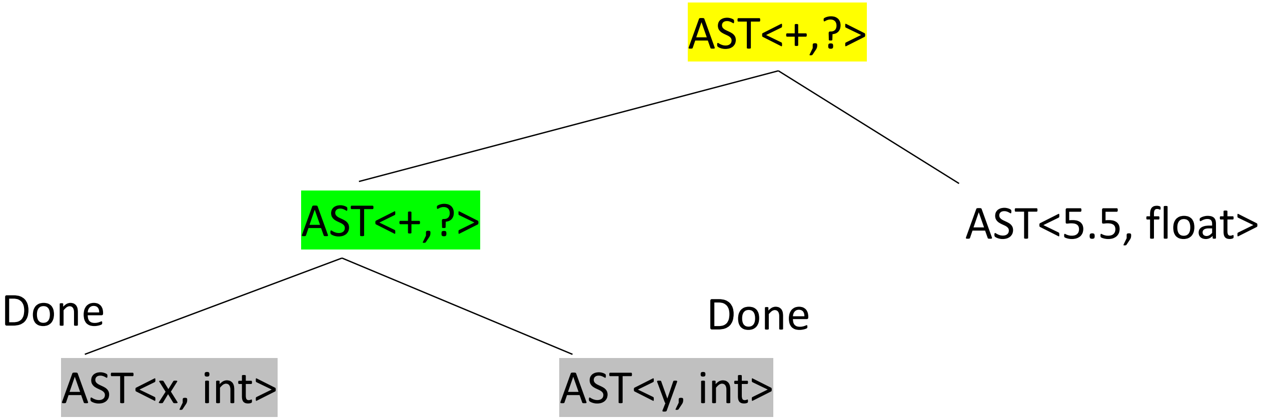
```
int x;
int y;
float w;
w = x + y + 5.5
```



```
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

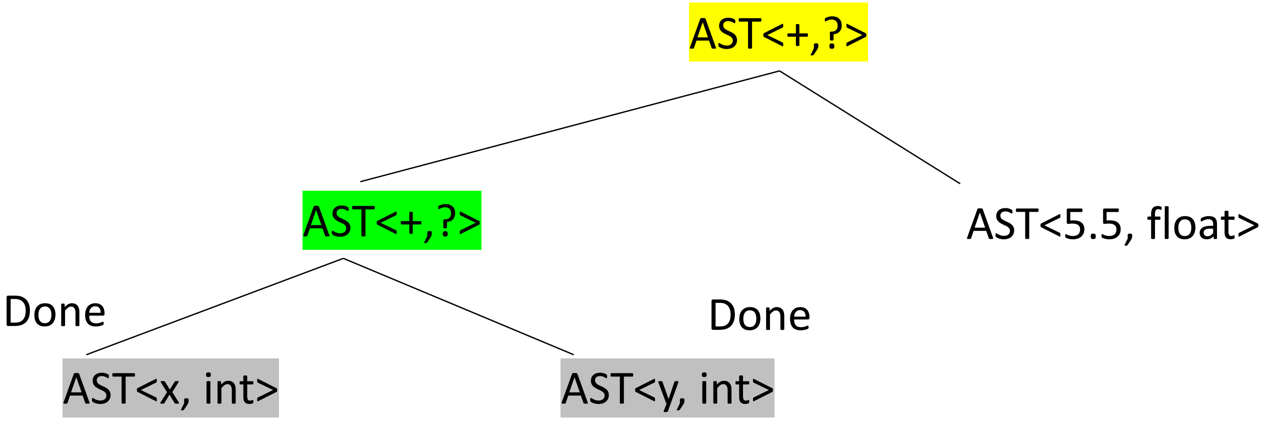
```
int x;
int y;
float w;
w = x + y + 5.5
```



```
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
```

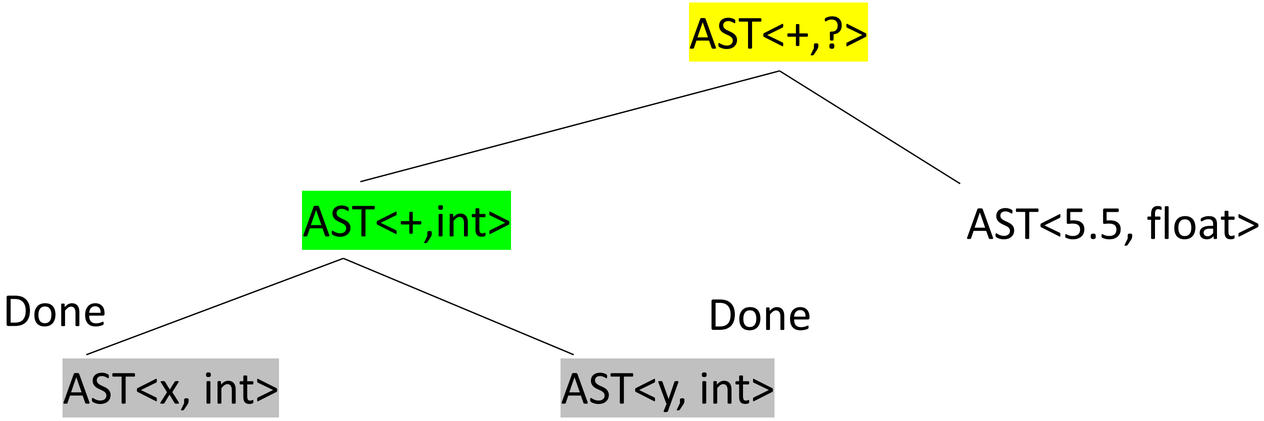
Table for most binary ops

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
```



AST<+,?>

AST<+,int>

AST<5.5, float>

Done

Done

AST<x, int>

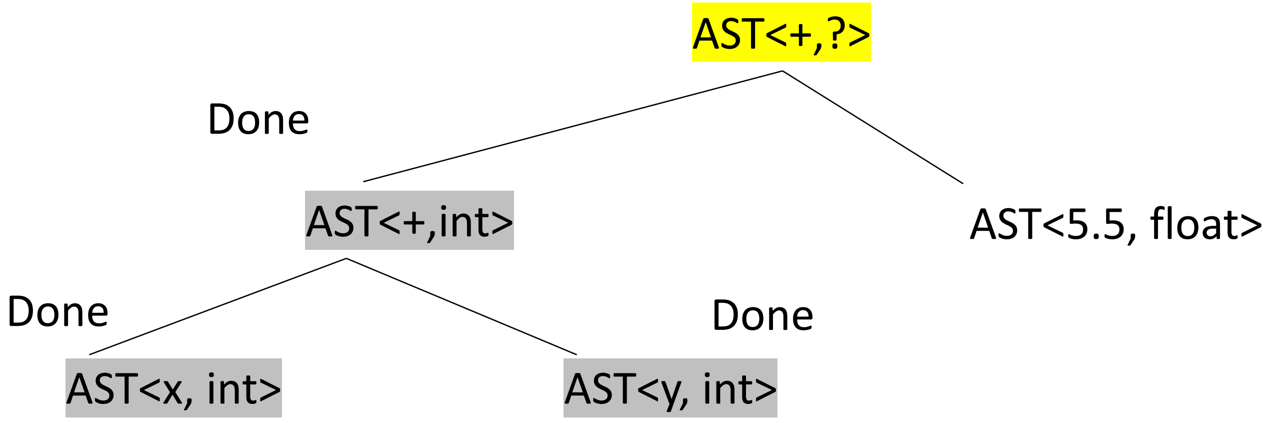
AST<y, int>

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
```

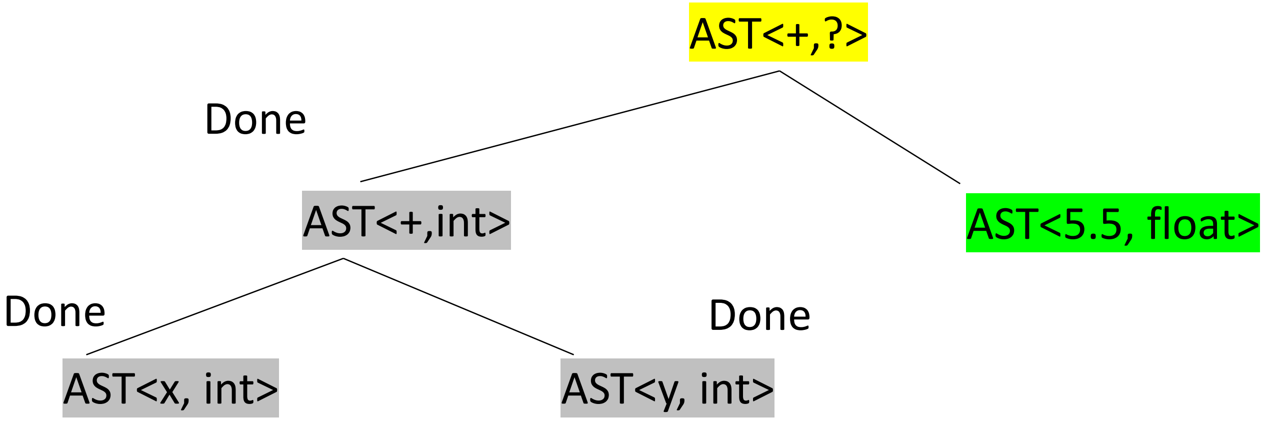
Table for most binary ops

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
```



AST<+,?>

Done

AST<+,int>

AST<5.5, float>

Done

Done

AST<x, int>

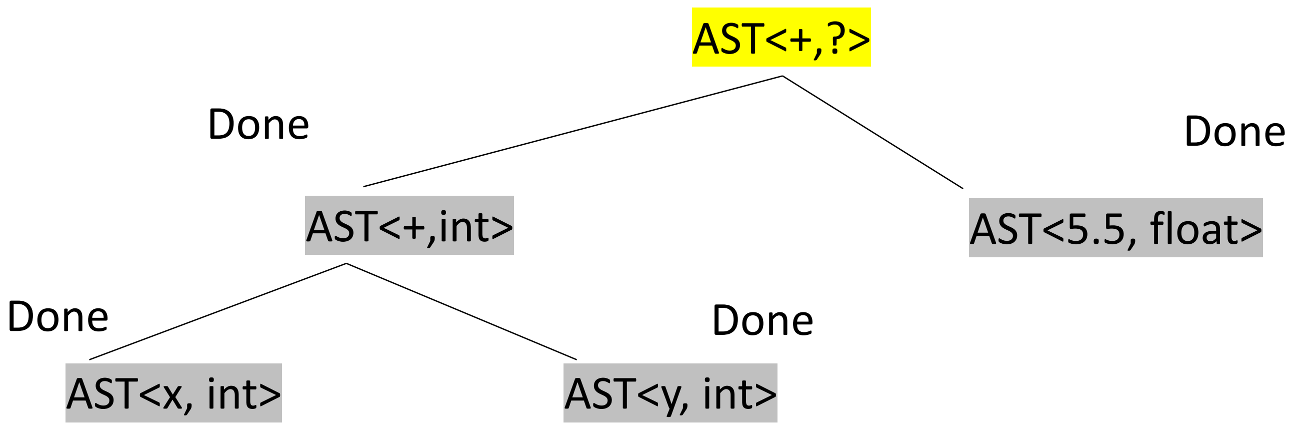
AST<y, int>

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
```



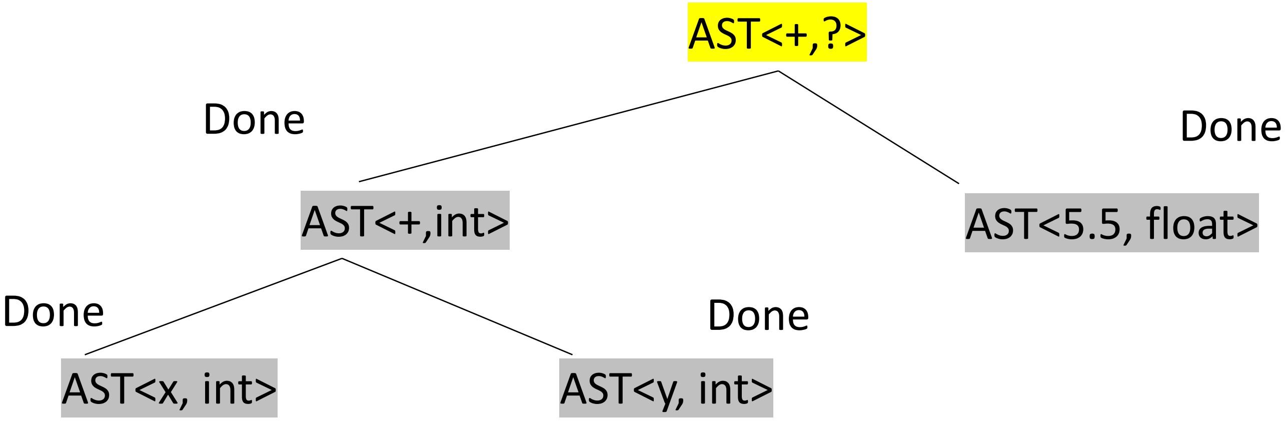
```
def type_inference(n):

  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
```



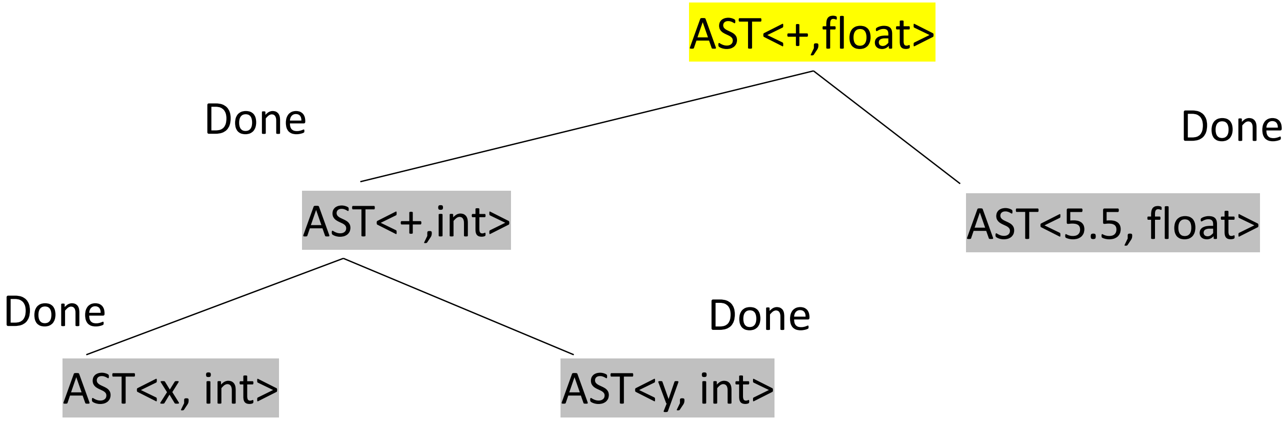
```
def type_inference(n):

    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
```



AST<+,float>

Done

Done

AST<+,int>

AST<5.5, float>

Done

Done

AST<x, int>

AST<y, int>

```
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
```

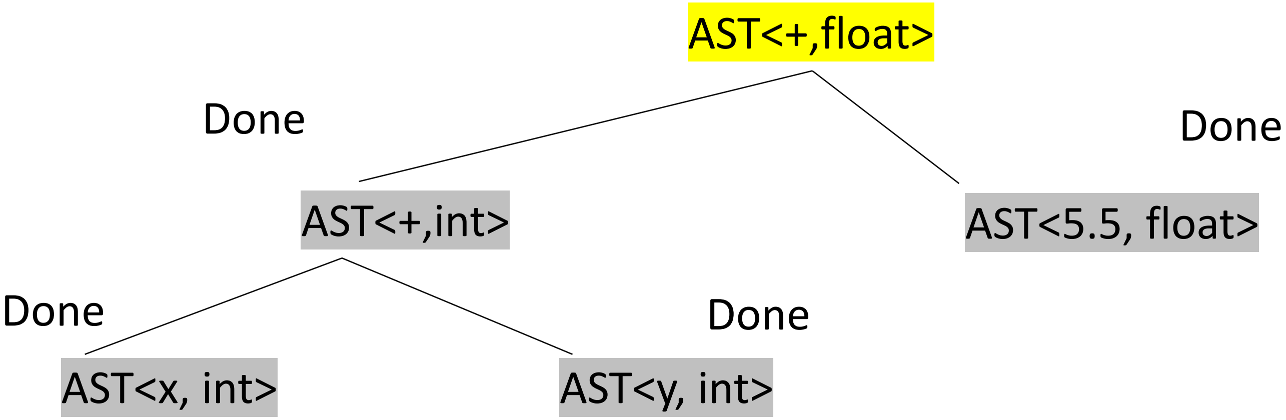
Table for most binary ops

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
```



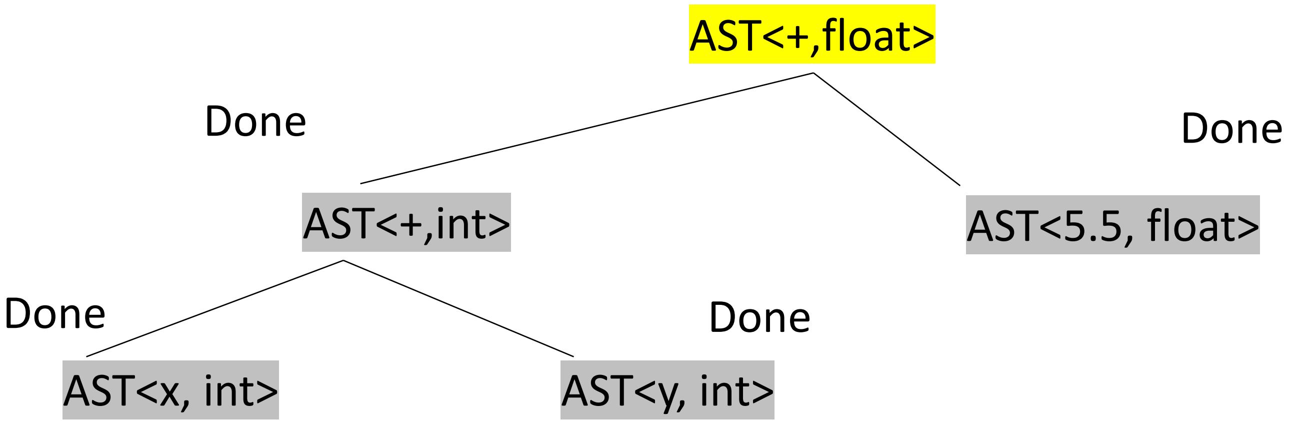
Are we done?

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
  do any required type conversions
  return t
```

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



AST<+,float>

Done

AST<+,int>

Done

AST<5.5, float>

Done

AST<x, int>

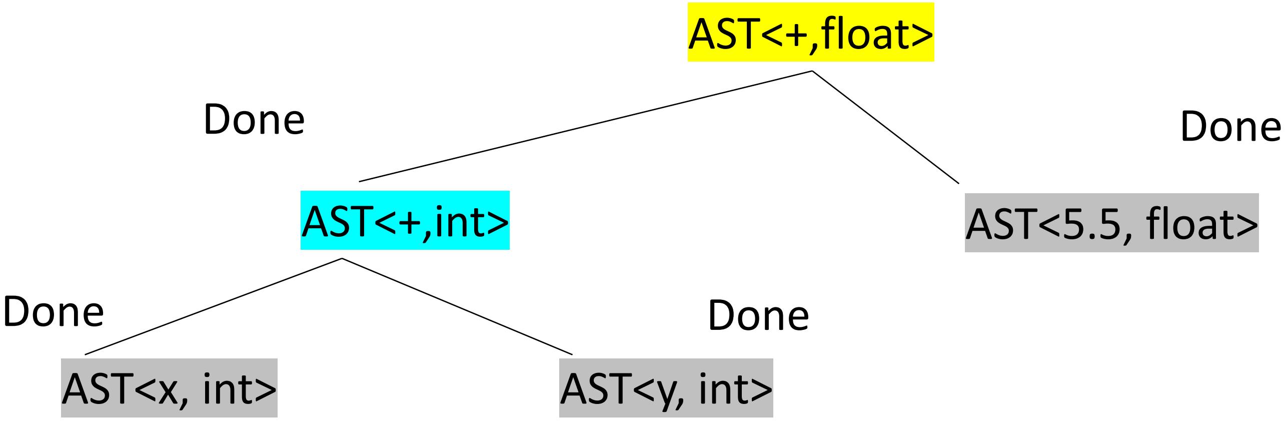
Done

AST<y, int>

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

```
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)
```

New type of AST nodes: unary operators

```
from enum import Enum

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

what types are these nodes?

```
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)
```

New type of AST nodes: unary operators

```python
from enum import Enum

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

what types are these nodes?

```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)
```

New type of AST nodes: unary operators

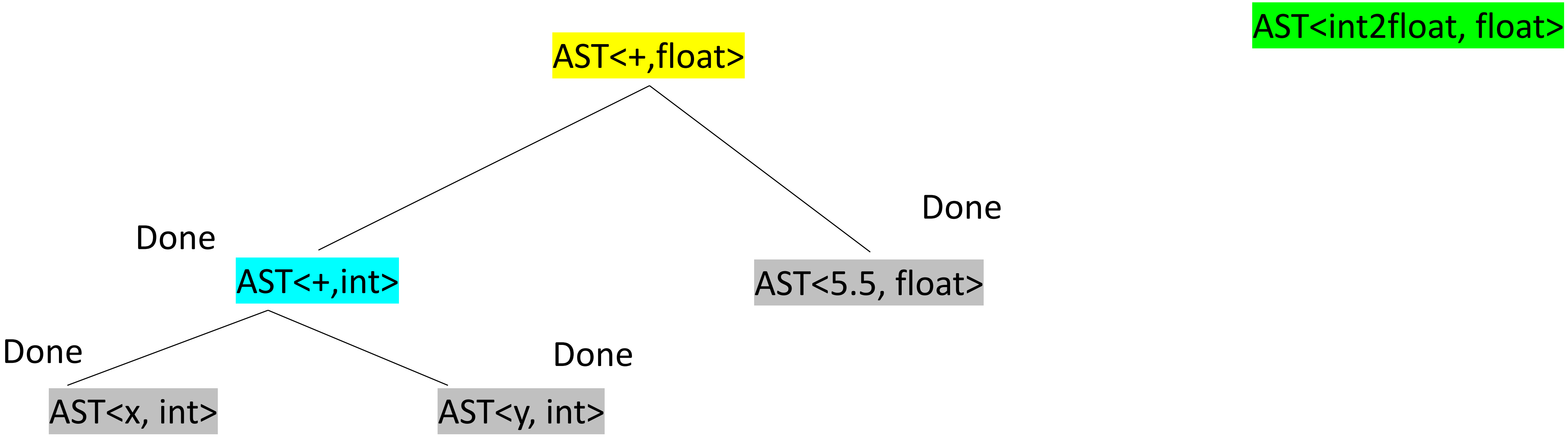
```
from enum import Enum

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

what types are these nodes?

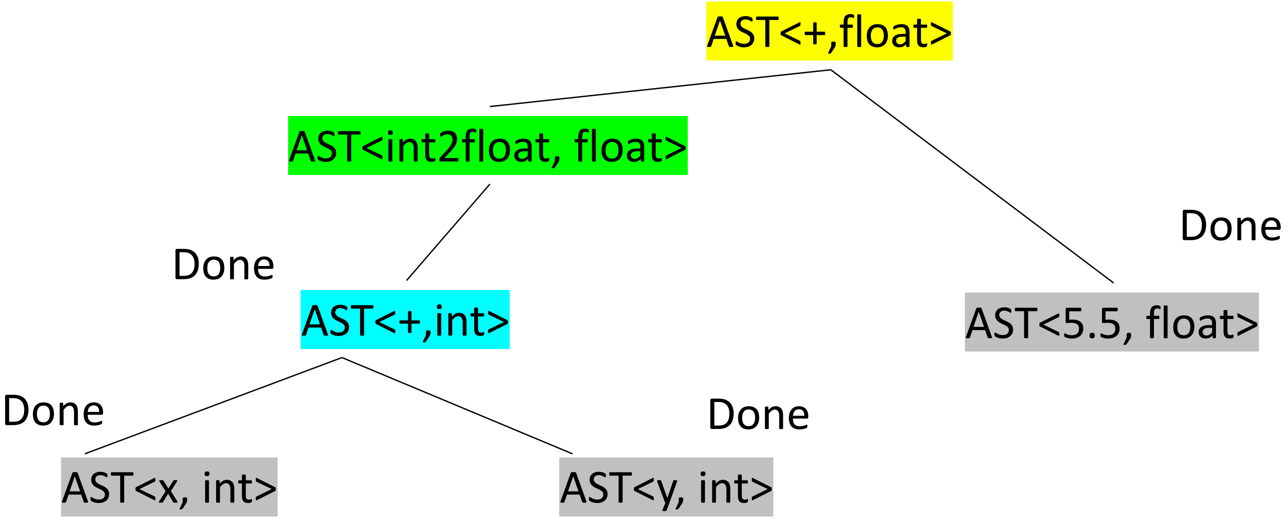
```
class ASTUnOpNode(ASTNode):
    def __init__(self, child):
        self.child = child


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)
```

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

```
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
```

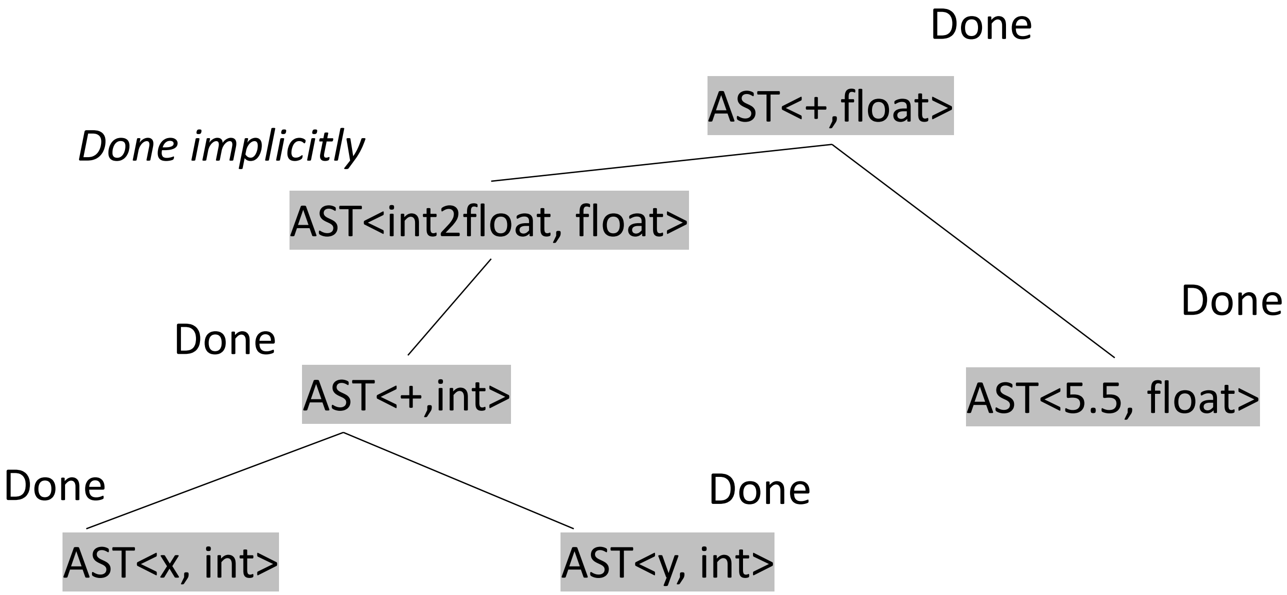


```
def type_conversion(n):

    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
```

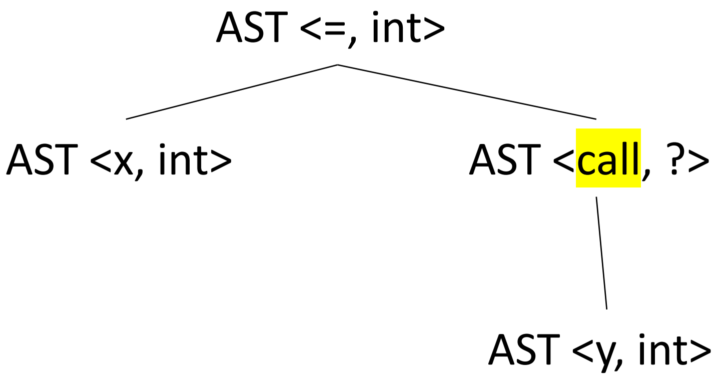


```
def type_inference(n):

    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

How are functions handled?

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



AST <=, int>

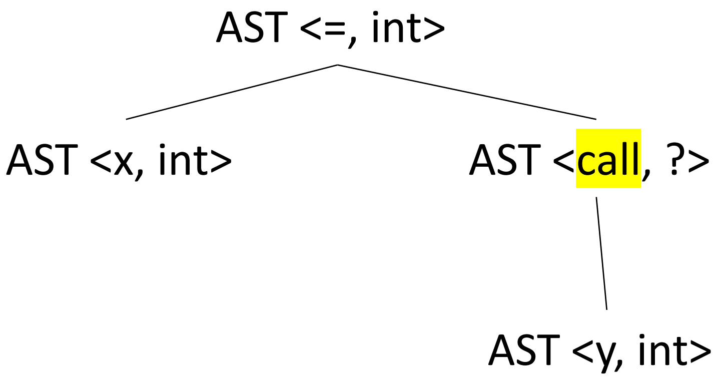
AST <x, int>

AST <call, ?>

AST <y, int>

How are functions handled?

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



AST <=, int>

AST <x, int>

AST <call, ?>

AST <y, int>

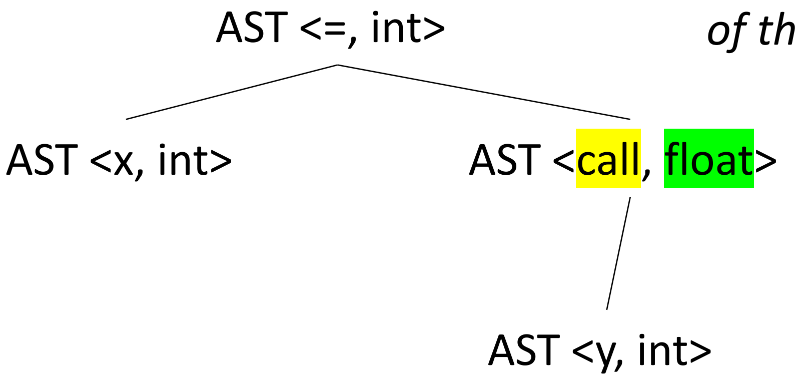
requires a function specification,
using in the .h file:

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

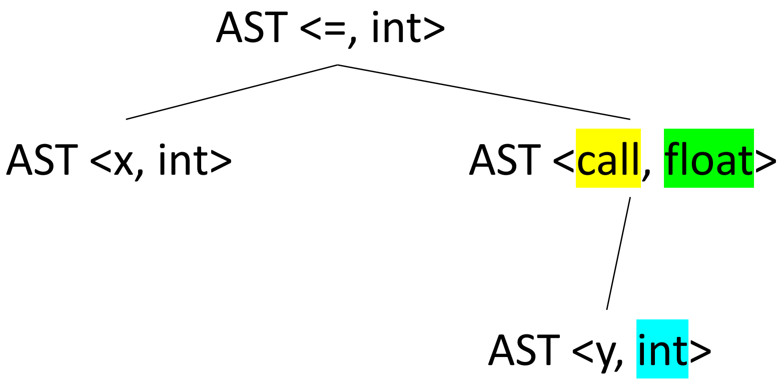
requires a function specification,
using in the .h file:

```
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)
```



AST <=, int>

AST <x, int>

AST <call, float>

AST <y, int>

type inference must make sure arguments match types

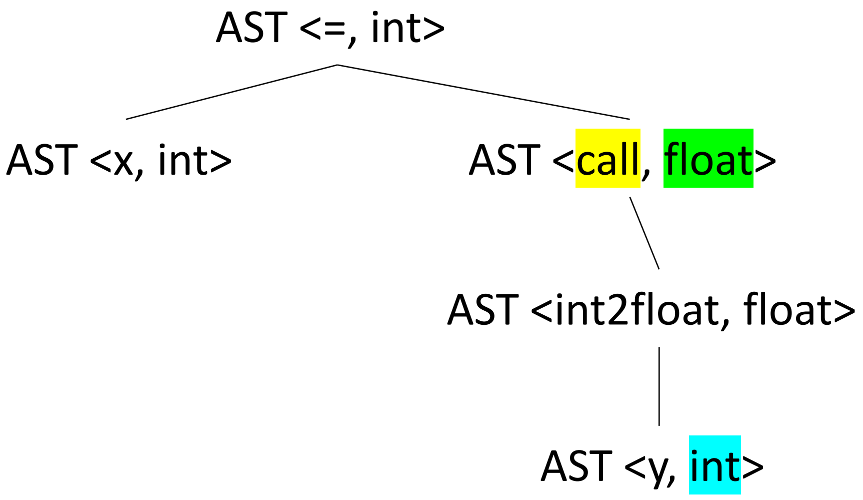
requires a function specification, using in the .h file:

```
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)
```



AST <=, int>

AST <x, int>

AST <call, float>

AST <int2float, float>

AST <y, int>

type inference must make sure arguments match types

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

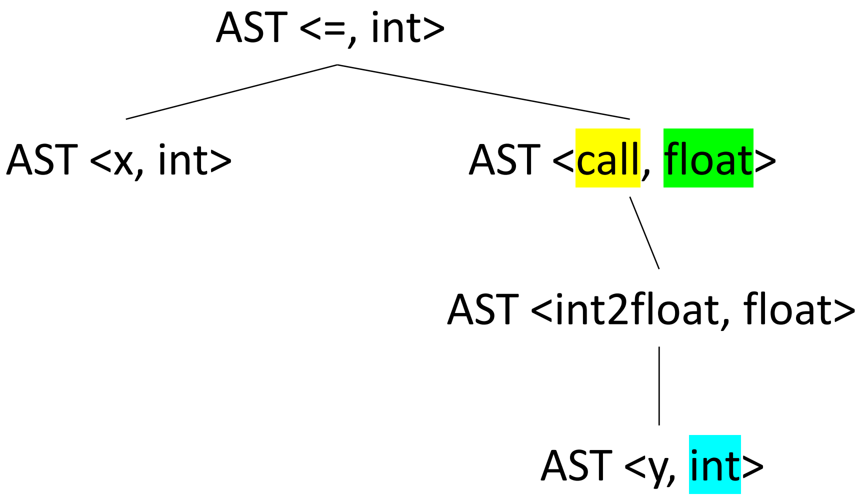
```
float sqrt(float x);
```

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?



AST <=, int>

AST <x, int>

AST <call, float>

AST <int2float, float>

AST <y, int>

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

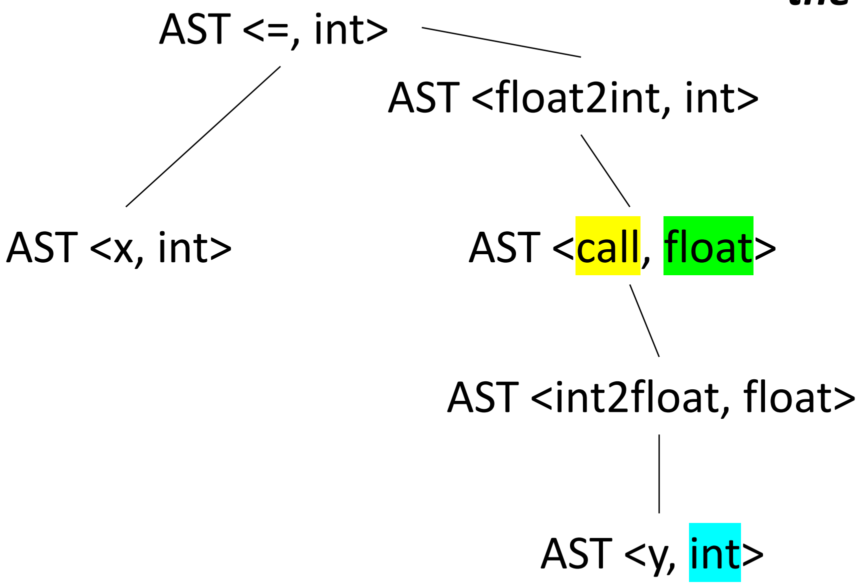
```
float sqrt(float x);
```

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)
```

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



AST <=, int>

AST <x, int>

AST <float2int, int>

AST <call, float>

AST <int2float, float>

AST <y, int>

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

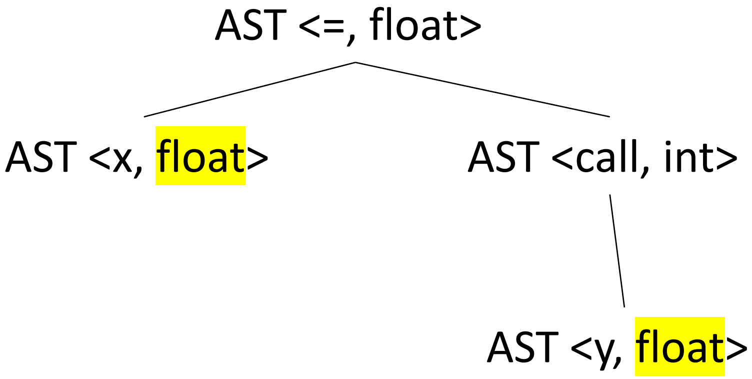
```
float sqrt(float x);
```

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

What about floats to ints?

```
int int_sqrt(int input);

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

Does this compile?



AST <=, float>

AST <x, float>

AST <call, int>

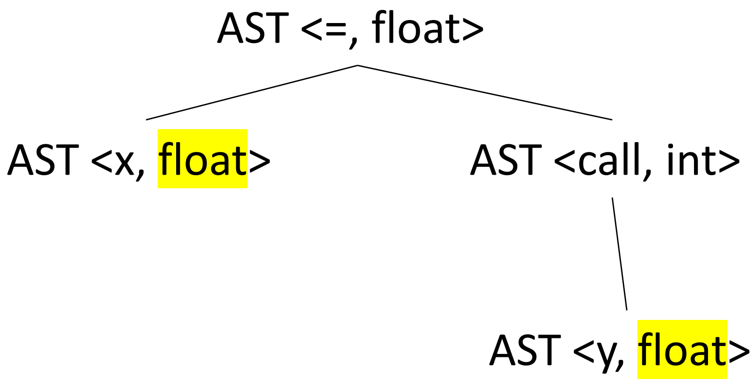
AST <y, float>

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? ...



AST <=, float>

AST <x, float>

AST <call, int>

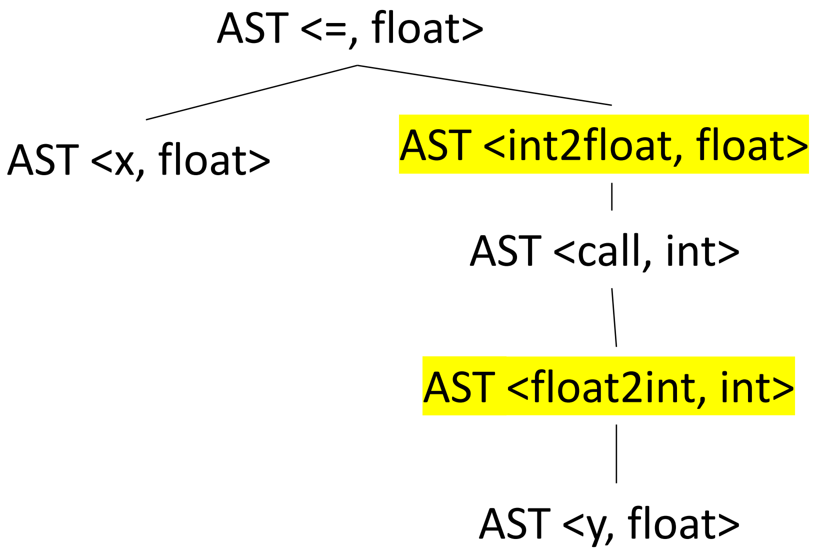
AST <y, float>

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

```
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

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