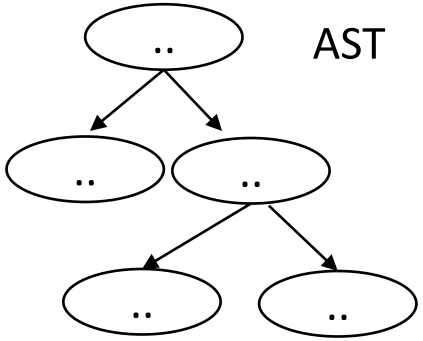
CSE110A: Compilers

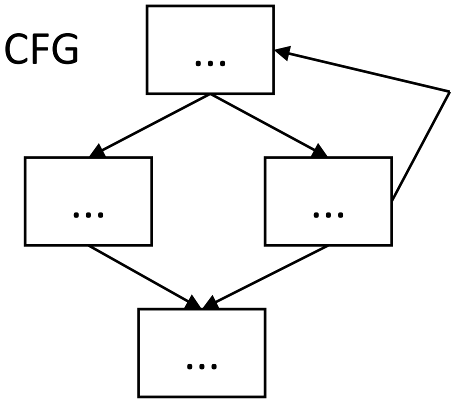
AST



Topics:

- *Module 3: Intermediate representations*
  - *ASTs*
  - *Type checking*

CFG



3 address code

```
store i32 0, ptr %2
%3 = load i32, ptr %1
%4 = add nsw i32 %3, 1,
store i32 %4, ptr %1
%5 = load i32, ptr %2
```

ASTs: Abstract Syntax Trees

```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)
```

Creating an AST from a parser

```
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)
```

Creating an AST from production rules

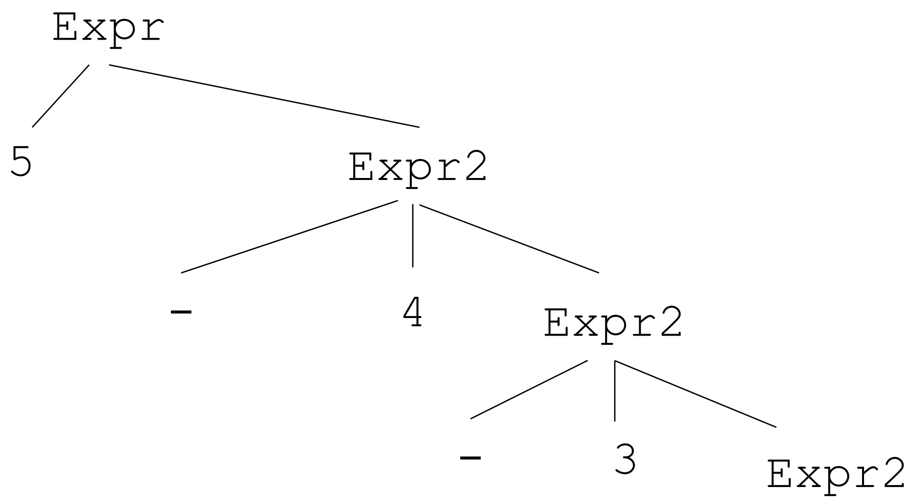
Operator	Name	Productions	Production action
+	`expr`	`: expr PLUS term` `| term`	`{return ASTAddNode($1,$3)}` `{return $1}`
*	`term`	`: term TIMES factor` `| factor`	`{return ASTMultNode($1,$3)}` `{return $1}`

()	`factor`	`: LPAR expr RPAR`
		`| NUM`
		`| ID`
			`{return $2}`
			`{return ASTNumNode($1)}`
			`{return ASTIDNode($1)}`

Creating an AST from top down grammar

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

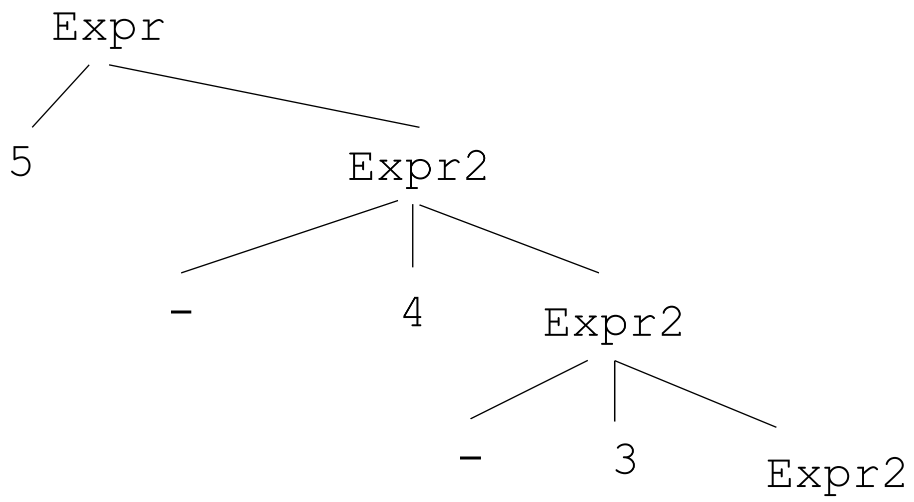
5 - 4 - 3

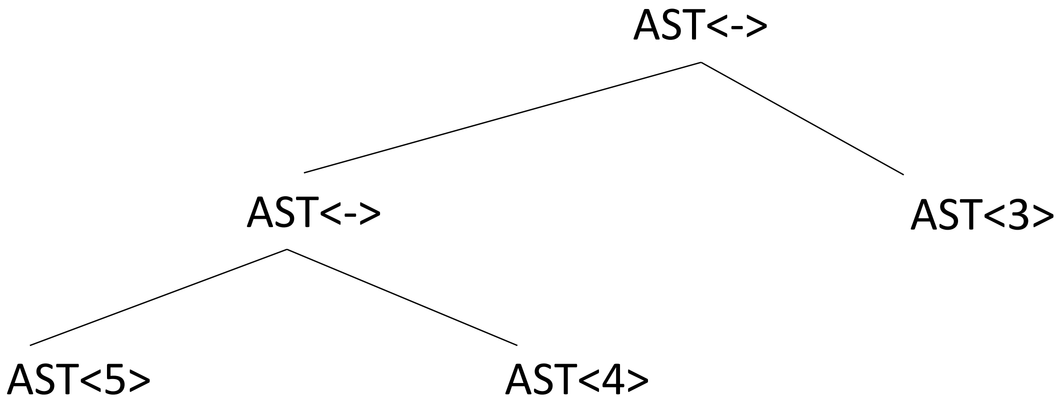


Creating an AST from top down grammar

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

`5 - 4 - 3`



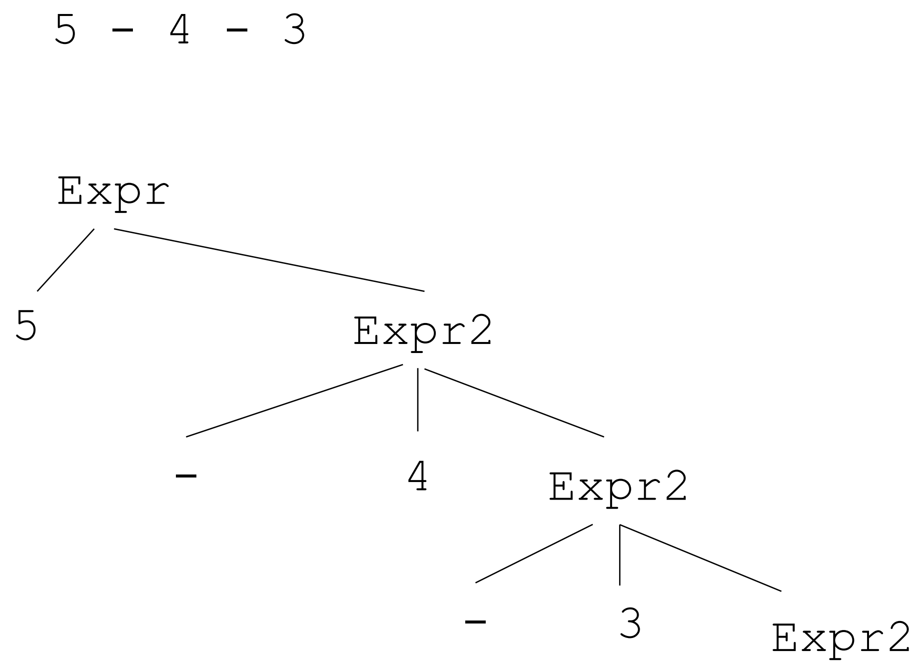


How do we get to the desired parse tree?

Creating an AST from top down grammar

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

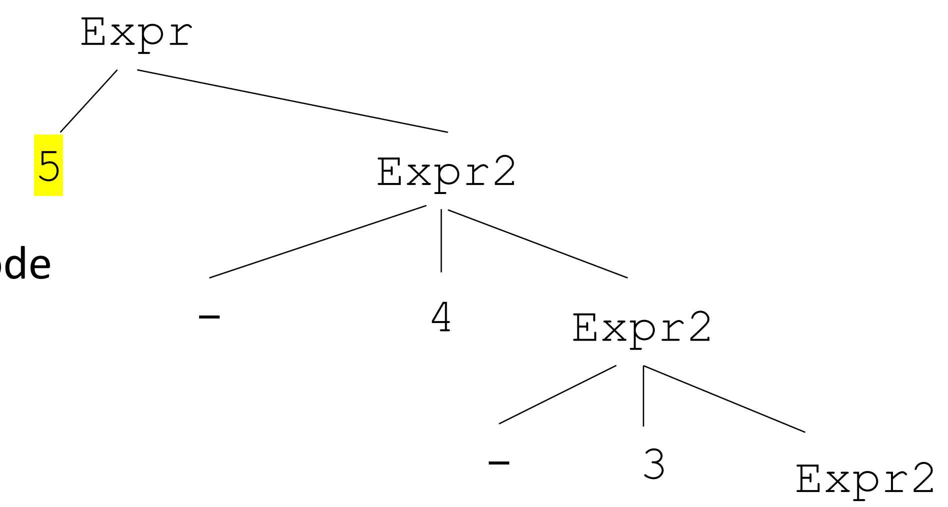
Keep in mind that because we wrote our own parser, we can inject code at any point during the parse.



Creating an AST from top down grammar

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

5 - 4 - 3



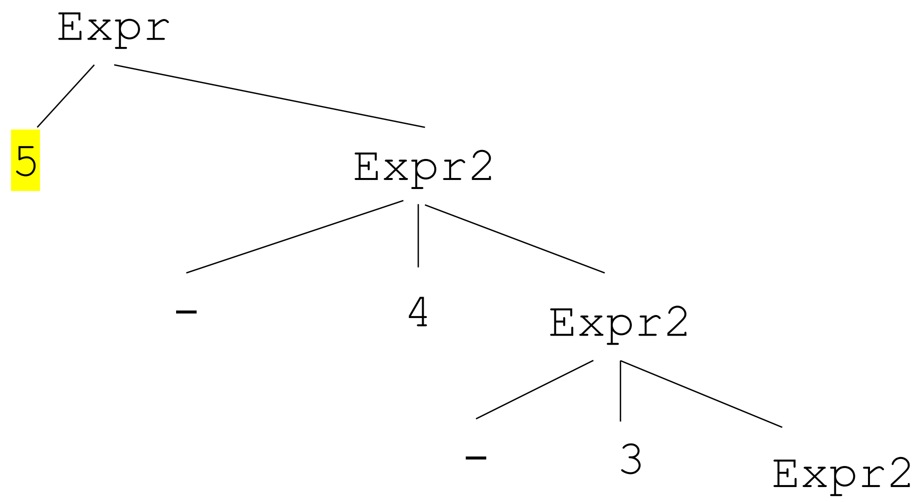
Get number node after we see a number

AST<5>

Creating an AST from top down grammar

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

5 - 4 - 3



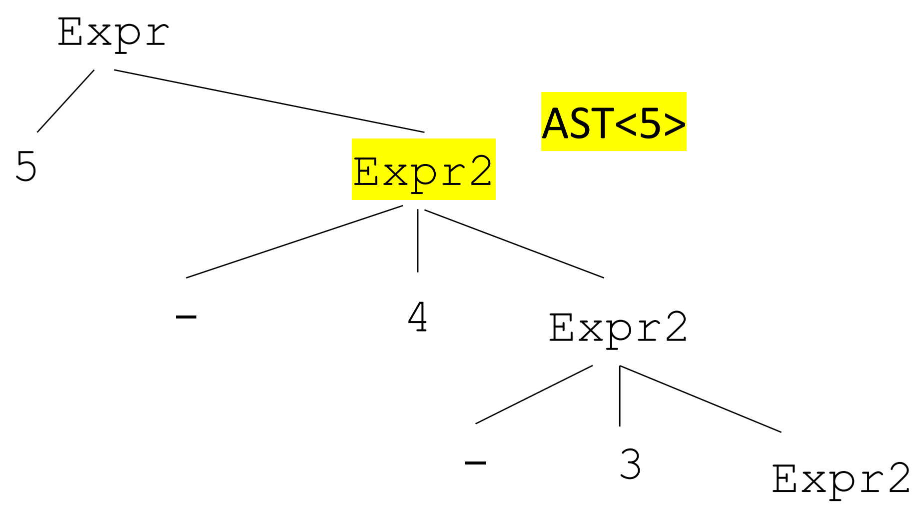
Pass the node down

AST<5>

Creating an AST from top down grammar

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

`5 - 4 - 3`



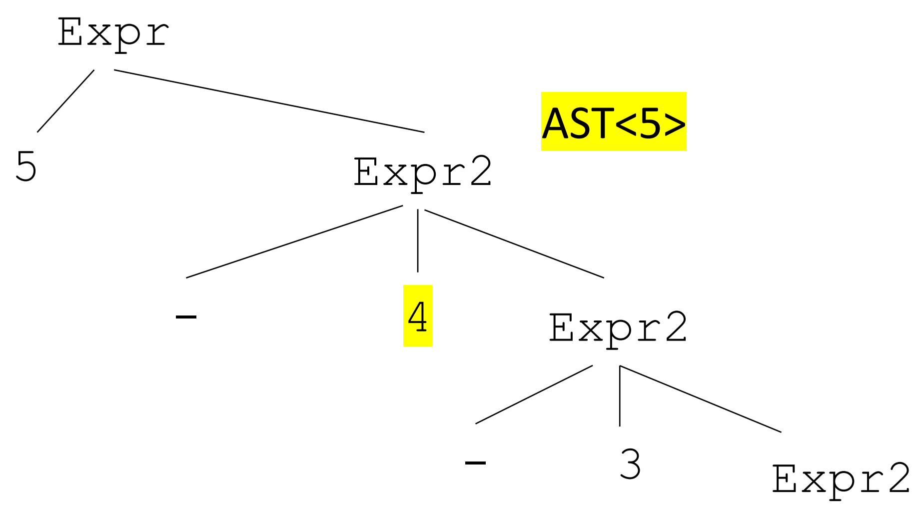
Pass the node down

AST<5>

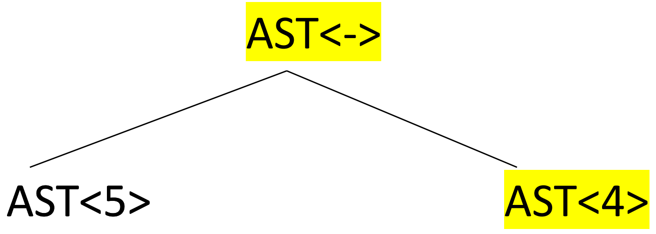
Creating an AST from top down grammar

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

5 - 4 - 3



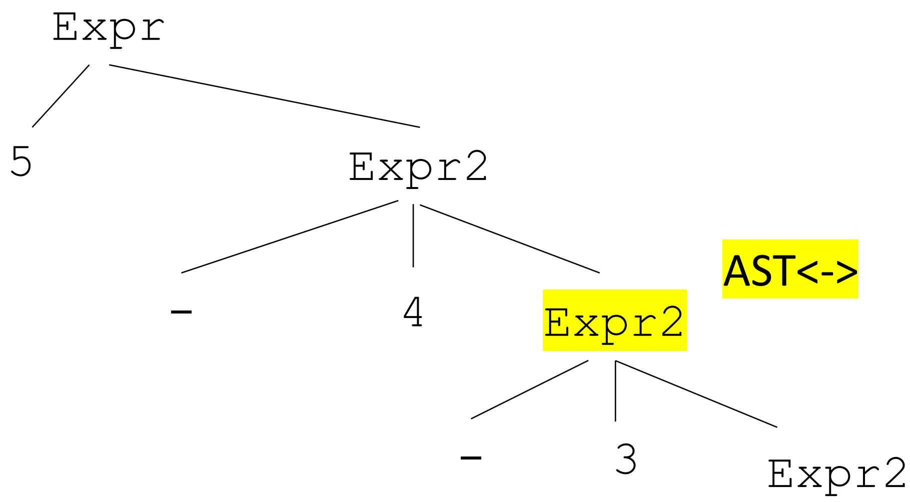
In Expr2, after 4 is parsed, create a number node and a minus node



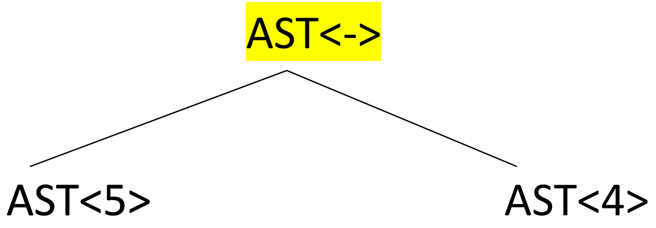
Creating an AST from top down grammar

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

5 - 4 - 3

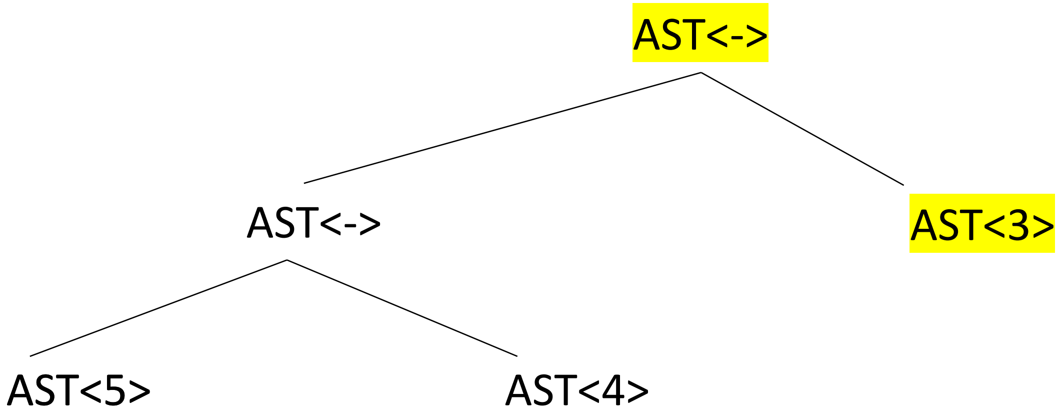


pass the new node down

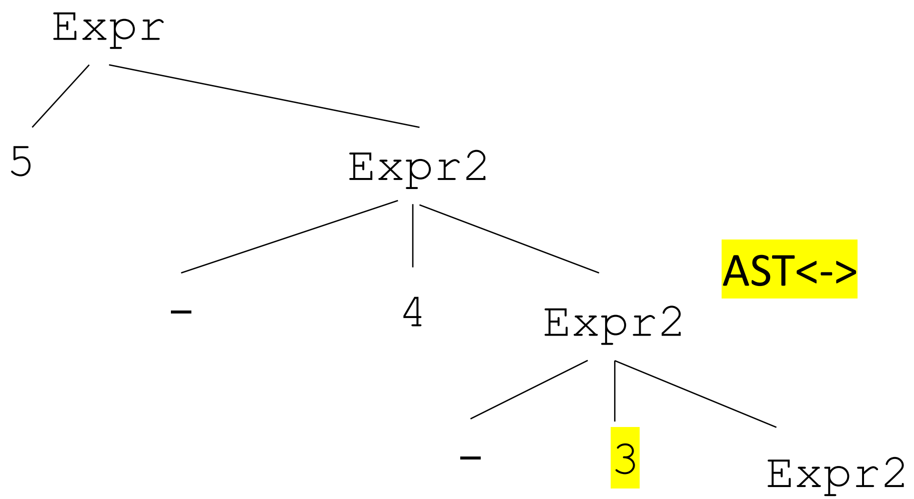


Creating an AST from top down grammar

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



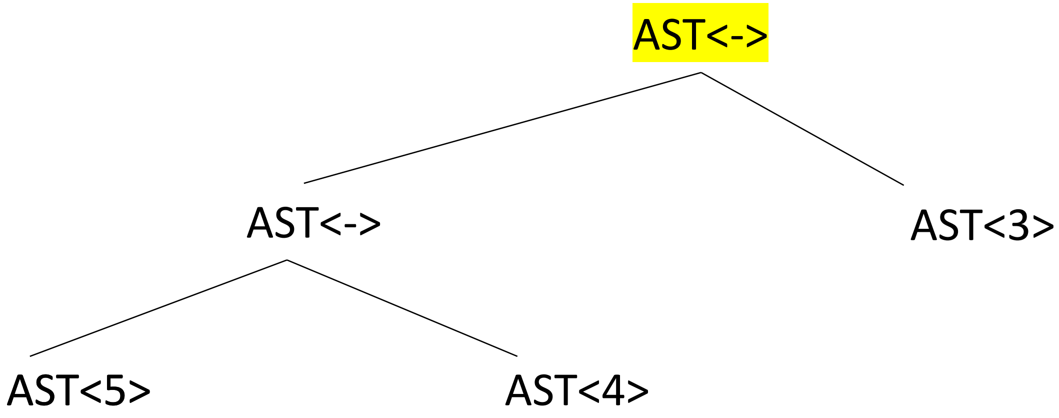
5 - 4 - 3



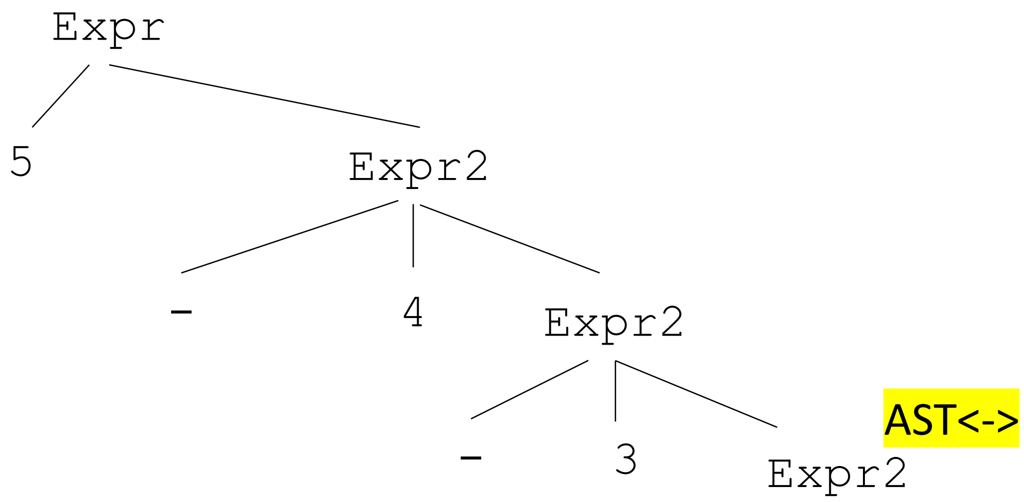
In Expr2, after 3 is parsed, create a number node and a minus node

Creating an AST from top down grammar

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



`5 - 4 - 3`

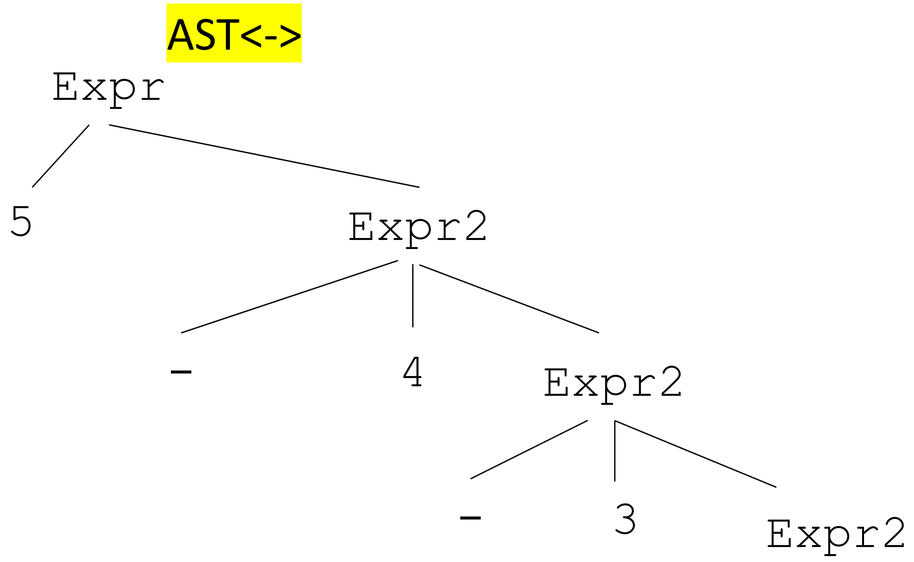


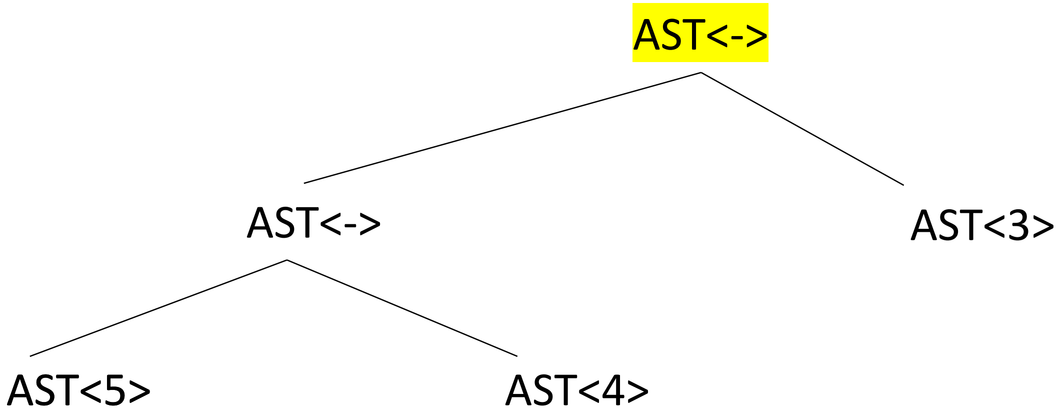
pass down the new node

Creating an AST from top down grammar

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

5 - 4 - 3





return the node when there is nothing left to parse

Creating an AST from top down grammar

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

```
def parse_expr(self):
   #get the value from the lexeme
   value = self.to_match.value       # the next terminal, a NUM
   node = ASTNumNode(value)
   self.eat("NUM")
   return self.parse_expr2(node)
```

Creating an AST from top down grammar

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

```
def parse_expr(self):
    #get the value from the lexeme
    value = self.to_match.value
    node = ASTNumNode(value)
    self.eat("NUM")
    return self.parse_expr2(node)
```

```
def parse_expr2(self, lhs_node):
    # ... for applying the first production rule
    self.eat("MINUS")
    value = self.to_match.value                  # the value of next terminal a NUM
    rhs_node = ASTNumNode(value)                 # build that node
    self.eat("NUM")
    node = ASTMinusNode(lhs_node, rhs_node)     # build the Expr2 dyadic node
    return self.parse_expr2(node)               # return the dyadic node
```

Creating an AST from top down grammar

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

```
def parse_expr(self):
    #get the value from the lexeme
    value = self.to_match.value
    node = ASTNumNode(value)
    self.eat("NUM")
    return self.parse_expr2(node)
```

```
def parse_expr2(self, lhs_node):
    # ... for applying the second production rule of Expr2, when there is no MINUS
    return lhs_node
```

Creating an AST from top down grammar

```
Expr   ::= Term Expr2
Expr2  ::= MINUS Term Expr2
         |    ""
```

In a more realistic grammar, you might have more layers: e.g. a Term

how to adapt?

```
def parse_expr(self):
    #get the value from the lexeme
    value = self.to_match.value
    node = ASTNumNode(value)
    self.eat("NUM")
    return self.parse_expr2(node)
```

```
def parse_expr2(self, lhs_node):
    # ... for applying the first production rule
    self.eat("MINUS")
    value = self.to_match.value
    rhs_node = ASTNumNode(value)
    self.eat("NUM")
    node = ASTMinusNode(lhs_node, rhs_node)
    return self.parse_expr2(node)
```

Creating an AST from top down grammar

```
Expr   ::= Term Expr2
Expr2  ::= MINUS Term Expr2
         |    ""
```

```
def parse_expr(self):
    node = self.parse_term()
    return self.parse_expr2(node)
```

In a more realistic grammar, you might have more layers: e.g. a Term

how to adapt?

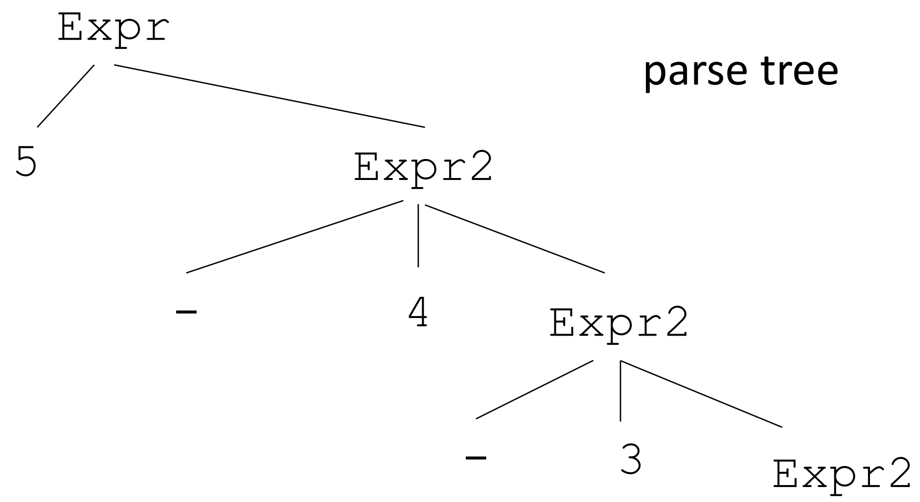
```
def parse_expr2(self, lhs_node):
    # ... for applying the first production rule
    self.eat("MINUS")
    rhs_node = self.parse_term()
    node = ASTMinusNode(lhs_node, rhs_node)
    return self.parse_expr2(node)
```

The parse_term will figure out how to get you an AST node for that term.

Evaluate an AST by doing a post order traversal

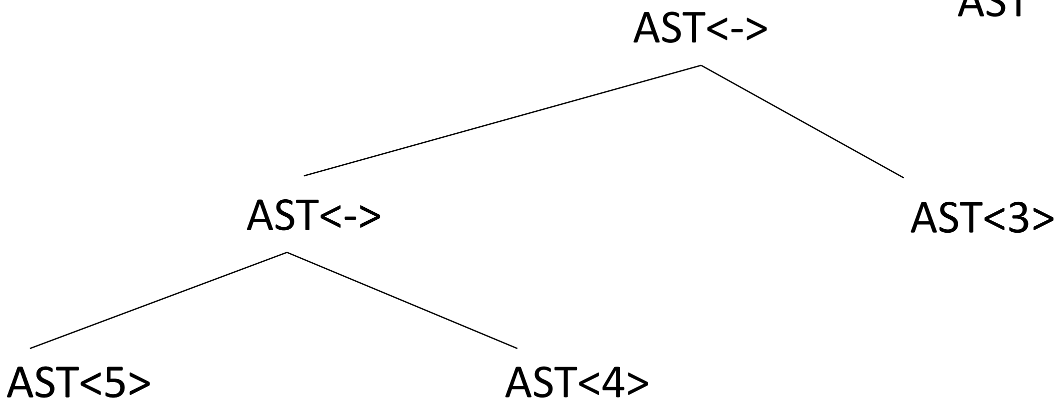
```
Expr  ::= NUM Expr2
Expr2 ::= MINUS NUM Expr2
       |   “”
```

`5 - 4 - 3`



parse tree

AST

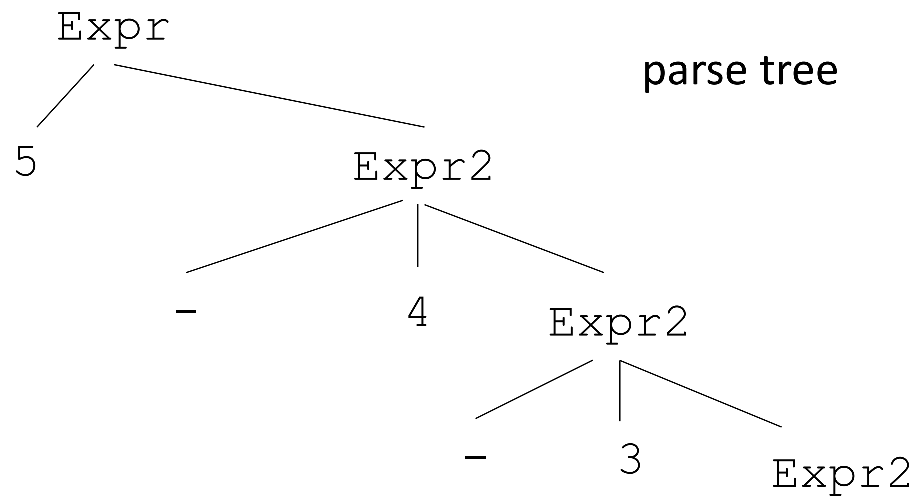


Parse trees cannot always be evaluated in post-order. An AST should always be

Strategy to Build AST with Top-Down Parsing

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

5 - 4 - 3



parse tree

Strategy: Build AST as You Return from Recursive Calls

- Each function corresponds to a grammar rule.
- When you parse a component like an expr, your return an ASTNode.
- You construct the node *after* parsing its subparts — so **you build bottom-up**, even though you're recursing top-down.

This means: You build AST nodes in a way that **resembles post-order**, even though your parser runs top-down.

Parse trees cannot always be evaluated in post-order. An AST should always be

Example

- Python AST

```python
import ast

print(ast.dump(ast.parse('5-4-2')))
```

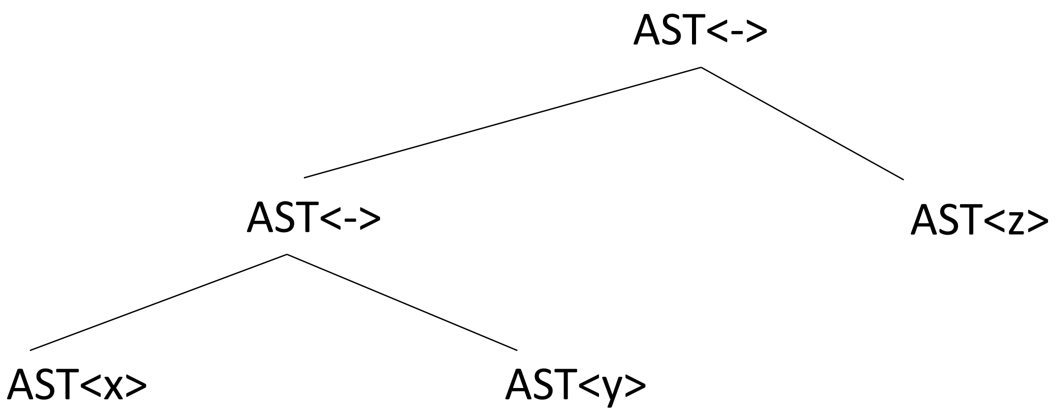
```
Expr(value=BinOp(left=BinOp(left=Num(n=5), op=Sub(), right=Num(n=4)), op=Sub(), right=Num(n=2)))
```

Evaluate an AST by doing a post order traversal

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

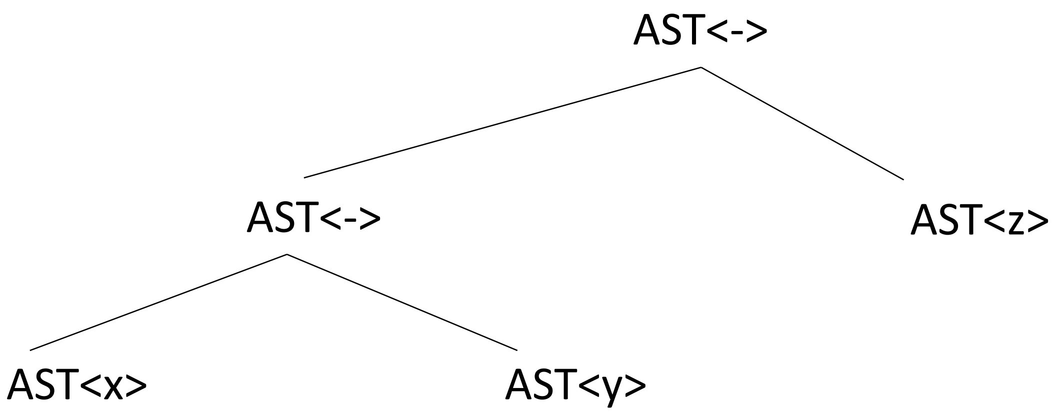
What if you cannot evaluate it?
What else might you do?

```
x - y - z
```



Evaluate an AST by doing a post order traversal

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



What if you cannot evaluate it as is?
The primitive types do not match.

What else might you do?

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

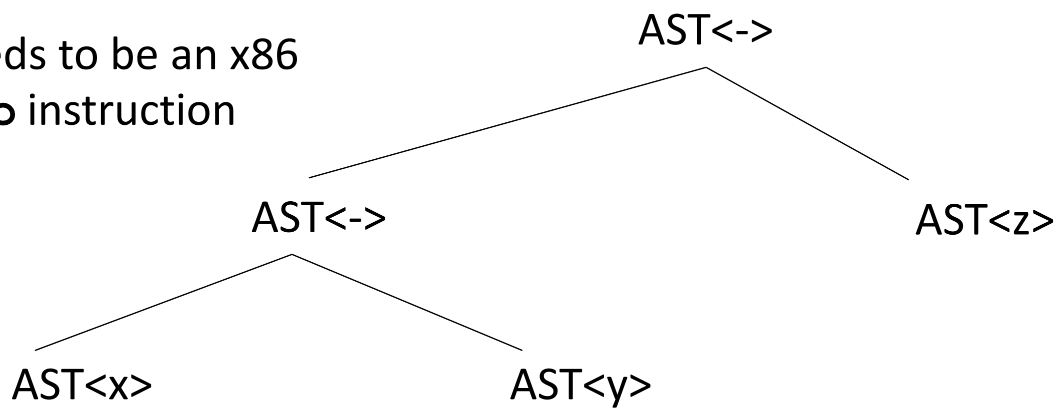
How does this change things?

Evaluate an AST by doing a post order traversal

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

needs to be an x86
`subss` instruction

needs to be an x86
`sub` instruction



Is this all?

What if you cannot evaluate it as is?
The primitive types do not match.

What else might you do?

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

How does this change things?

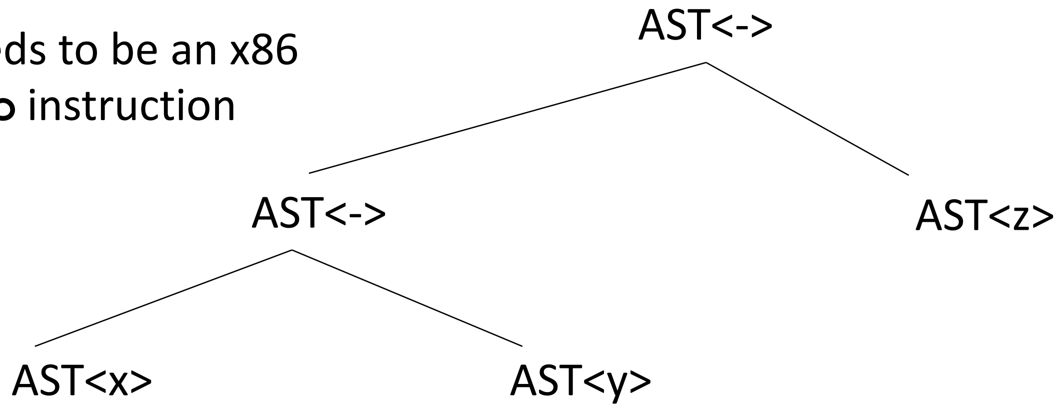
Evaluate an AST by doing a post order traversal

```
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



Let’s do some experiments.

What should 5 - 5.0 be?

Is this all?

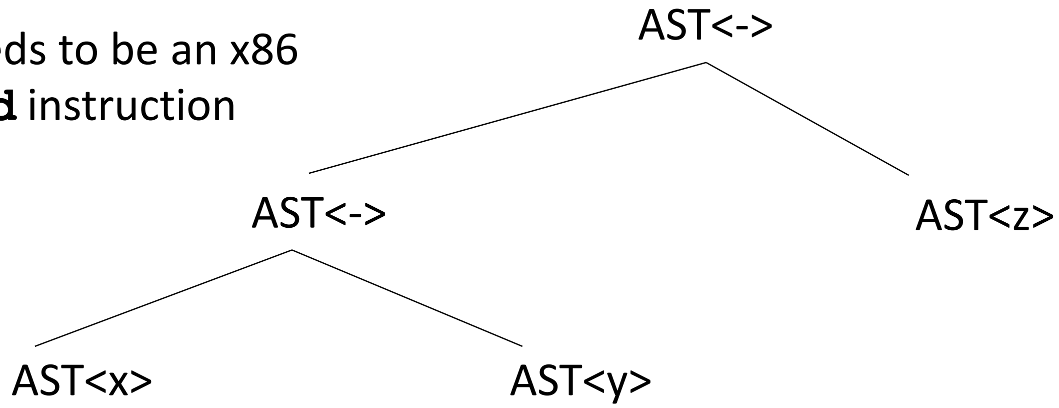
Evaluate an AST by doing a post order traversal

```
Expr  ::= NUM Expr2
Expr2 ::= MINUS 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



Is this all?

Lets do some experiments.

What should 5 - 5.0 be?

but

`addss r1 r2`

interprets both registers as floats

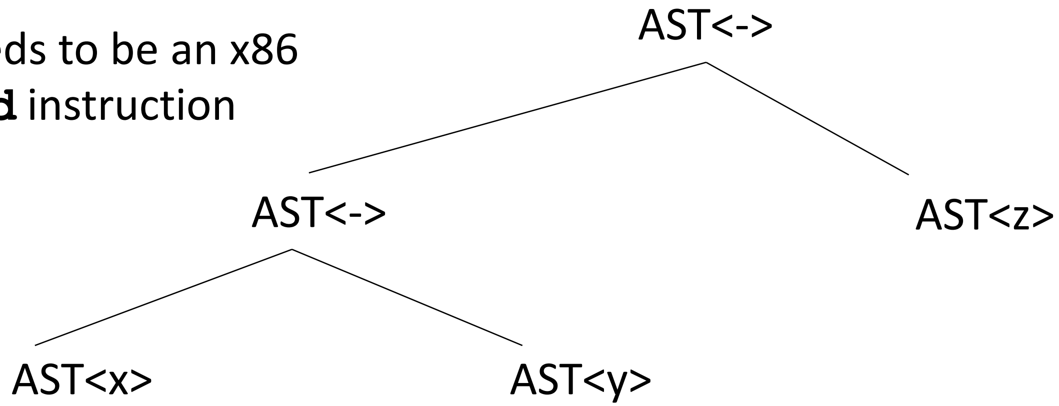
Evaluate an AST by doing a post order traversal

```
Expr  ::= NUM Expr2
Expr2 ::= MINUS 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



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

We cannot just subtract them!

Is this all?

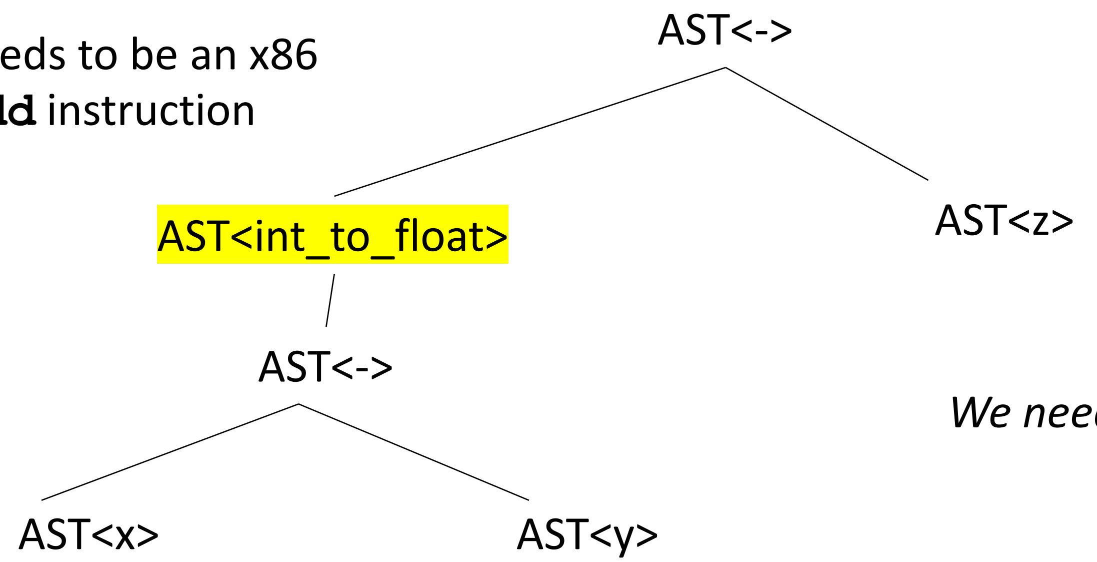
Evaluate an AST by doing a post order traversal

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

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

needs to be an x86
`addss` instruction

needs to be an x86
`add` instruction



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

- 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

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

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

```
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

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

Can write more generic code

```
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:
  - 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

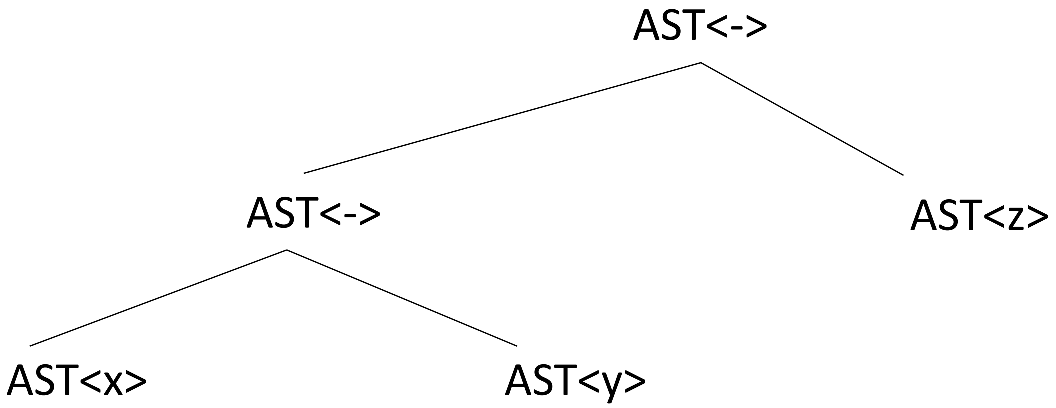
- Type inference
  - Determines a type for each AST node
  - Modifies the AST into a type-safe form

- Catches type-related errors

Type checking on an AST

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

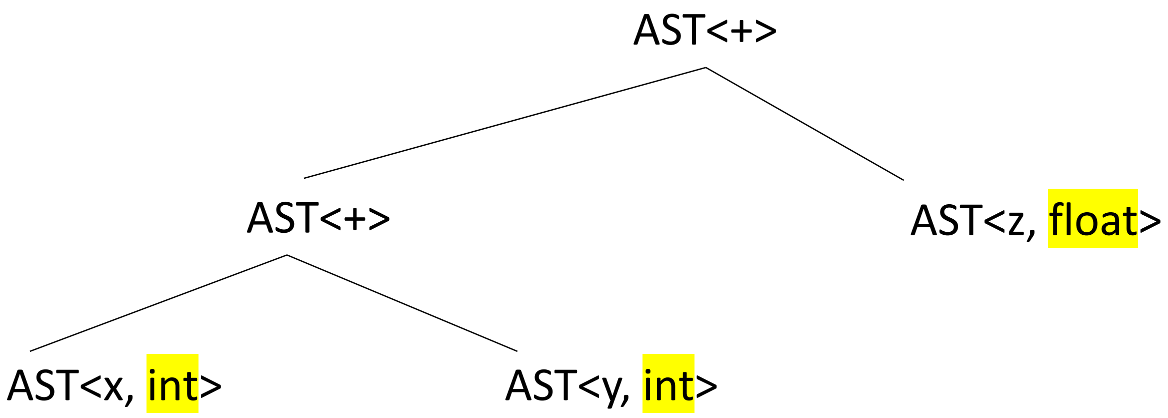
each node additionally gets a type



Type checking on an AST

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

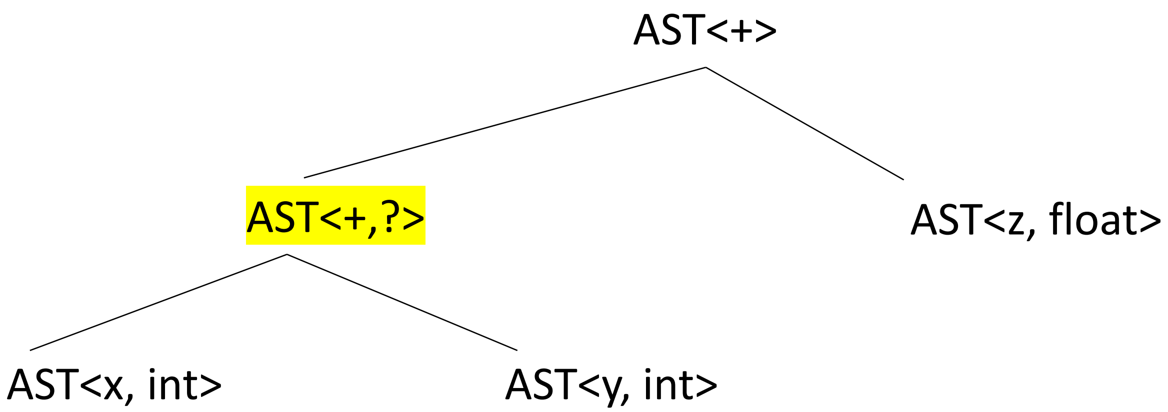
each node additionally gets a type
we can get this from the symbol table for the leaves or based on the input (e.g. 5 vs 5.0)



Type checking on an AST

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

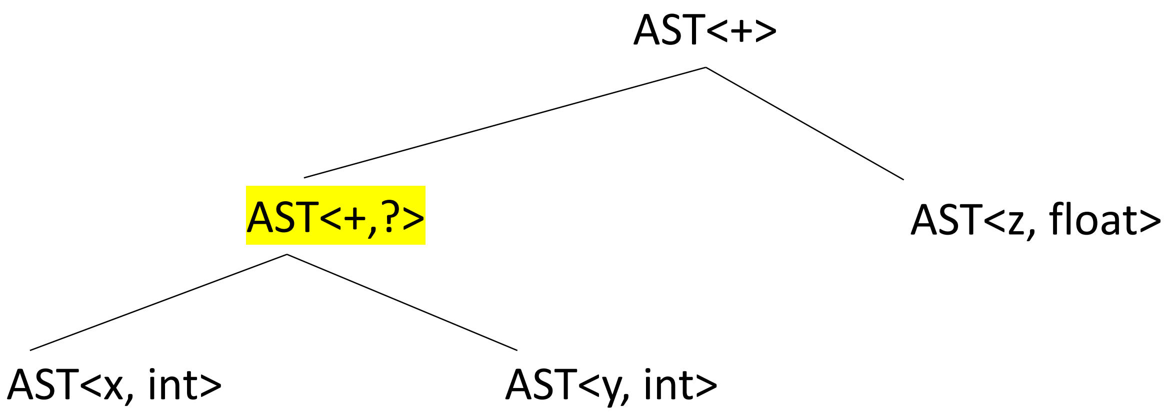
How do we get the type for this one?



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?



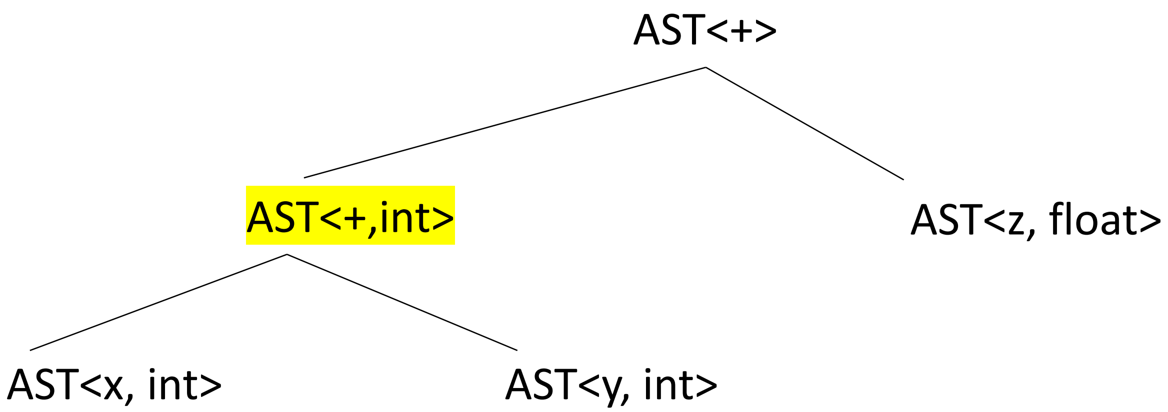
inference rules for addition:

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?



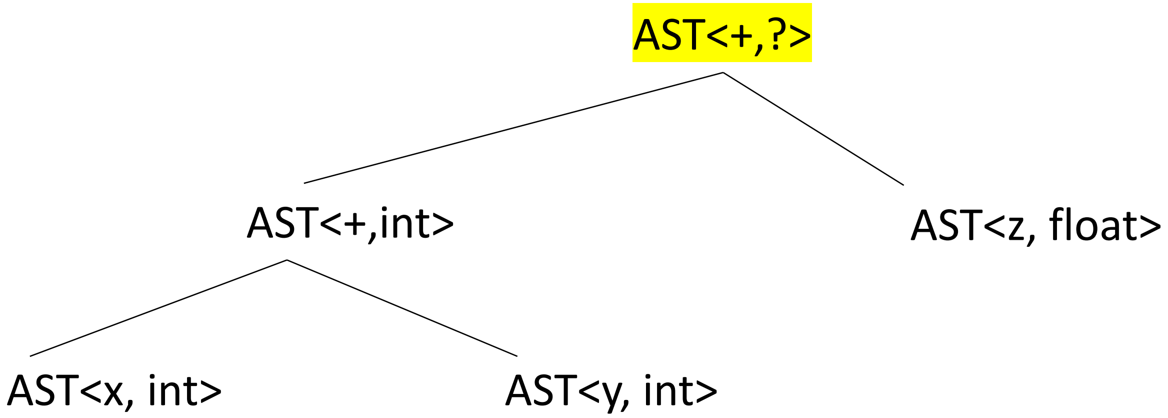
inference rules for addition:

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?



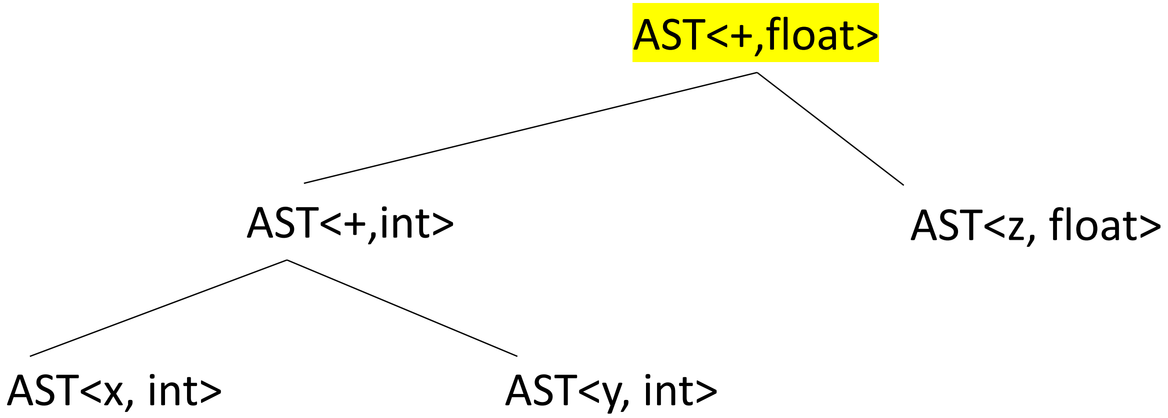
inference rules for addition:

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?



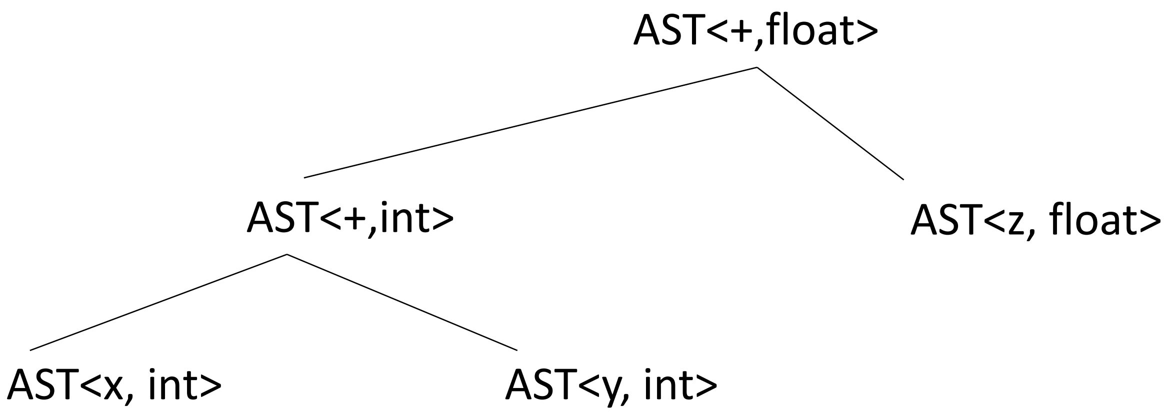
inference rules for addition:

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:

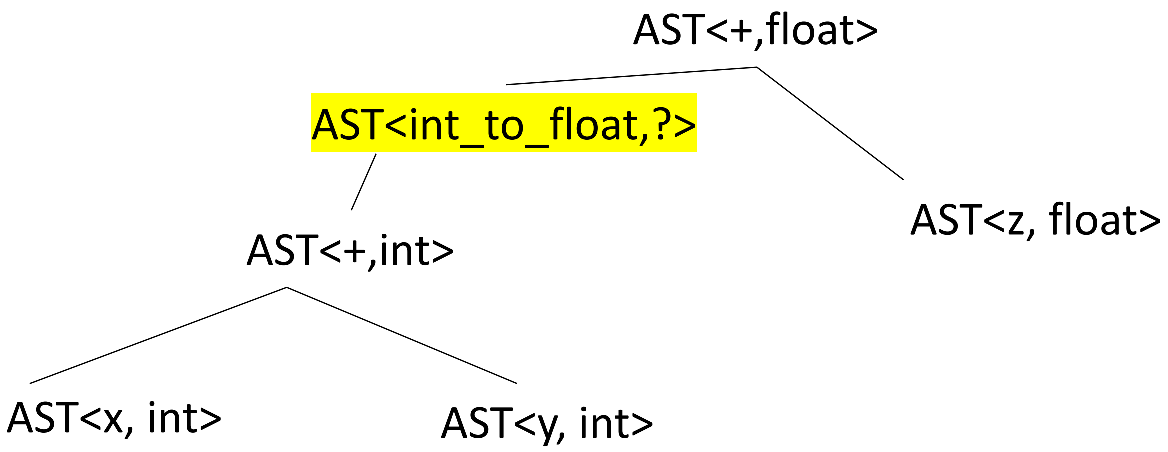
first	second	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

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.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

Number impies a type: e.g. 134 vs. 134.5

```
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

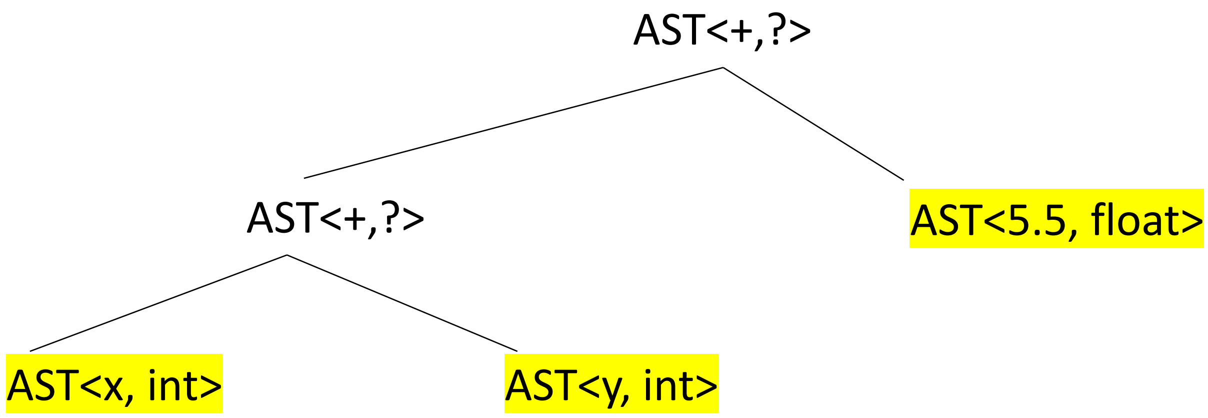
```
Unit ::= ID
     |   NUM
```

```
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

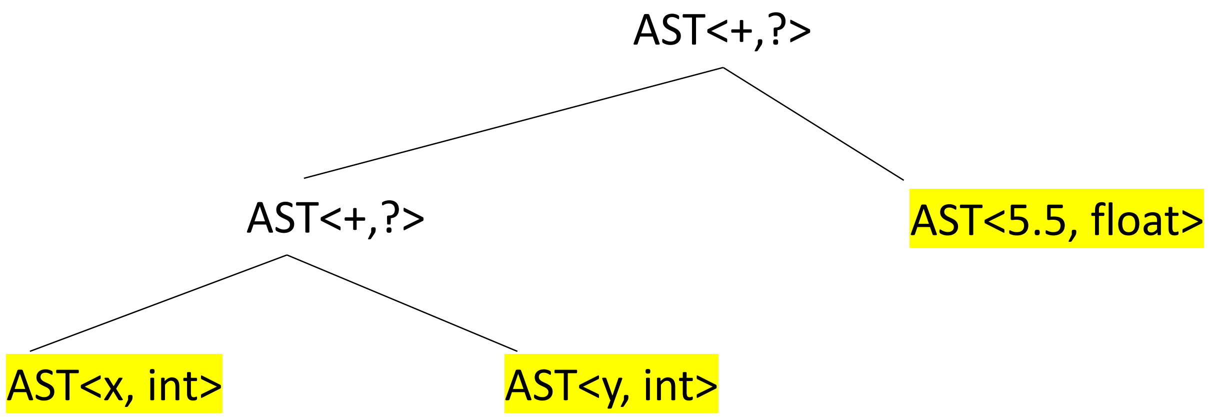
- We now have the types for the leaf nodes

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



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 node n:

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

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 node 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 node 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):

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`

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):

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
```

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()

do type inference on children
if n is a plus node:

   t = lookup type from table
   set n type to t
   return t
```

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

Assignments are expressions in C/C++

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

Assignments are
expressions in C/C++

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 (the type of the ID)

Type inference

Assignments are expressions in C/C++

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 (the type of the ID)