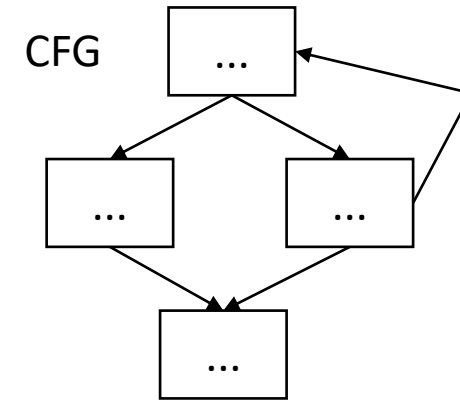
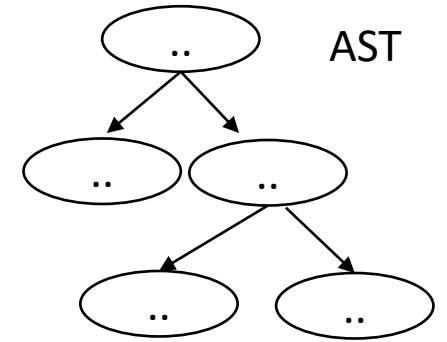


# CSE110A: Compilers

May 2, 2022

## Topics:

- *ASTs*
  - *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

- HW 1 grades are released
  - Let us know in 1 week if there are any issues
  - Please let us know through a private piazza post
- We plan to grade midterm and midterm next week
- HW 3 is due on Monday
  - No guaranteed help during the weekend

# Homework 3 notes

- Issue with test cases: please fix according to Rithik's Piazza post
- One issue with the provided grammar:

## Example grammar

```
comp := factor comp2    {NUM, ID, LPAR}  
comp2 := LT factor expr2 {LT}  
      | ""              {SEMI, RPAR, EQ}
```

Is that `expr2` in the reference grammar correct? It seems like that would create a single less than statement followed by equivalence statements, which feels wrong. Should it not be `comp2`?

# Homework 3 notes

- Output:
  - If the string is a valid program, then the program does nothing. It just terminates normally
  - If the string is not, then it should throw an exception:
    - Scanner Exception
    - Parser Exception
    - Symbol Table Exception
- What to print if you want to test/debug?

# Homework 3 notes

- What information for each variable does the symbol table hold?
  - For this assignment, nothing! It just keeps track of which variables have been declared and in which scope.
  - For the next homework we will add type information to the symbol table

# Quiz

# Quiz

Both parse trees and ASTs are explicitly created using node classes. These trees can then be traversed and analyzed.

---

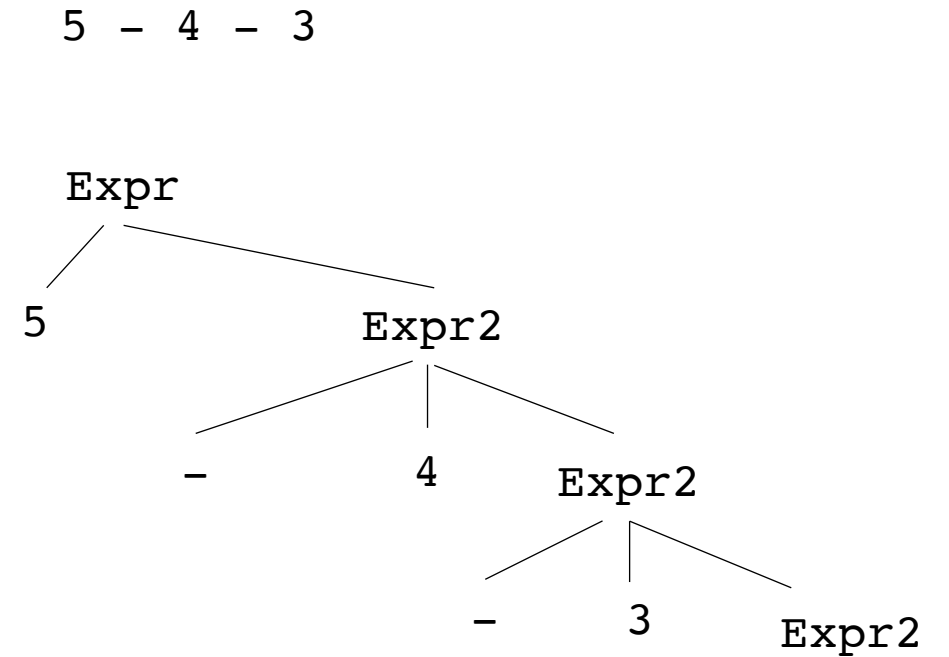
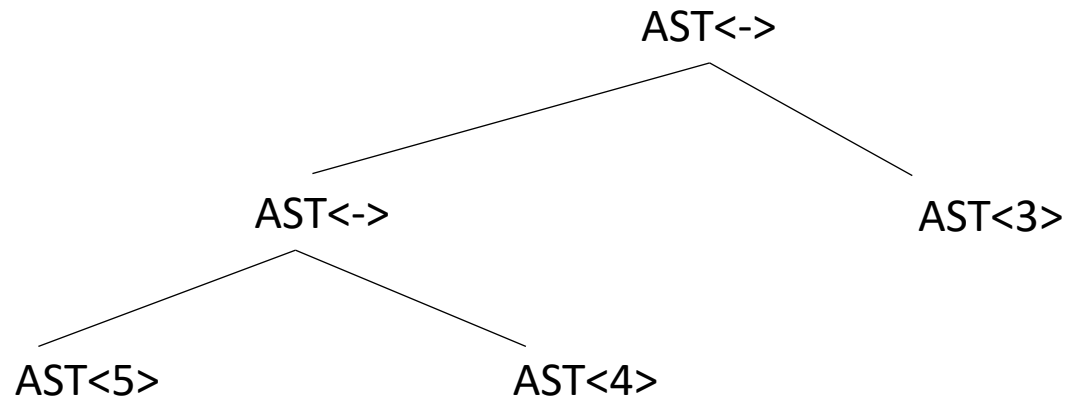
☐ True

---

☐ False

# Creating an AST from predictive grammar

Expr	::=	NUM	Expr2
Expr2	::=	MINUS	NUM Expr2
			""



*How do we get to the desired parse tree?*



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

```
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)
```

```
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)
```

# Quiz

If you have a left recursive grammar for expressions, you can create an AST entirely using production actions

---

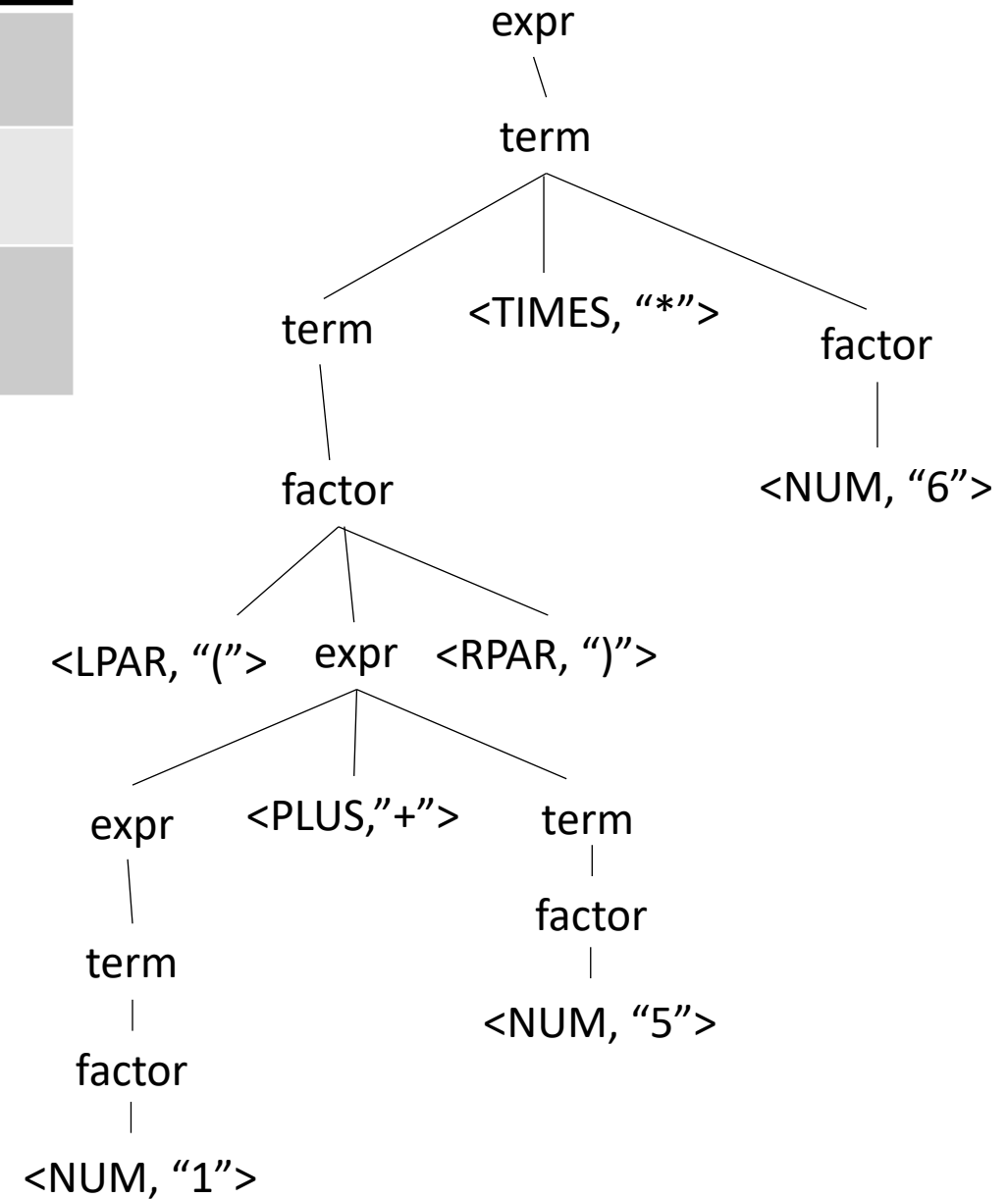
☐ True

---

☐ False

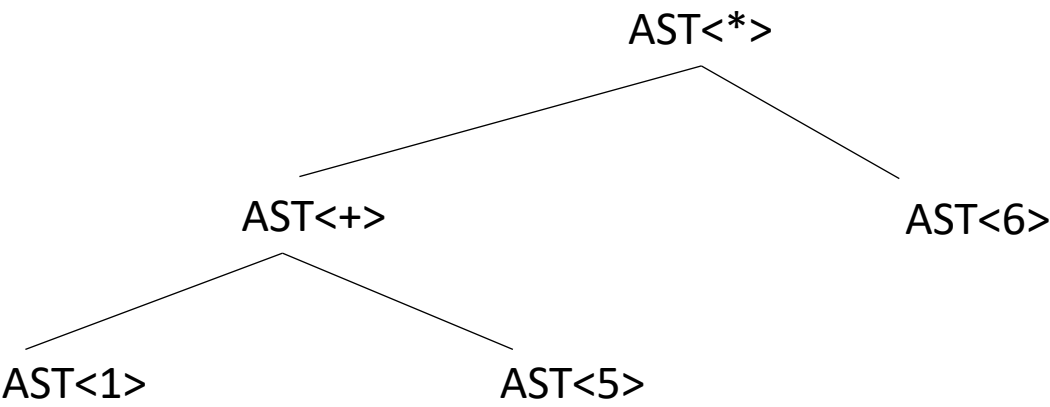
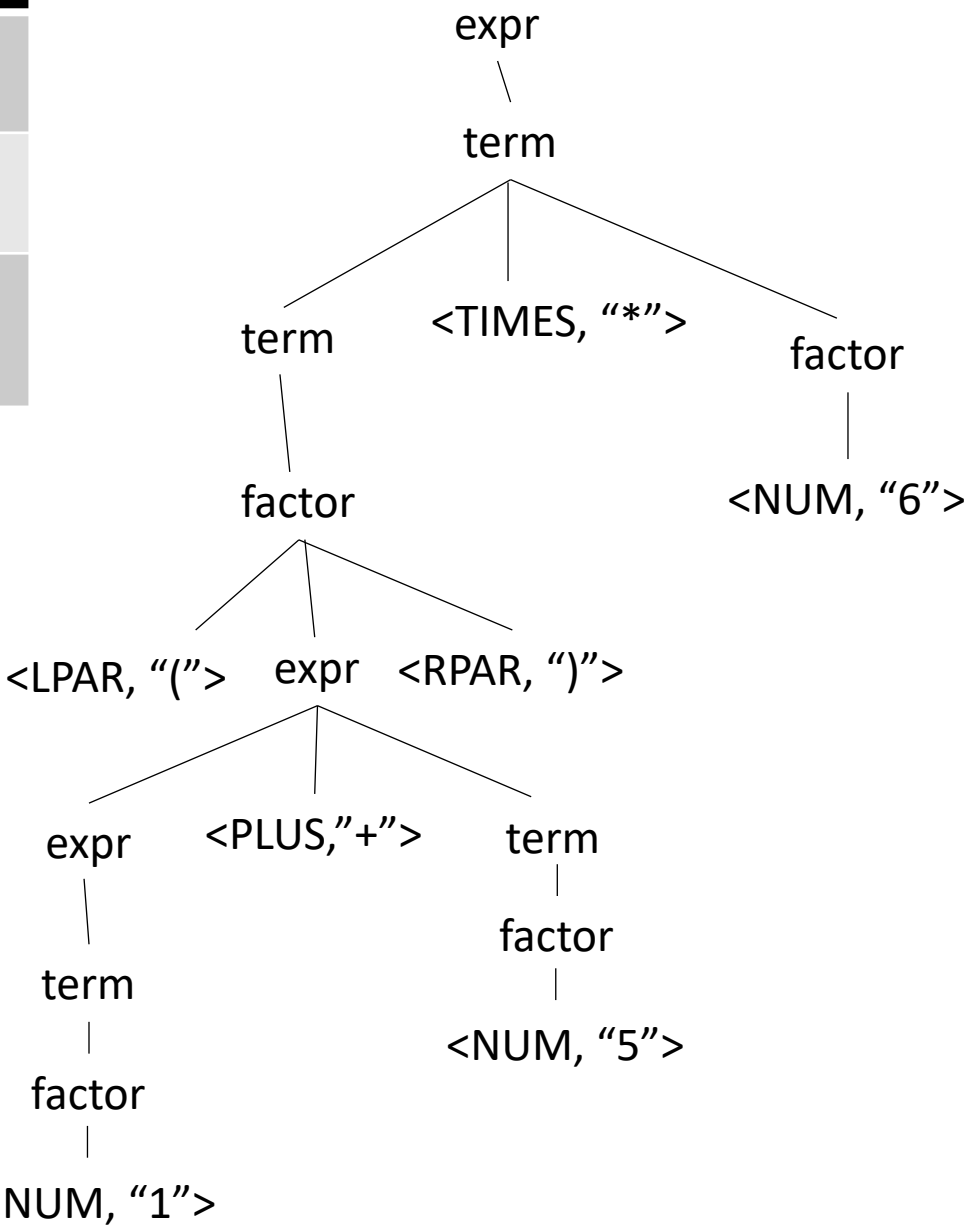
Name	Productions	Production action
expr	: expr PLUS term   term	{return ASTAddNode(\$1,\$3)} {return \$1}
term	: term TIMES factor   factor	{return ASTMultNode(\$1,\$3)} {return \$1}
factor	: LPAR expr RPAR   NUM   ID	{return \$2} {return ASTNumNode(\$1)} {return ASTIDNode(\$1)}

input: (1+5)\*6



Name	Productions	Production action
expr	: expr PLUS term   term	{return ASTAddNode(\$1,\$3)} {return \$1}
term	: term TIMES factor   factor	{return ASTMultNode(\$1,\$3)} {return \$1}
factor	: LPAR expr RPAR   NUM   ID	{return \$2} {return ASTNumNode(\$1)} {return ASTIDNode(\$1)}

input: (1+5)\*6



# Quiz

AST leaf nodes contain which of the following:

---

☐ a lexeme

---

☐ a number

---

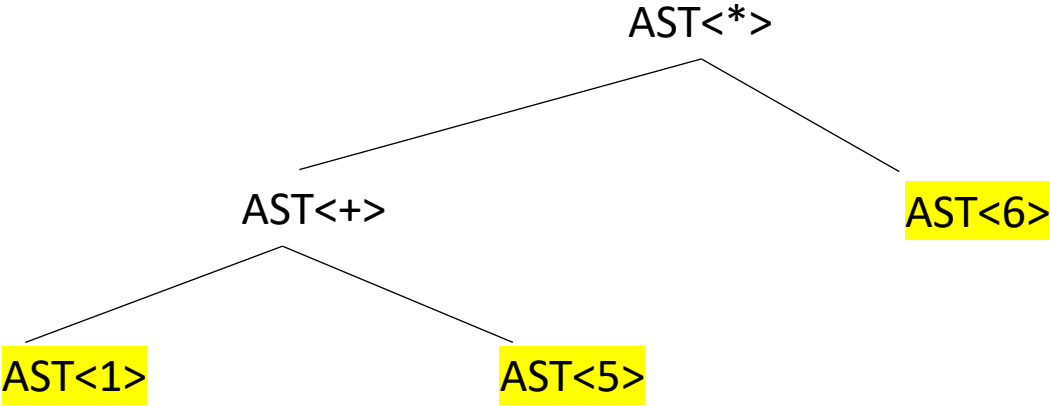
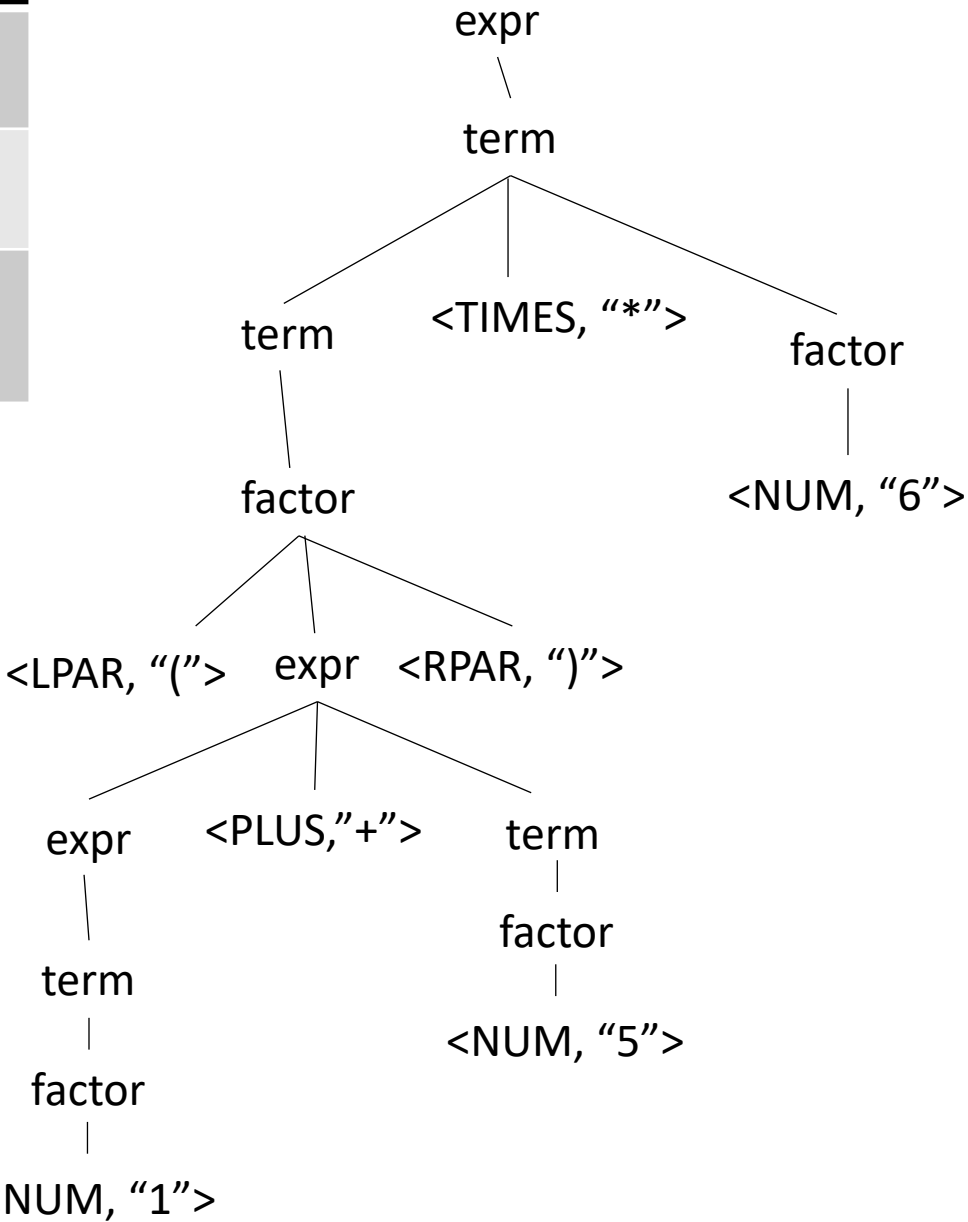
☐ an id

---

☐ a function call

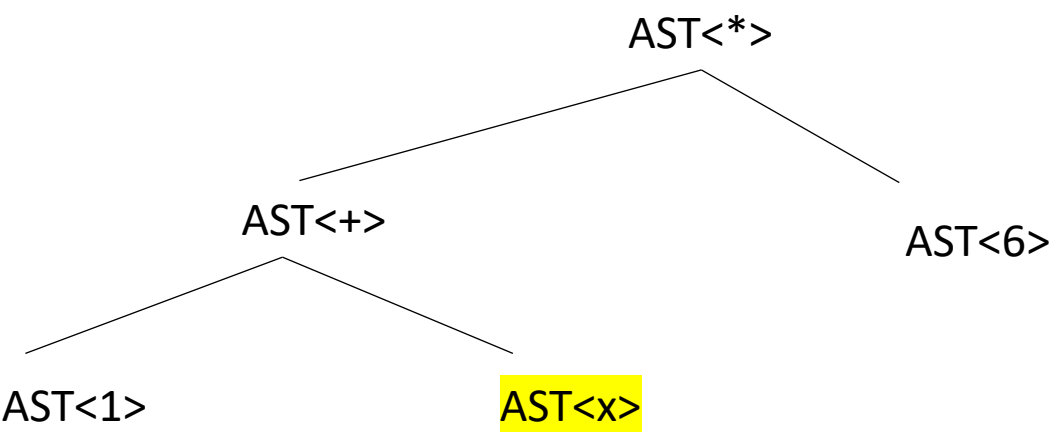
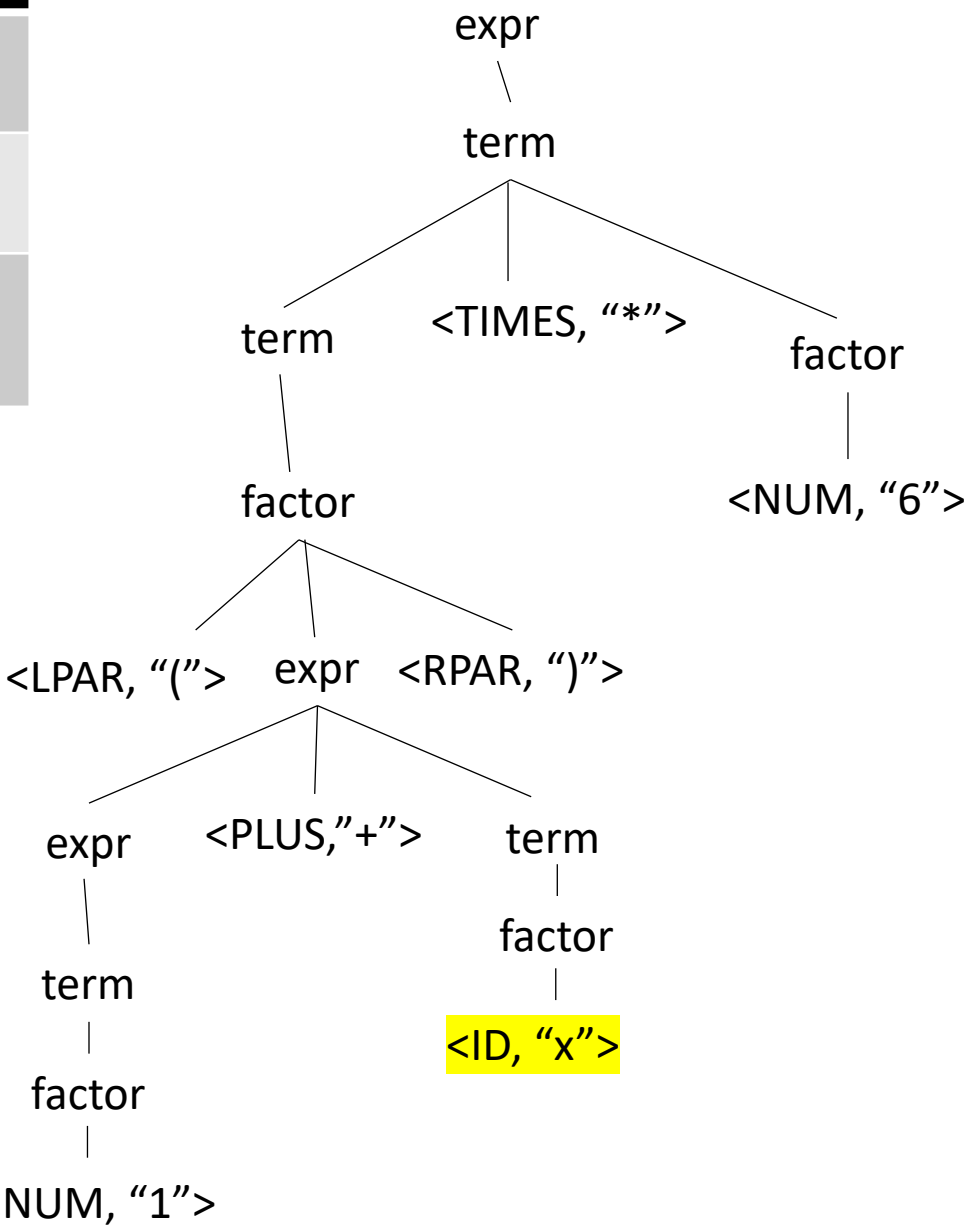
Name	Productions	Production action
expr	: expr PLUS term   term	{return ASTAddNode(\$1,\$3)} {return \$1}
term	: term TIMES factor   factor	{return ASTMultNode(\$1,\$3)} {return \$1}
factor	: LPAR expr RPAR   NUM   ID	{return \$2} {return ASTNumNode(\$1)} {return ASTIDNode(\$1)}

input: (1+5)\*6



Name	Productions	Production action
expr	: expr PLUS term   term	{return ASTAddNode(\$1,\$3)} {return \$1}
term	: term TIMES factor   factor	{return ASTMultNode(\$1,\$3)} {return \$1}
factor	: LPAR expr RPAR   NUM   ID	{return \$2} {return ASTNumNode(\$1)} {return ASTIDNode(\$1)}

input: (1+x)\*6



# Quiz

AST leaf nodes contain which of the following:

---

☐ a lexeme

---

☐ a number

---

☐ an id

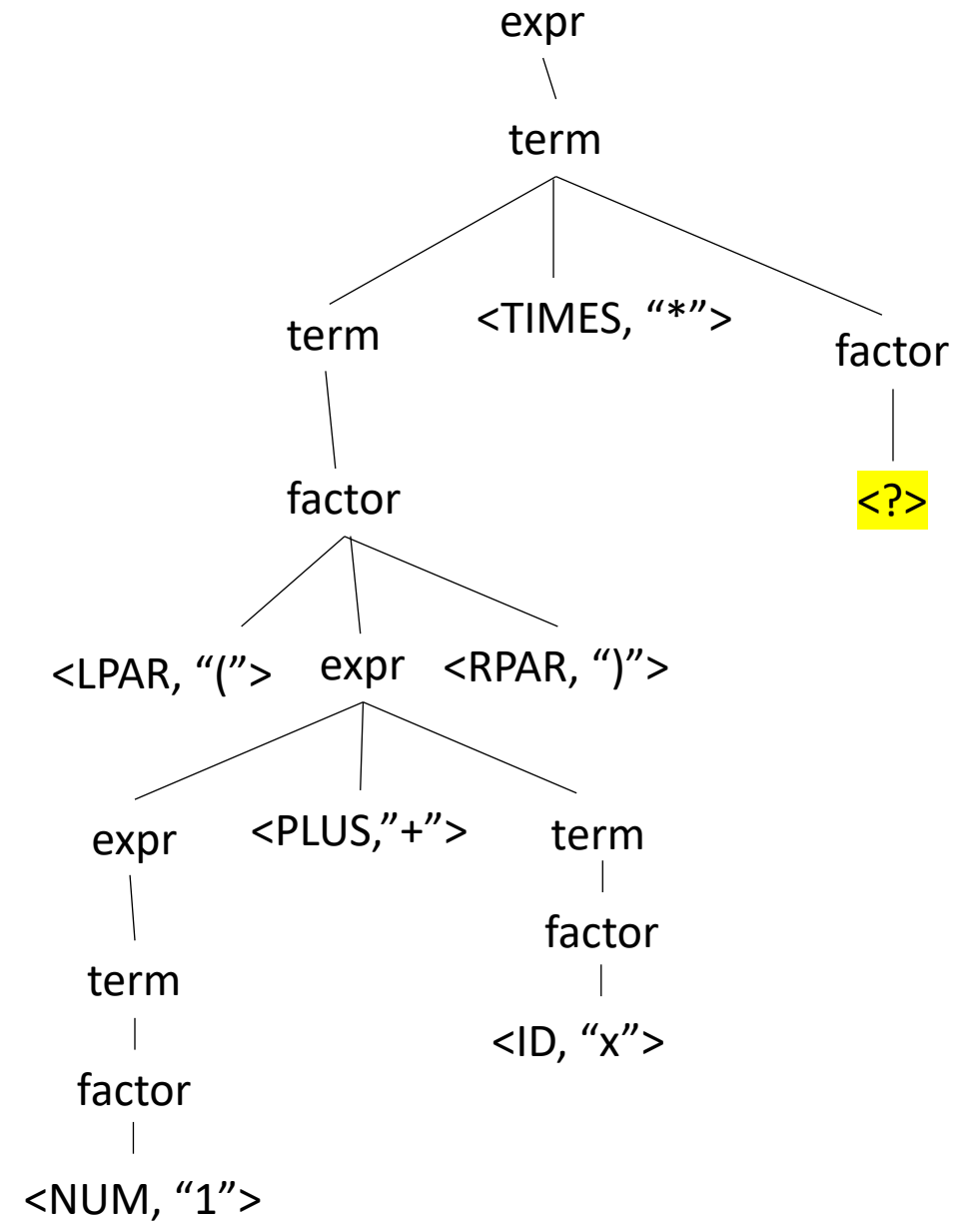
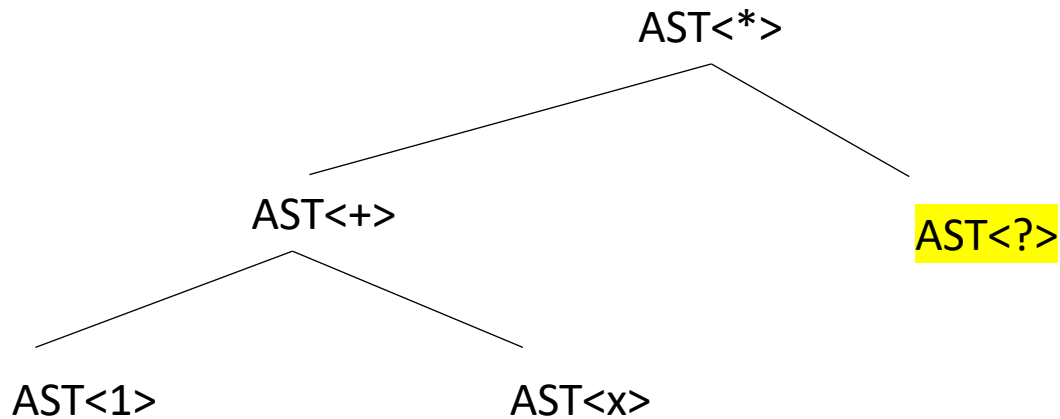
---

☒ a function call



*Our language doesn't have function calls,  
but what do we think?*

$(1+x) * \text{sqrt}(x)$



# Quiz

Write a few sentences about the differences between a parse tree and an AST

# New material

- Type systems
  - Evaluating an AST
  - Type systems
  - Type checking

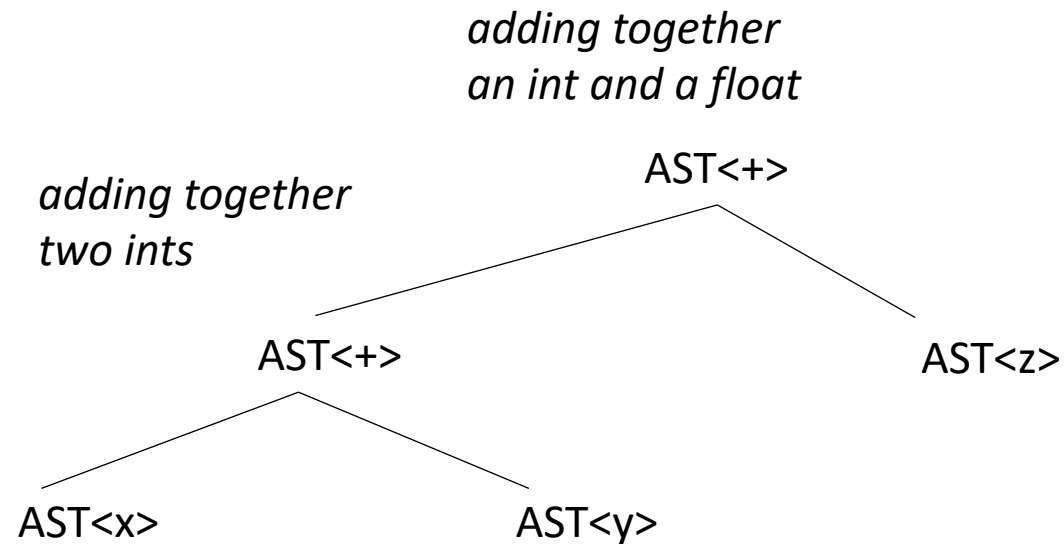
# Evaluate an AST by doing a post order traversal

Expr	::=	NUM	Expr2
Expr2	::=	PLUS	NUM Expr2
		" "	

*What if you cannot evaluate it?  
What else might you do?*

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

*How does this change things?*

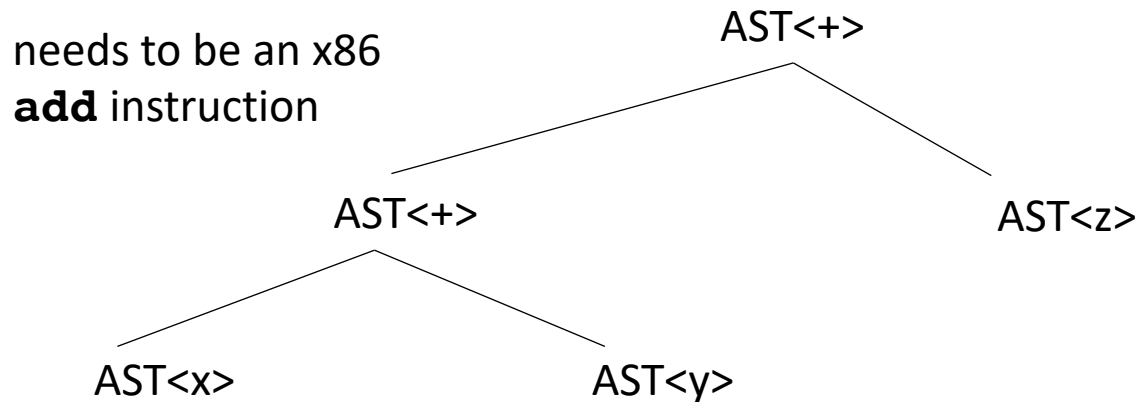


in many languages this is fine, but we are working towards assembly language

# Evaluate an AST by doing a post order traversal

```
Expr ::= NUM Expr2
Expr2 ::= PLUS NUM Expr2
      | ""
```

needs to be an x86  
**addss** 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

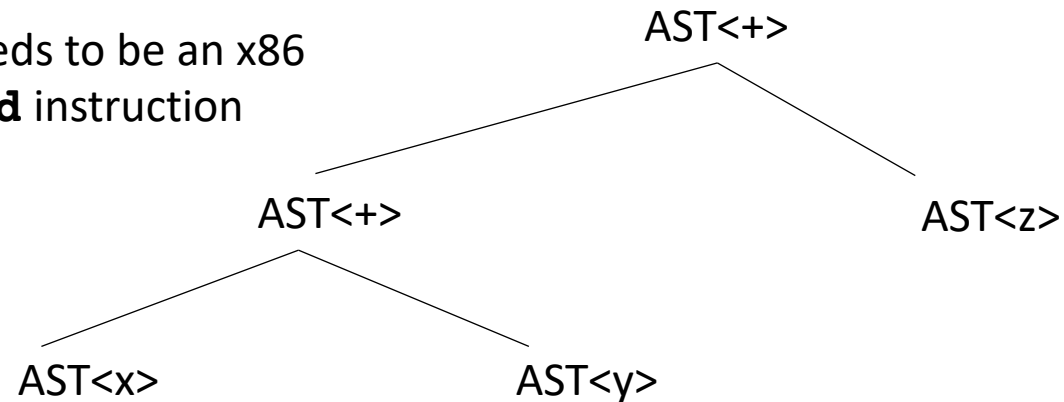
# Evaluate an AST by doing a post order traversal

Expr	::=	NUM	Expr2
Expr2	::=	PLUS	NUM Expr2
			""

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

needs to be an x86  
**addss** instruction

needs to be an x86  
**add** instruction



Lets do some experiments.

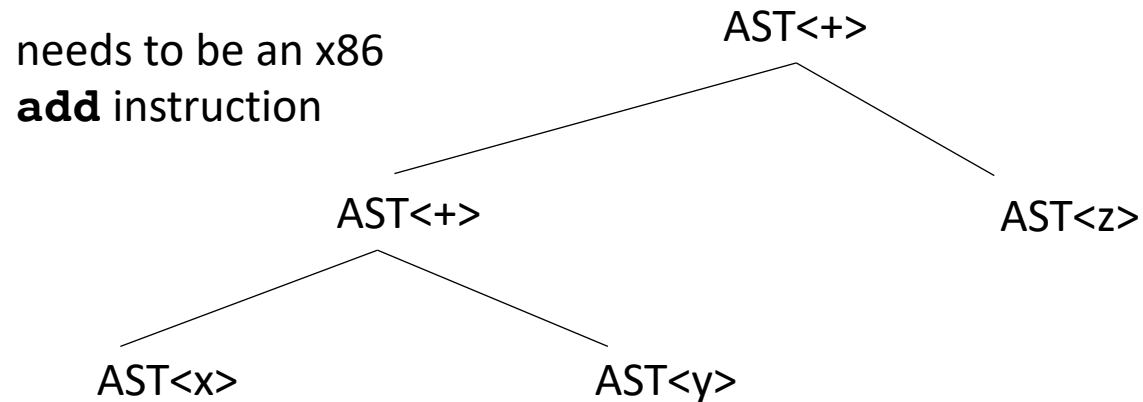
What should `5 + 5.0` be?

*Is this all?*

# Evaluate an AST by doing a post order traversal

Expr	::=	NUM	Expr2
Expr2	::=	PLUS	NUM Expr2
		" "	

needs to be an x86  
**addss** instruction



*Is this all?*

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

Lets do some experiments.

What should 5 + 5.0 be?

but

**addss r1 r2**

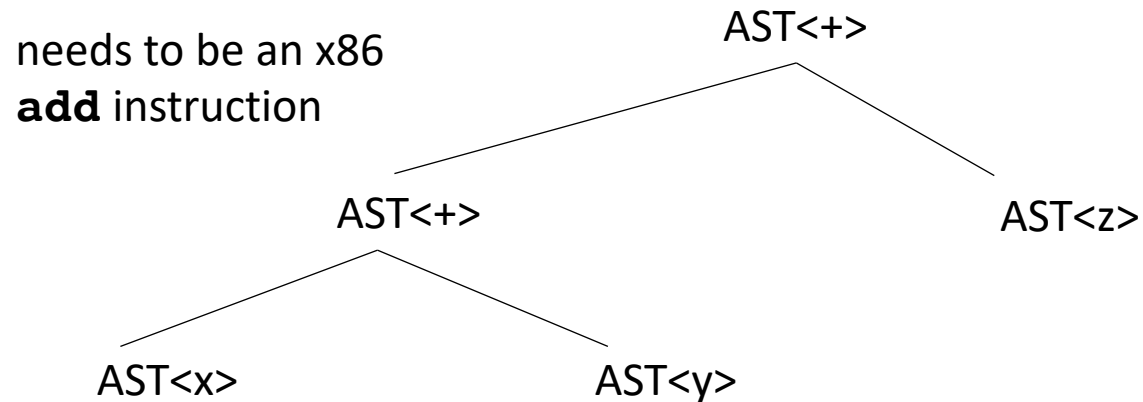
interprets both registers  
as floats

# Evaluate an AST by doing a post order traversal

Expr	::=	NUM	Expr2
Expr2	::=	PLUS	NUM Expr2
			""

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

needs to be an x86  
**addss** instruction



But the binary of 5 is 0b101  
the float value of 0b101 is 7.00649232162e-45

We cannot just add them!

*Is this all?*

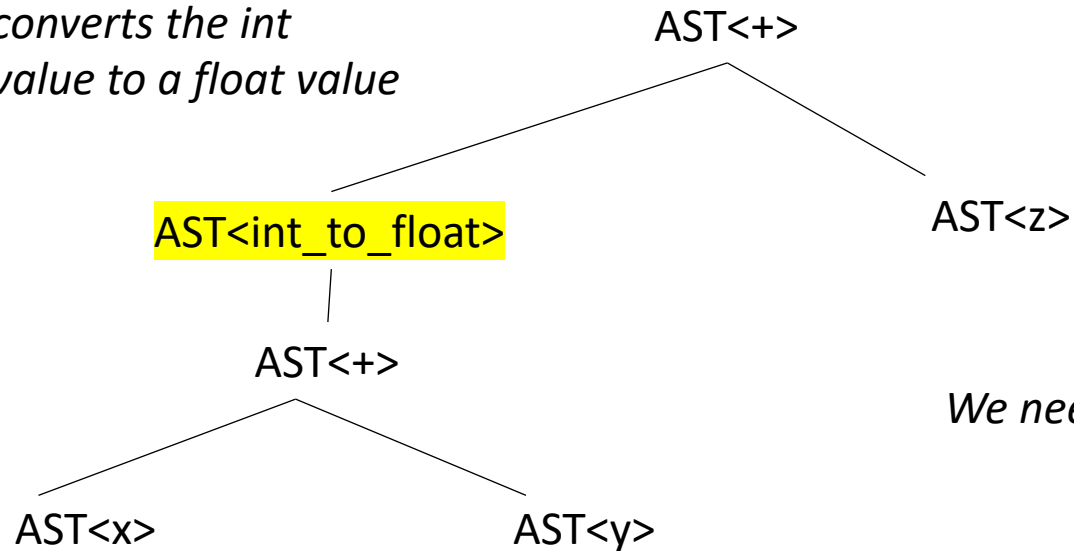


# Evaluate an AST by doing a post order traversal

```
Expr ::= NUM Expr2
Expr2 ::= PLUS NUM Expr2
      | ""
```

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

*converts the int  
value to a float value*



*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

- 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

- Different types of Type Systems for languages:
  - **statically typed**: types can be determined at compile time
  - **dynamically typed**: types are determined at runtime
  - **untyped**: the language has no types
- What are examples of each?
- What are pros and cons of each?

# Type systems

- Different types of Type Systems for languages:
  - **statically typed**: types can be determined at compile time
  - **dynamically typed**: types are determined at runtime
  - **untyped**: the language has no types

do type conversion at compile time  
otherwise you have to check without  
static types, this would need to be  
translated to:

- What are examples of each?
- What are **pros** and cons of each?

$x + y$

```
if type(x) == int and type(y) == int:
    add(x,y)
if type(x) == int and type(y) == float:
    addss(int_to_float(x), y)
if ...
```

# Type systems

- Different types of Type Systems for languages:
  - **statically typed**: types can be determined at compile time
  - **dynamically typed**: types are determined at runtime
  - **untyped**: the language has no types

Can write more generic code

- What are examples of each?
- What are **pros** and cons of each?

```
def add(x,y):  
    return x + y
```

You would need to write many different functions for each type

# Type systems

- Different types of Type Systems for languages:
  - **statically typed**: types can be determined at compile time
  - **dynamically typed**: types are determined at runtime
  - **untyped**: the language has no types
- What are examples of each?
- What are **pros** and cons of each?

*Very close to assembly. You can write really optimized code. But very painful*

# Type systems

Considerations:

# Type systems

## Considerations:

- Base types in the language:
  - ints
  - chars
  - strings
  - floats
  - bool
- How to combine types in expressions:
  - int and float?
  - int and char?
  - int and bool?



# 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 systems

## Considerations:

- Base types:
  - ints
  - chars
  - strings
  - floats
  - bool
- How to combine types in expressions:
  - int and float?
  - int and char?
  - int and bool?

*What do each of these do if they are +’ed together?*

# Type checking

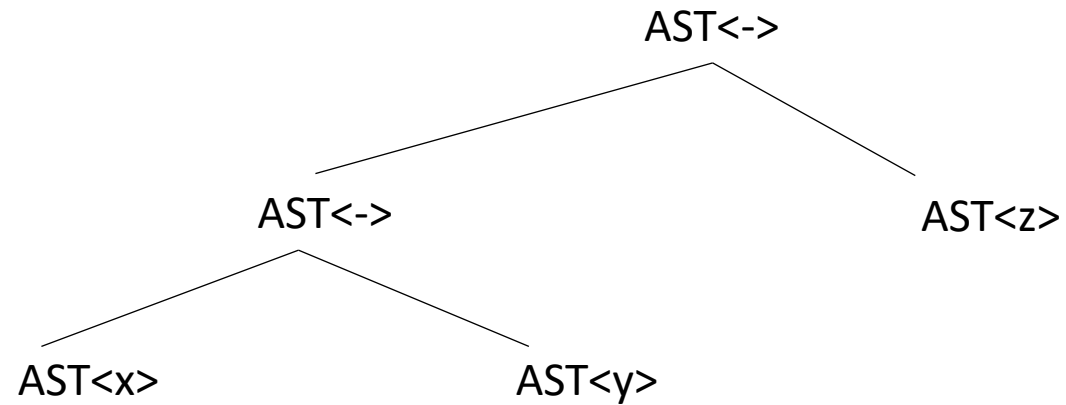
## Two components

- 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
```

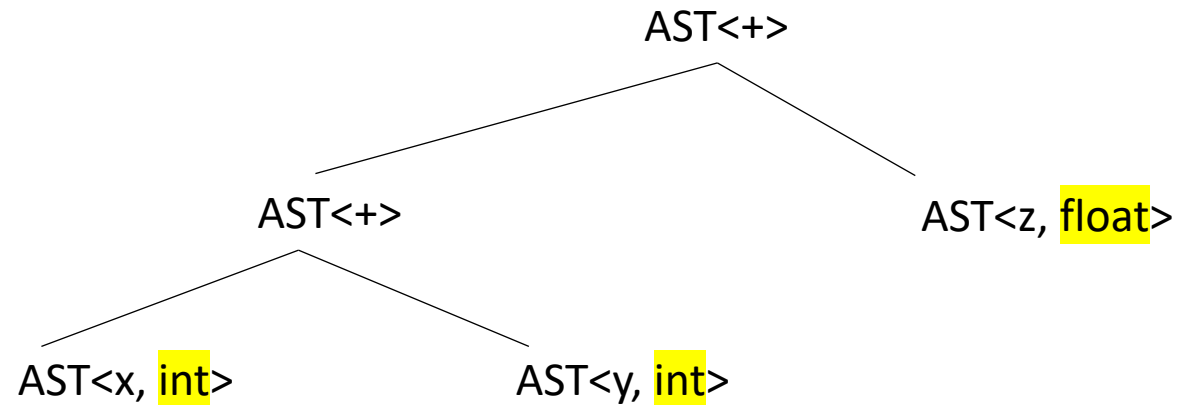
*each node additionally gets a type*



# Type checking on an AST

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

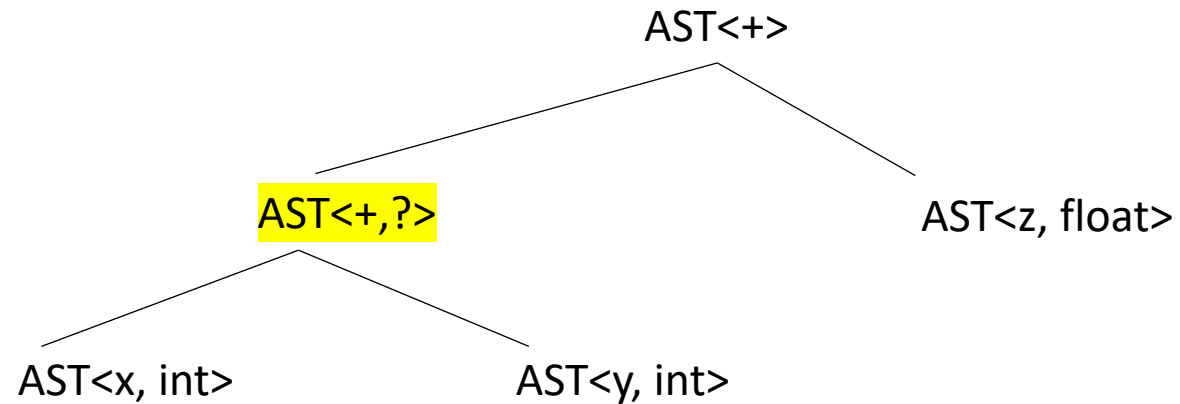
*each node additionally gets a type  
we can get this from the symbol table for the leaves or based  
on the input (e.g. 5 vs 5.0)*



# 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?*



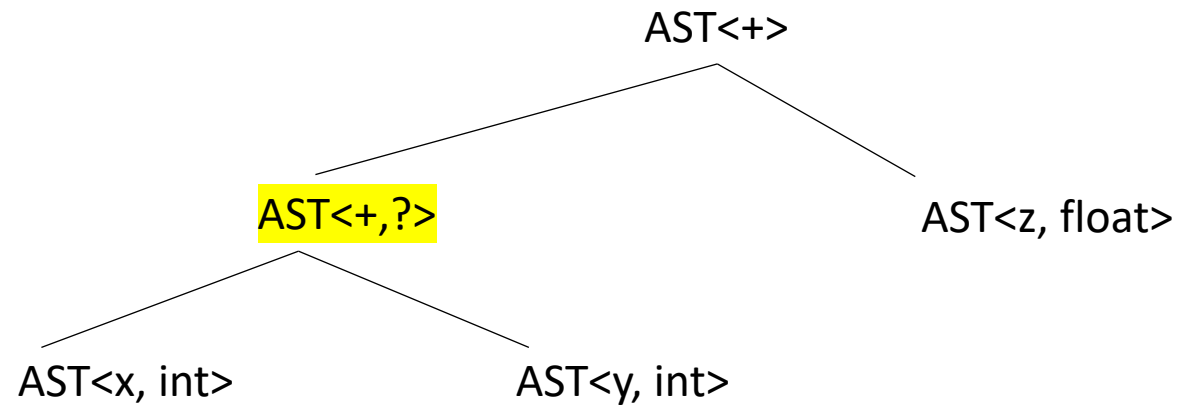
# Type checking on an AST

```
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



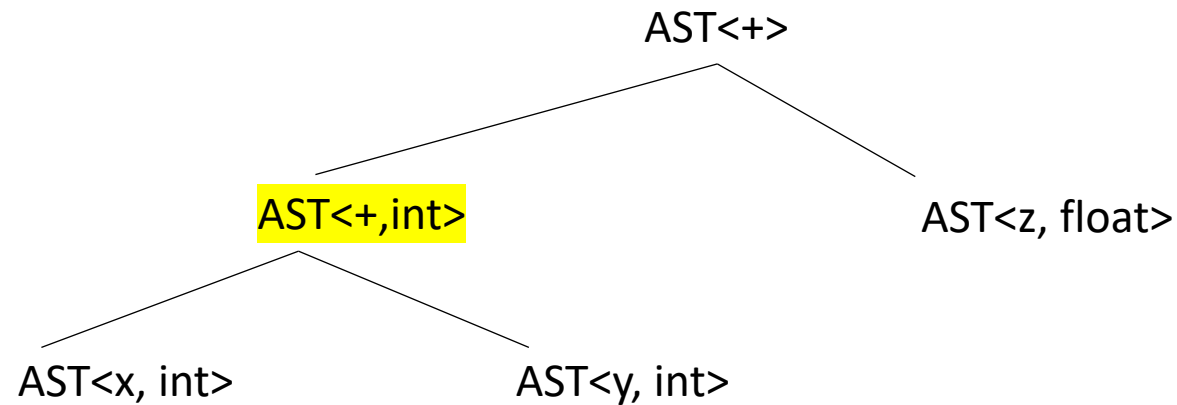
# Type checking on an AST

```
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





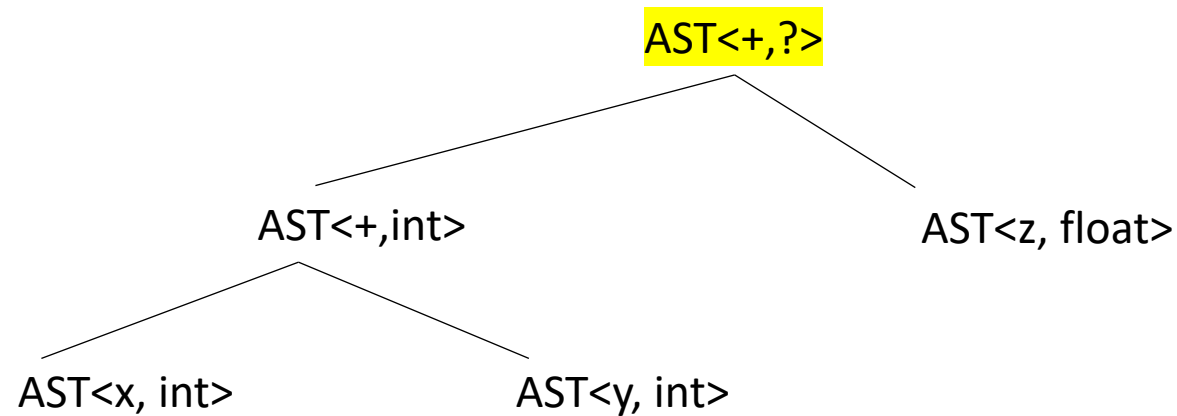
# Type checking on an AST

```
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



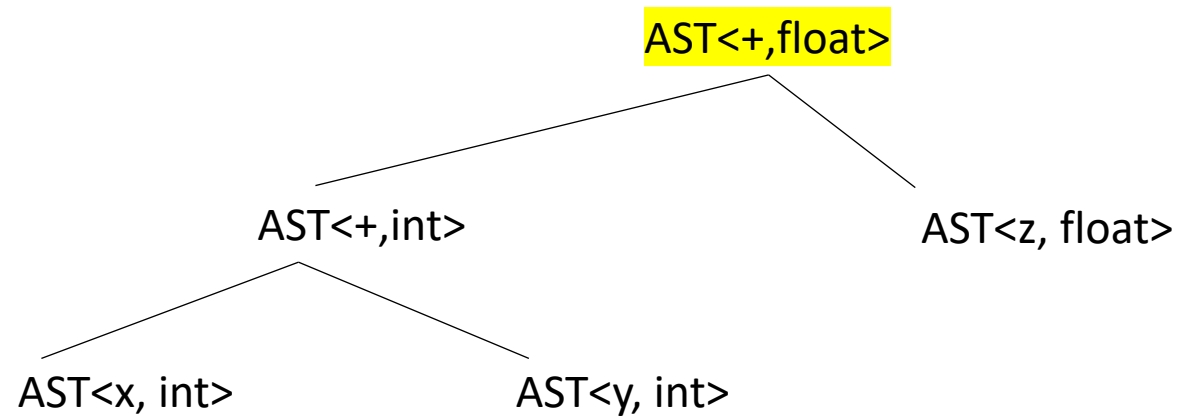
# Type checking on an AST

```
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

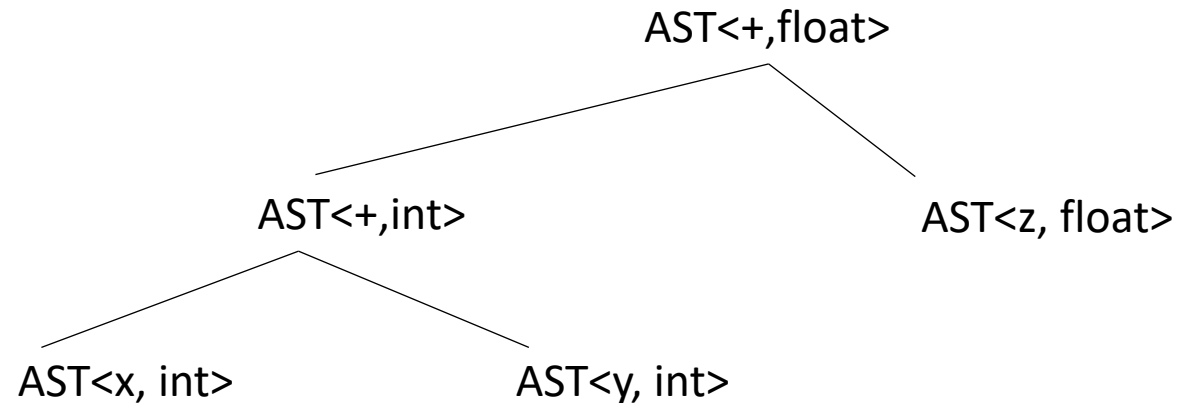
```
int x;  
int y;  
float z;  
float w;  
w = x + y + z
```

*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

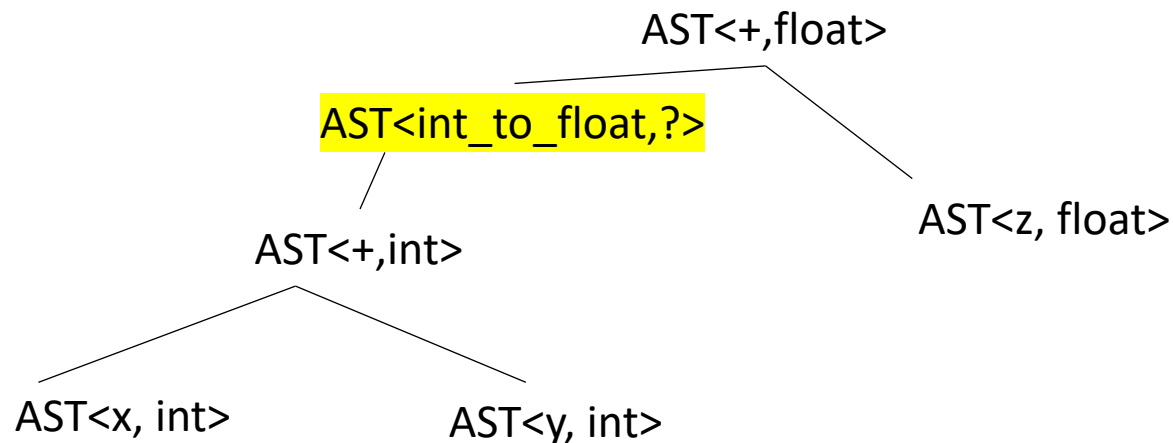
```
int x;  
int y;  
float z;  
float w;  
w = x + y + z
```

*How do we get the type for this one?*

*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



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

```
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)
```

```
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
```

*Now we need to set the types for the leaf nodes*

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
```

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)
```

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[1]
    eat(ID)
    ST.insert(id_name, None)
    eat(SEMI)
}
```

*in homework 2 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...

# add the type at parse time

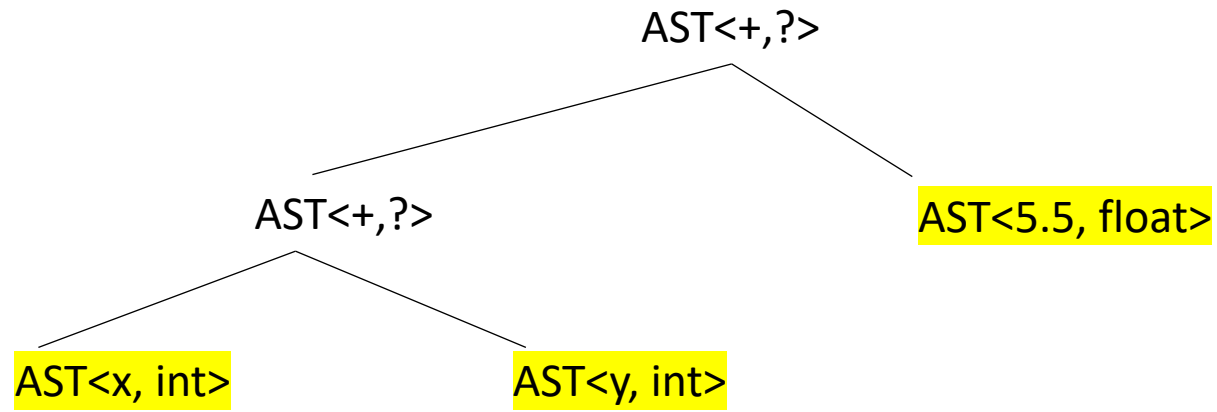
Unit ::= ID
NUM

```
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
```

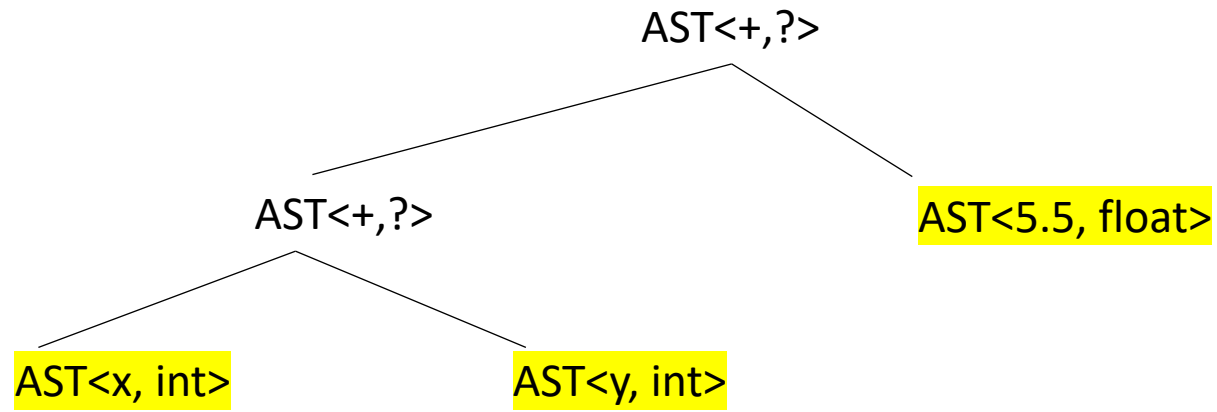


# Type inference

- 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):  
    case split on n:  
  
    if n is a leaf node:  
        return n.get_type()
```

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

*base case*



# 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:
            ...
```

# 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:      *lookup the rule for plus*  
        return lookup type from table

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:                      *lookup the rule for plus*  
        return lookup type from table

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()
```

*we need to make sure the  
children have types!*

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

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()
```

*we should record our type*

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

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()

is this just for plus?

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

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()
```

is this just for plus?

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

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

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()
```

is this just for plus?

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

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

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 inference

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

```
    case split on n:
```

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

```
        if n is an assignment:
            ....
```

```
        if n is a bin op node:
            ...
```

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()
```

*we should record our 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
```

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()
```

*we should record our 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
```

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()
```

*we should record our 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
```

inference rules for plus

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

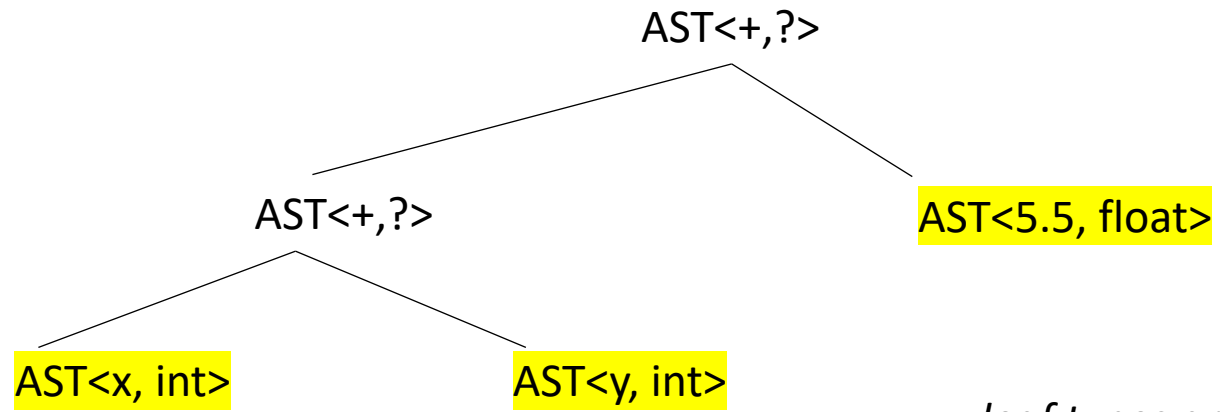
*like in Python*

Example



# Type inference

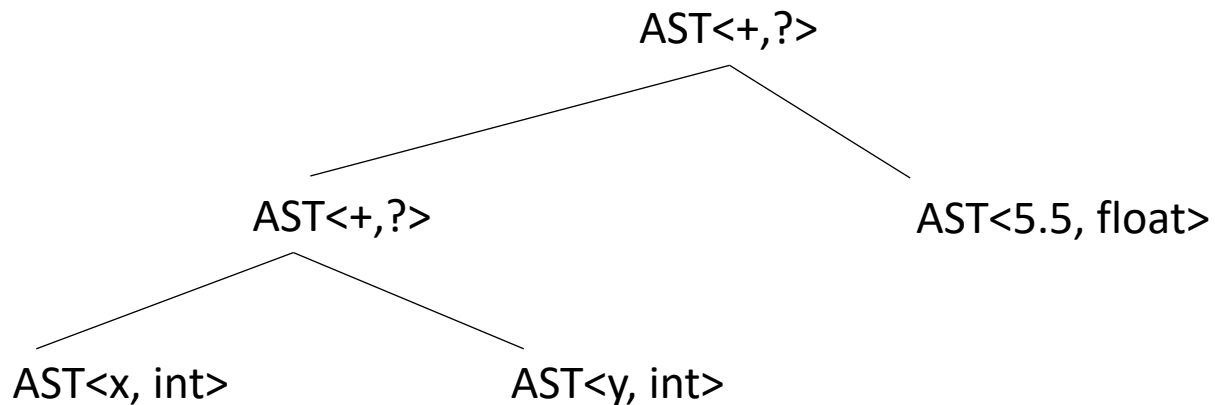
```
int x;  
int y;  
float w;  
w = x + y + 5.5
```



*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:
```

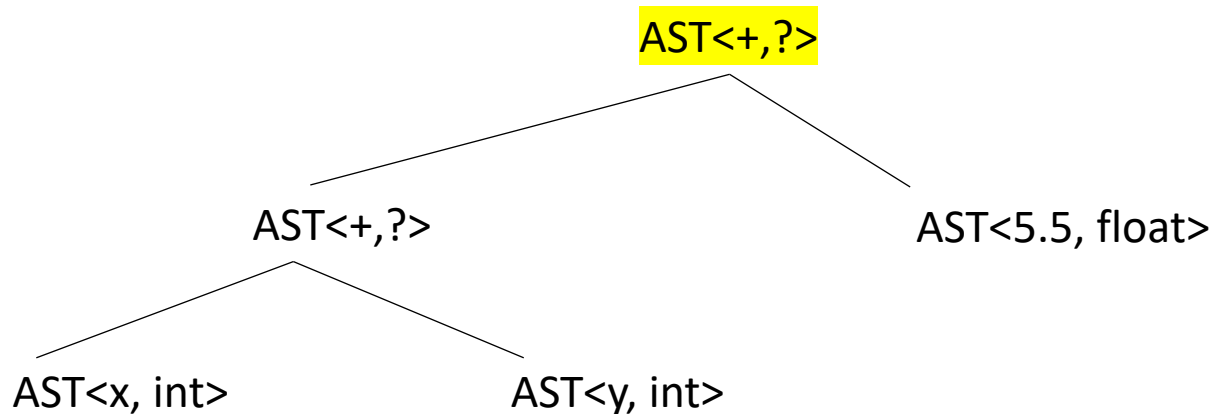
```
        if n is a leaf node:  
            return n.get_type()
```

```
        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):
```

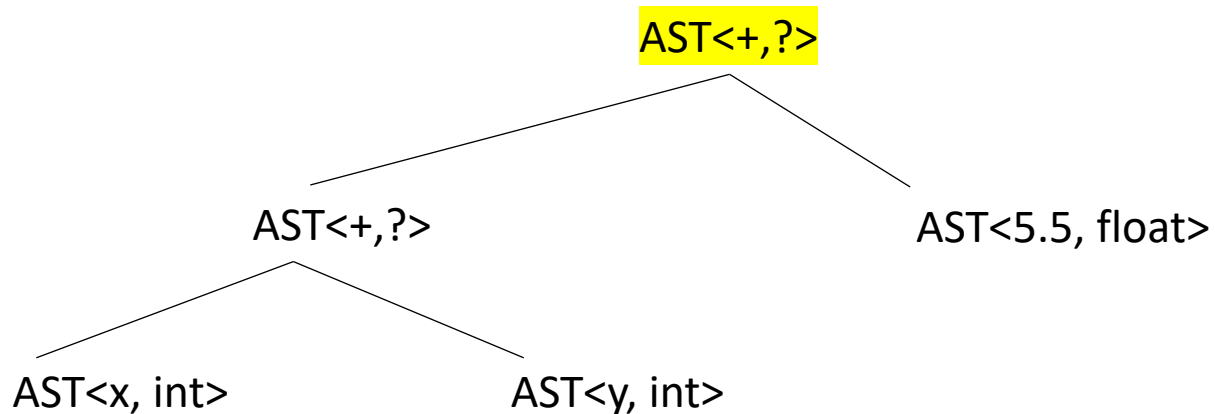
```
    case split on type of n:
```

```
        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()
```

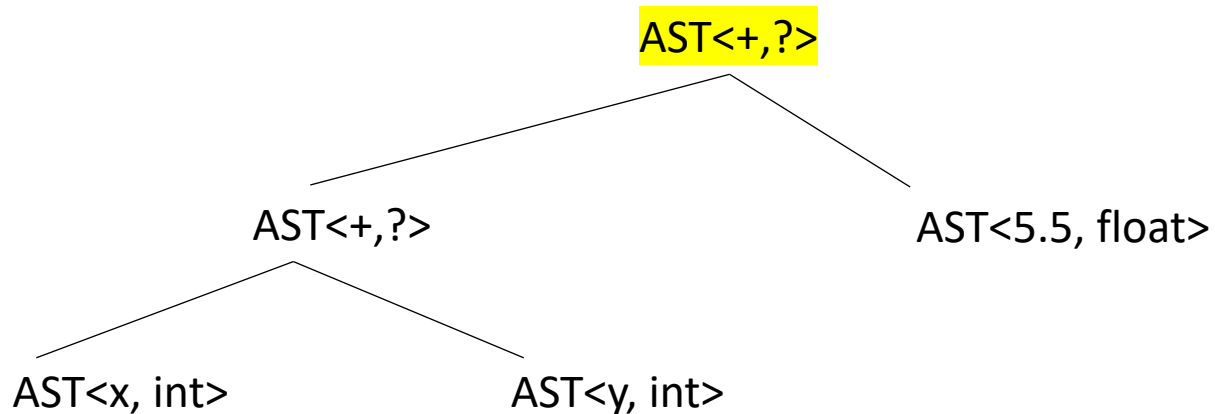
```
        if n is a bin op node:
```

```
            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
```

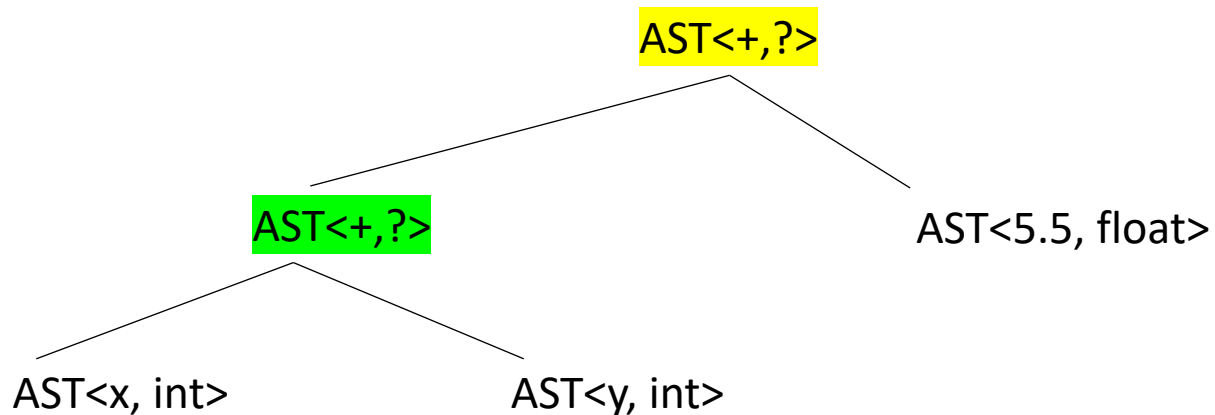
```
            t = lookup type from table
```

```
            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:
```

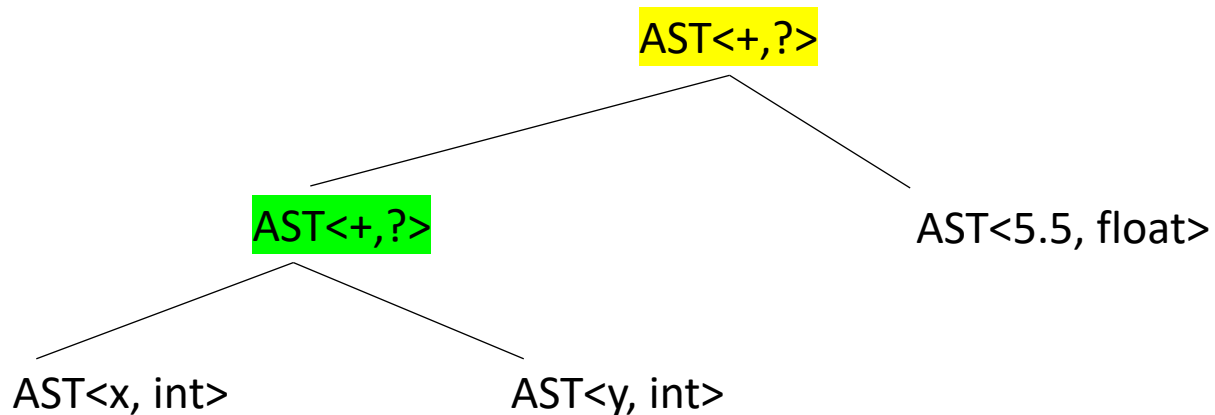
```
        if n is a leaf node:  
            return n.get_type()
```

```
        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()
```

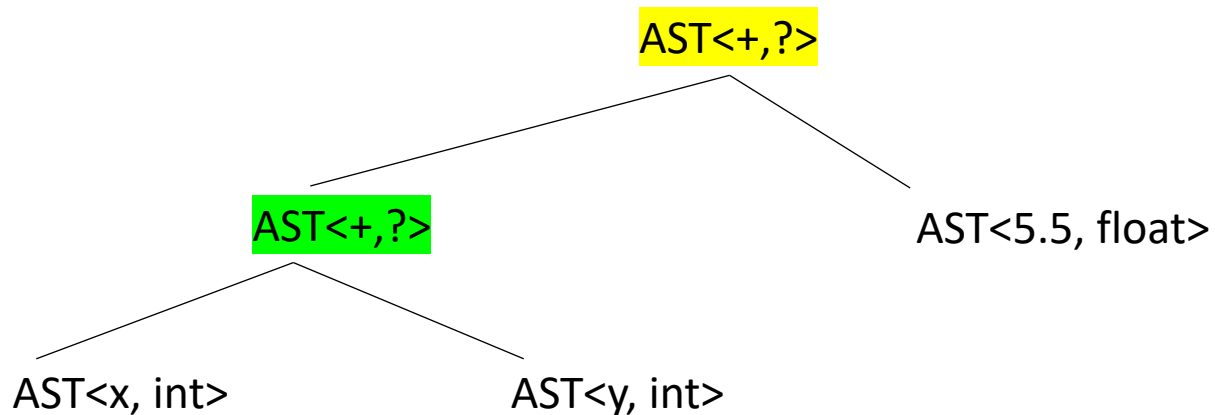
```
        if n is a bin op node:
```

```
            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
```

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
```

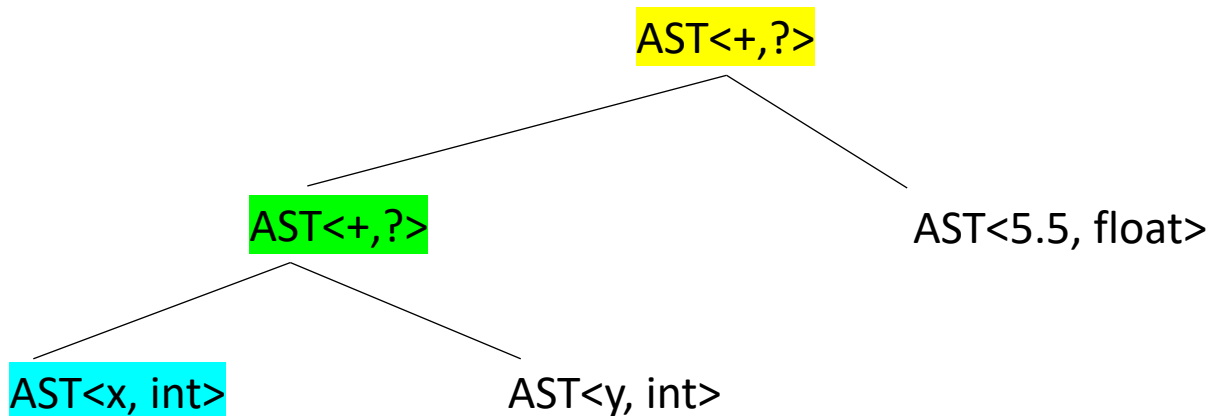
```
            set n type to t
```

```
            return t
```



# Type inference

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



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

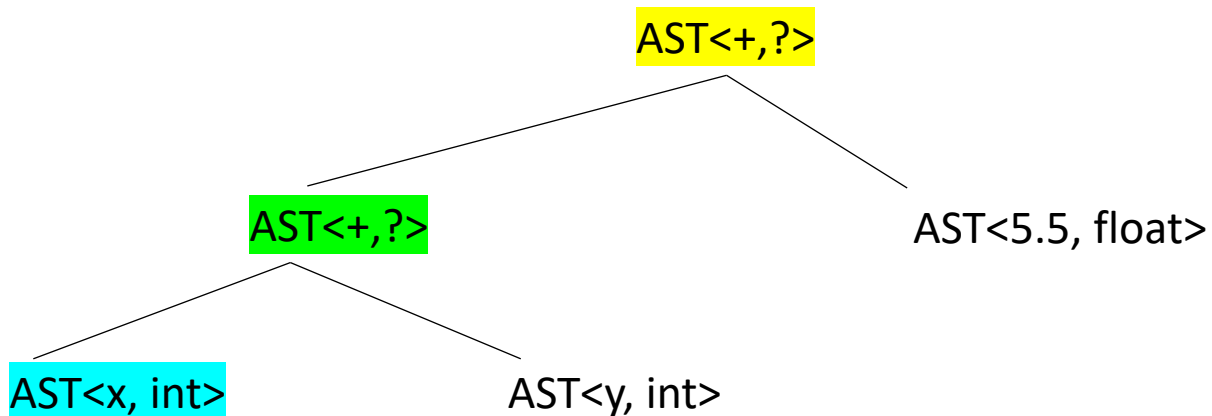
```
    case split on type of n:
```

```
        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
```

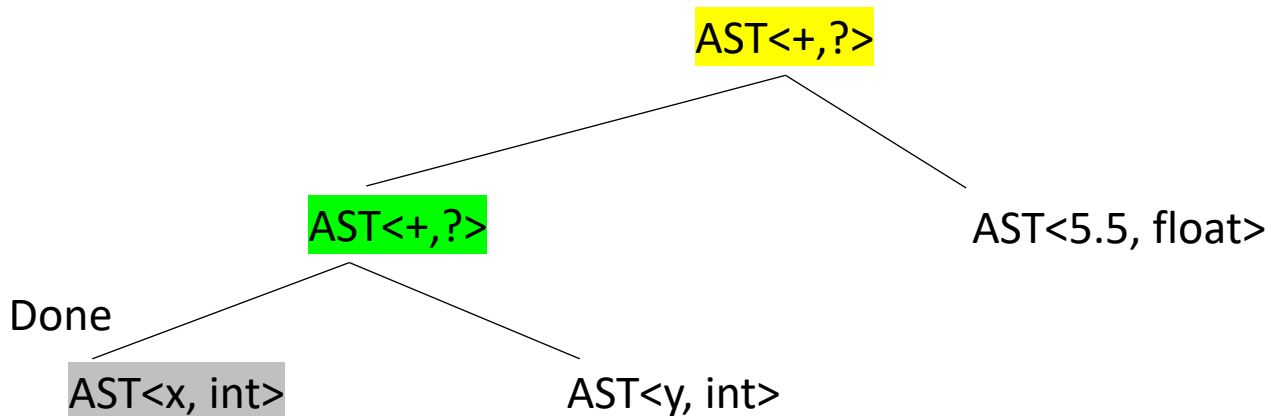
```
            t = lookup type from table
```

```
            set n type to t
```

```
            return t
```

# Type inference

```
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()
```

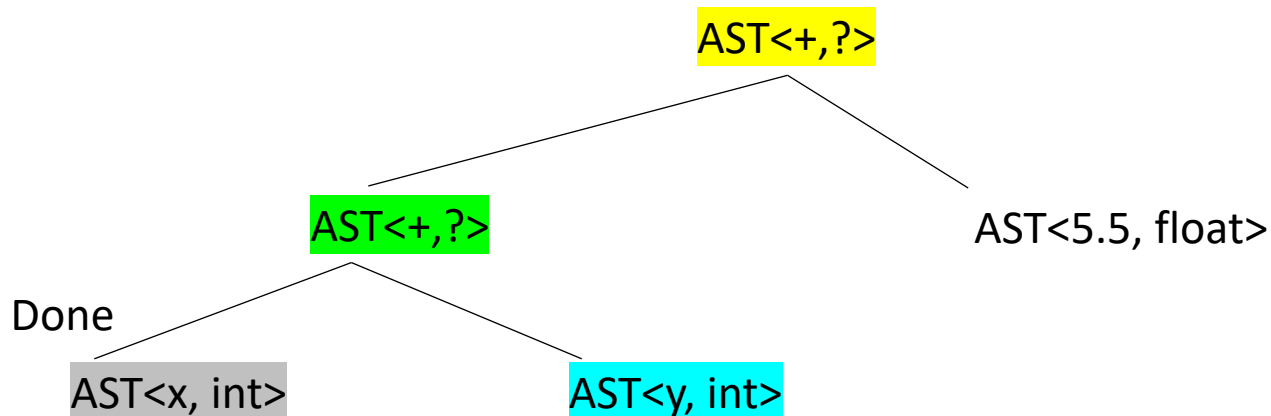
```
        if n is a bin op node:
```

```
            do type inference on children
```

```
            t = lookup type from table
```

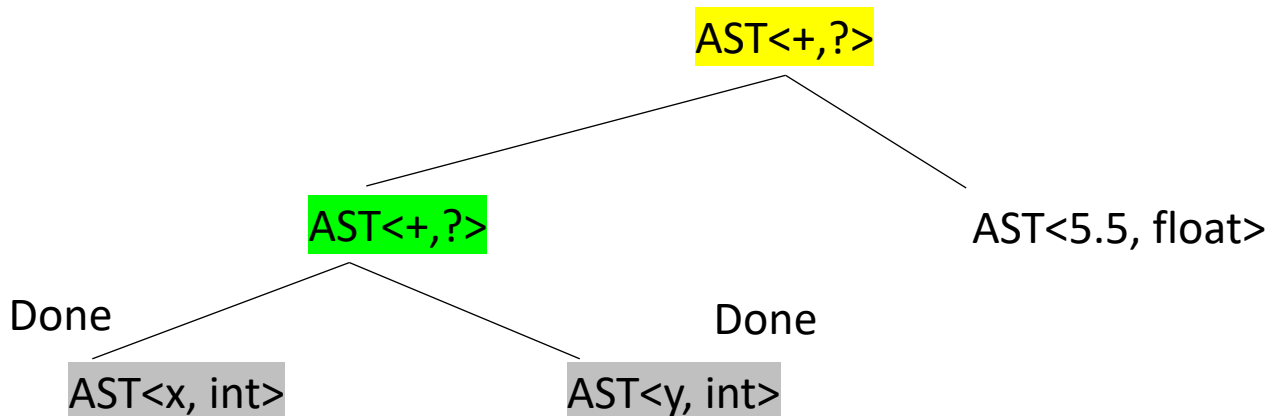
```
            set n type to t
```

```
            return t
```



# Type inference

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



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

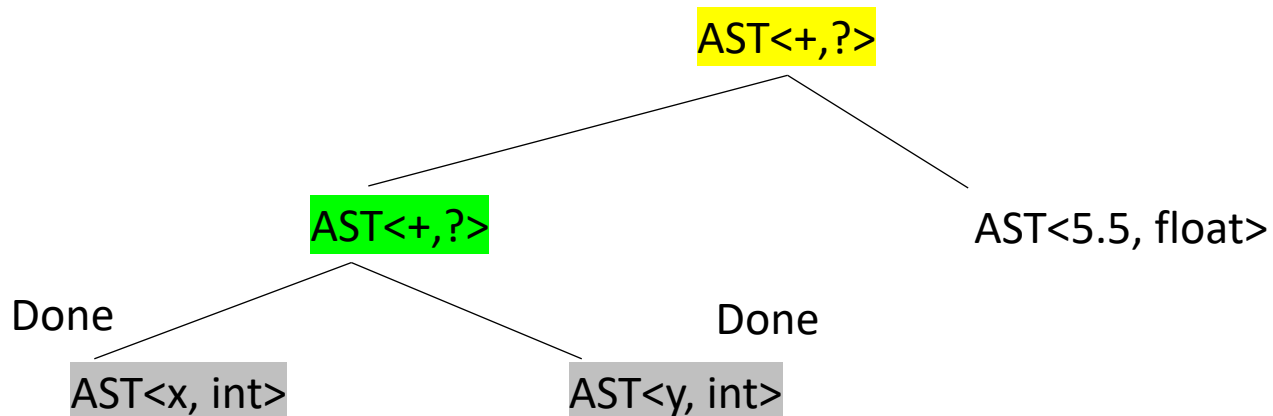
```
    case split on type of n:
```

```
        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()
```

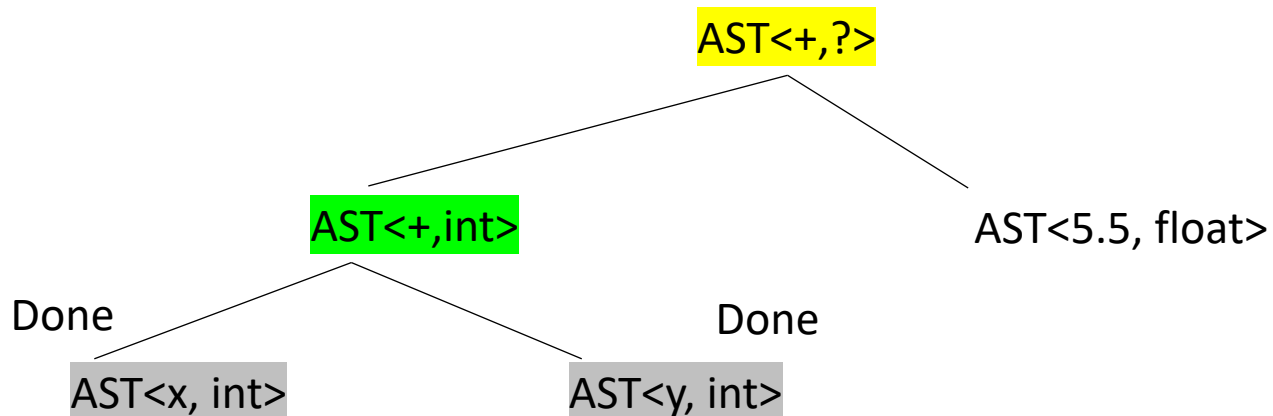
```
        if n is a bin op node:  
            do type inference on children  
            t = lookup type from table  
            set n type to t  
            return t
```

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()
```

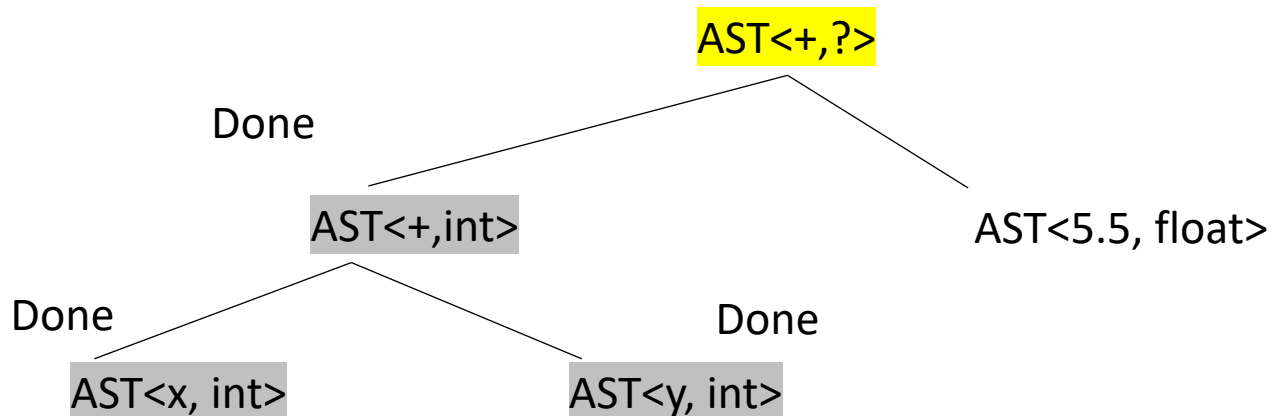
```
        if n is a bin op node:  
            do type inference on children  
            t = lookup type from table  
            set n type to t  
            return t
```

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
```

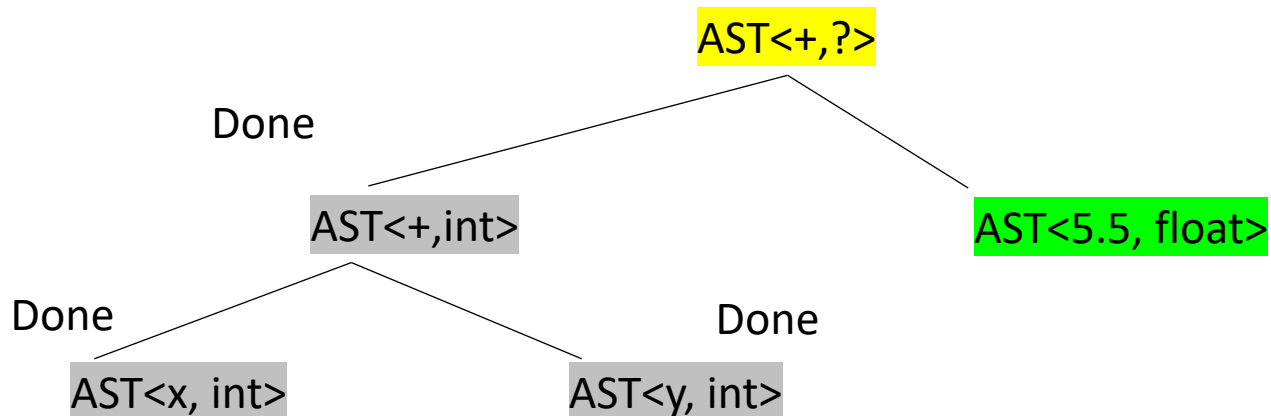
Table for **most** binary ops

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
```

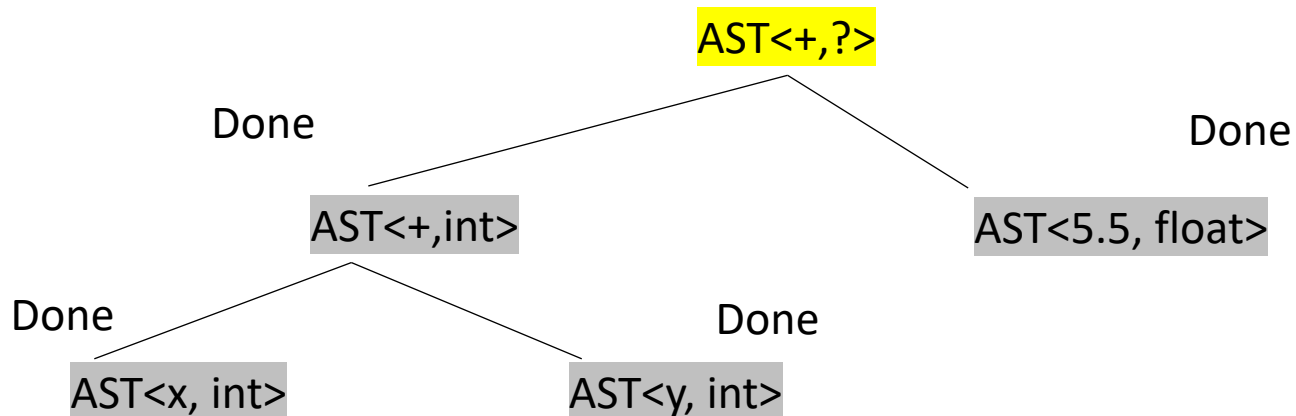
```
    return t
```

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
```

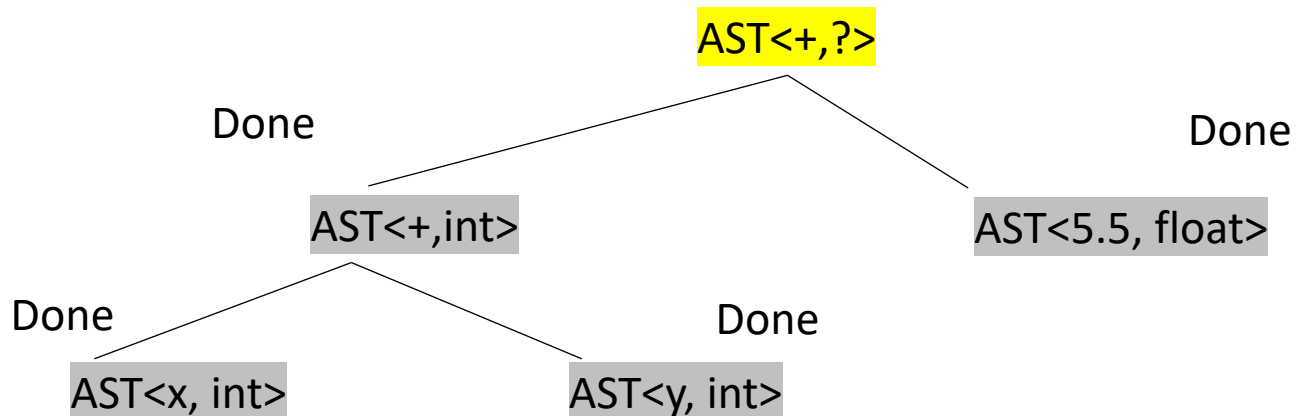
```
            return t
```

Table for **most** binary ops

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

# Type inference

```
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()
```

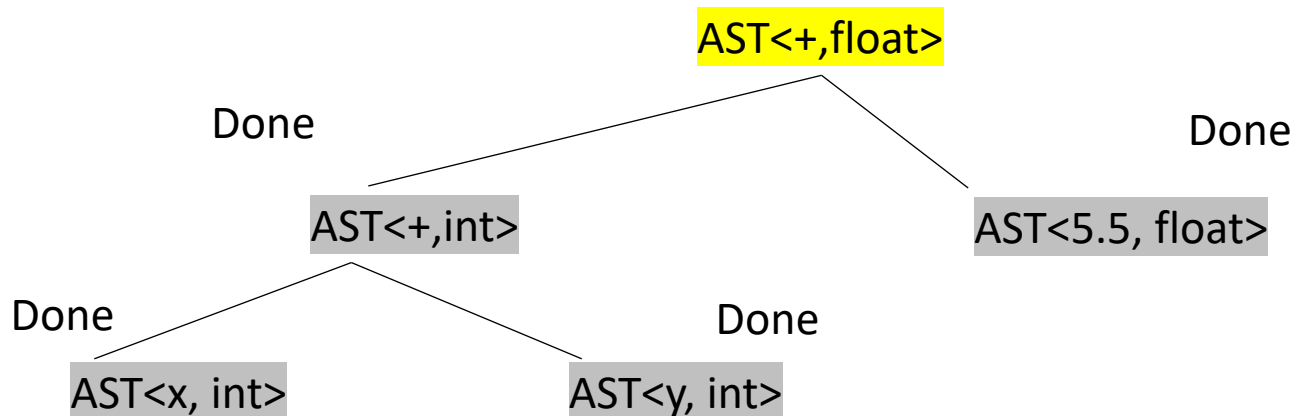
```
        if n is a bin op node:  
            do type inference on children  
            t = lookup type from table  
            set n type to t  
        return t
```

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()
```

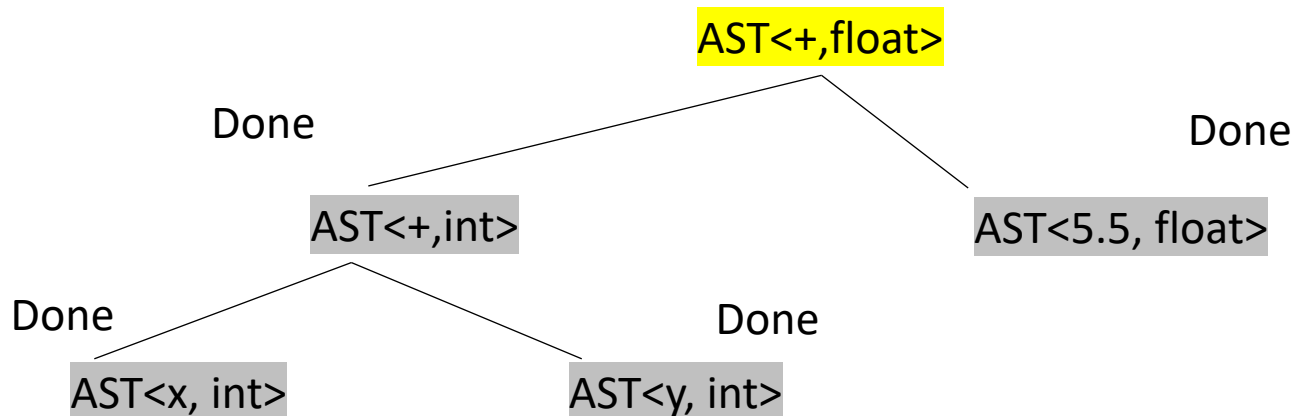
```
        if n is a bin op node:  
            do type inference on children  
            t = lookup type from table  
            set n type to t  
            return t
```

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:
```

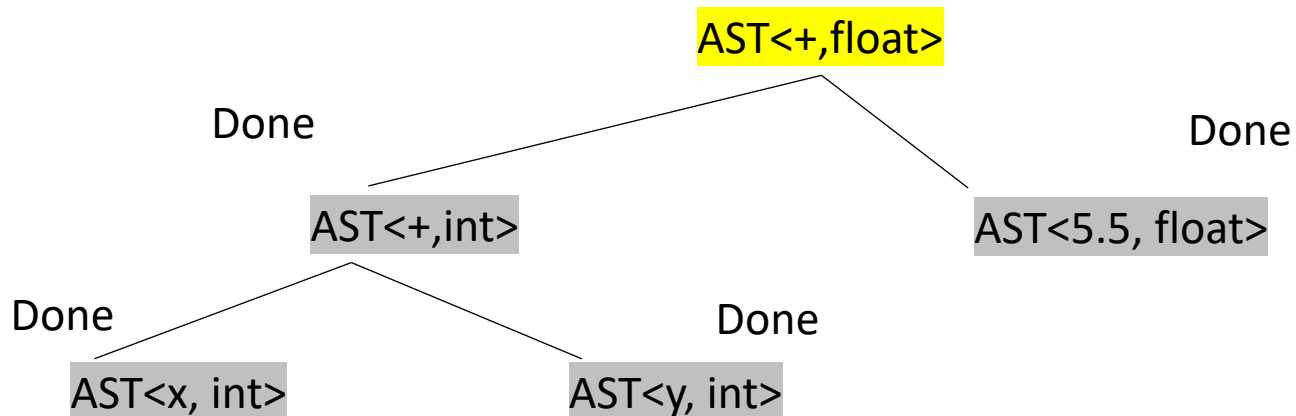
```
        if n is a leaf node:  
            return n.get_type()
```

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

*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):
```

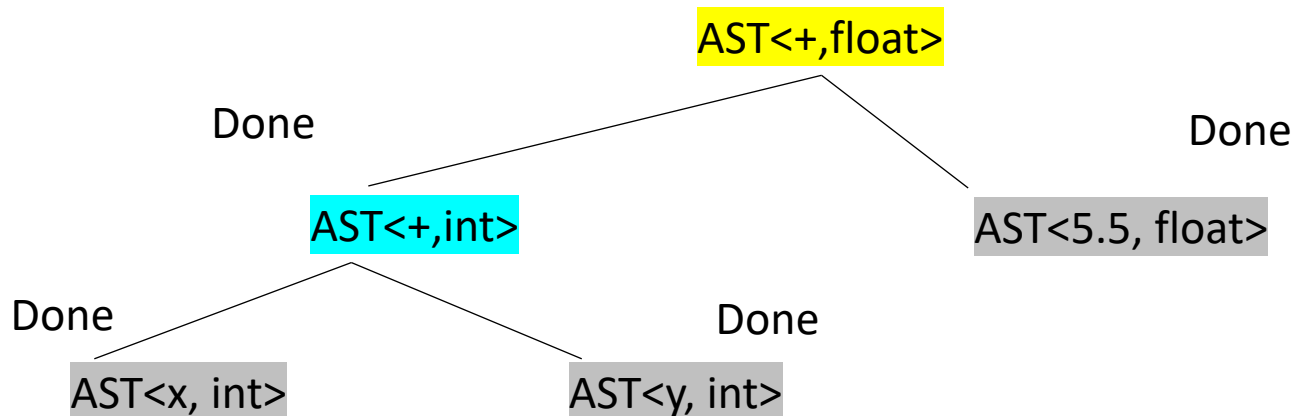
*this will need to be done for both children*

```
    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
```



## 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)
```



```
from enum import Enum

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

what types are these nodes?

## 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)
```

```
from enum import Enum

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

what types are these nodes?

## New type of AST nodes: unary operators

```
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)
```

```
from enum import Enum

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

what types are these nodes?

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

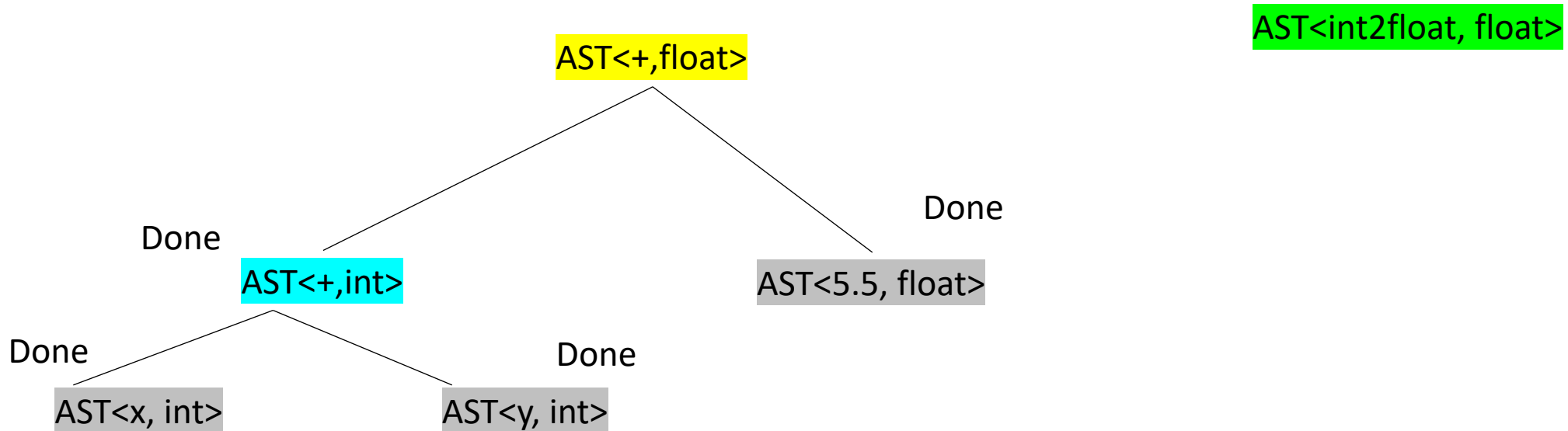
## New type of AST nodes: unary operators

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

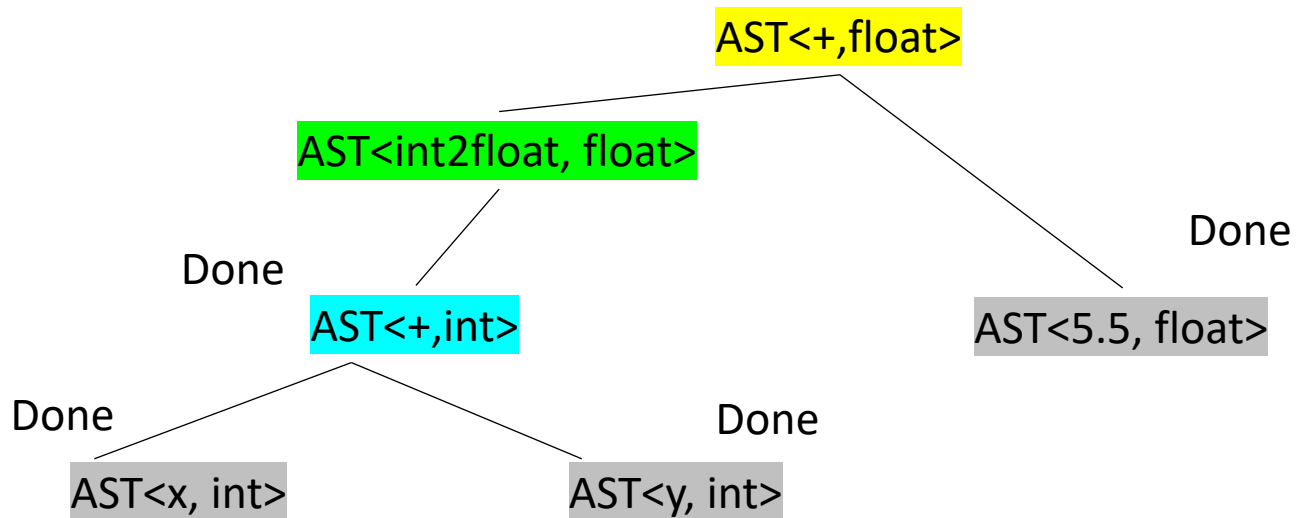
class ASTIntToFloatNode(ASTBinUnNode):
    def __init__(self, child):
        self.set_type(Types.FLOAT)
        assert(child.get_type() == Types.INT)
        super().__init__(child)

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
```

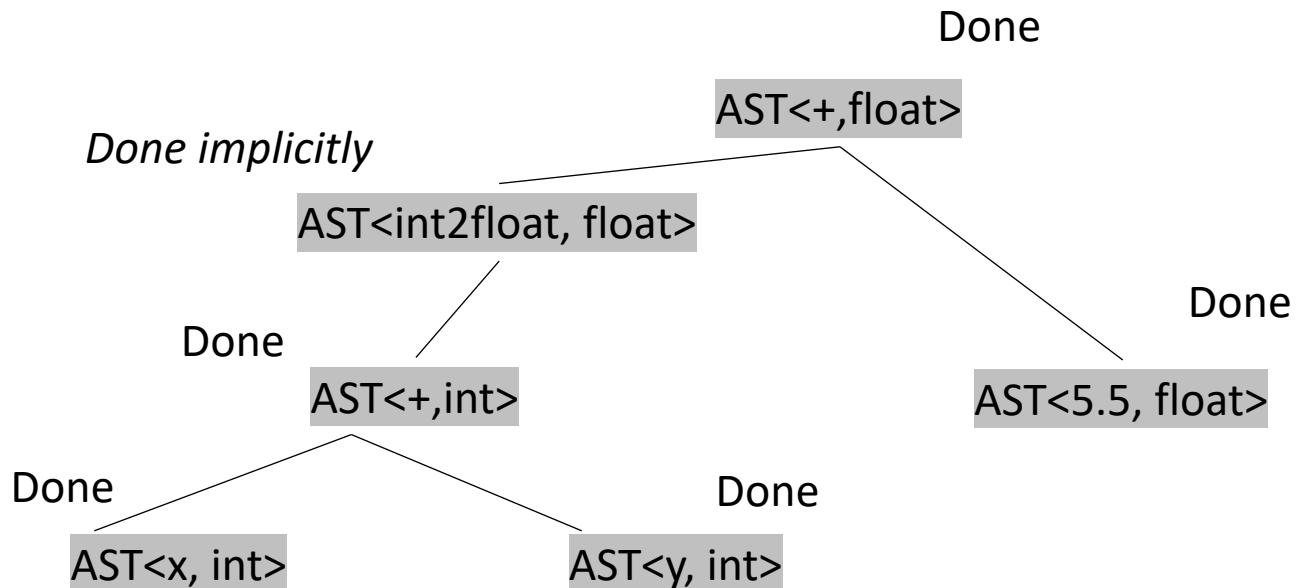


```
def type_conversion(n):  
    if n.left_child type is NOT the same as n type:  
        conv = get conversion AST node  
        conv.child = left_child  
        set n.left_child to = conv
```



# 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***

# See everyone on Monday!

- We will discuss linearizing code