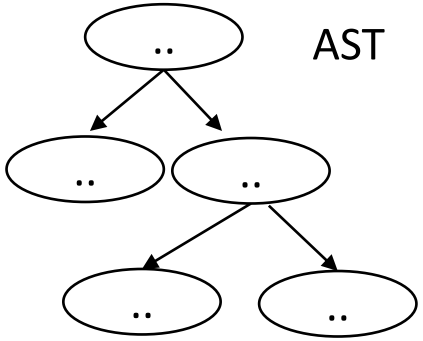
CSE110A: Compilers

May 3, 2024

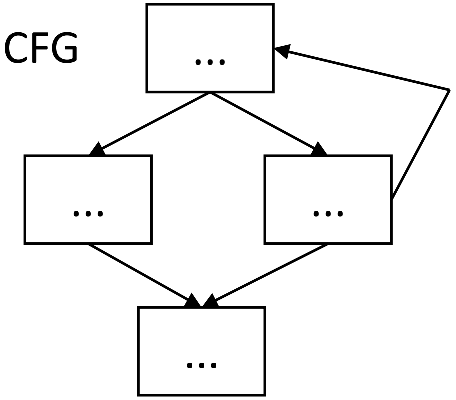
Topics:

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

AST



CFG



3 address code

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

Announcements

- Homeworks
  - HW 1 grades are coming
    - Aiming for Monday
  - HW 3 is out now, along with the grammar
    - Due on May 9
    - Time to study for the midterm

- Midterm will be given on Monday: May 6
  - Taken during class
  - 3 pages of notes are allowed
  - Study:
    - Slides
    - Homeworks
    - book readings

Midterm

- Given on Monday
- ~3 questions with multiple parts
- I will not be there
  - Proctored by Rithik and Sakshi and some tutors
- Split between 2 rooms
  - This room + Oakes 106
  - Will get an email about which room you should go to. Based on last name

Midterm study guide (so far)

Any of the following are fair game. Anything not listed below but in the lectures are fair game. Any combination of topics is fair game. This is only meant to be an overview of what we have discussed so far.

Midterm study guide (so far)

- Regular expressions
  - Operators, how to specify, how match vs full match works
- Scanners
  - What the API is, how strings are tokenized, how to specify tokens, token actions
- Grammars
  - How to specify a grammar, how to identify/avoid ambiguous grammars, how to show a derivation for match, parse trees
  - How to re-write grammars not to be left recursive, how to identify first+ sets
  - How the top down parsing algorithm works, how a recursive decent parser works
- Symbol tables
  - How scope can be tracked and manage during parse time, symbol table specification and implementation

- No material from Module 3

Quiz

Quiz

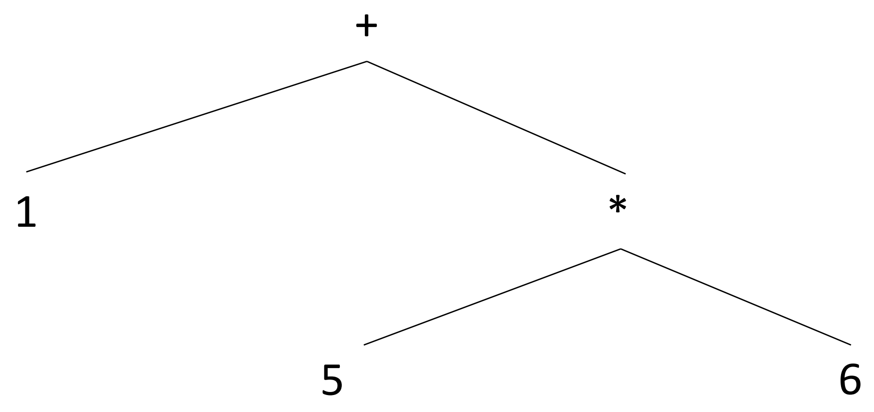
A parse tree is an abstract syntax tree

○ True

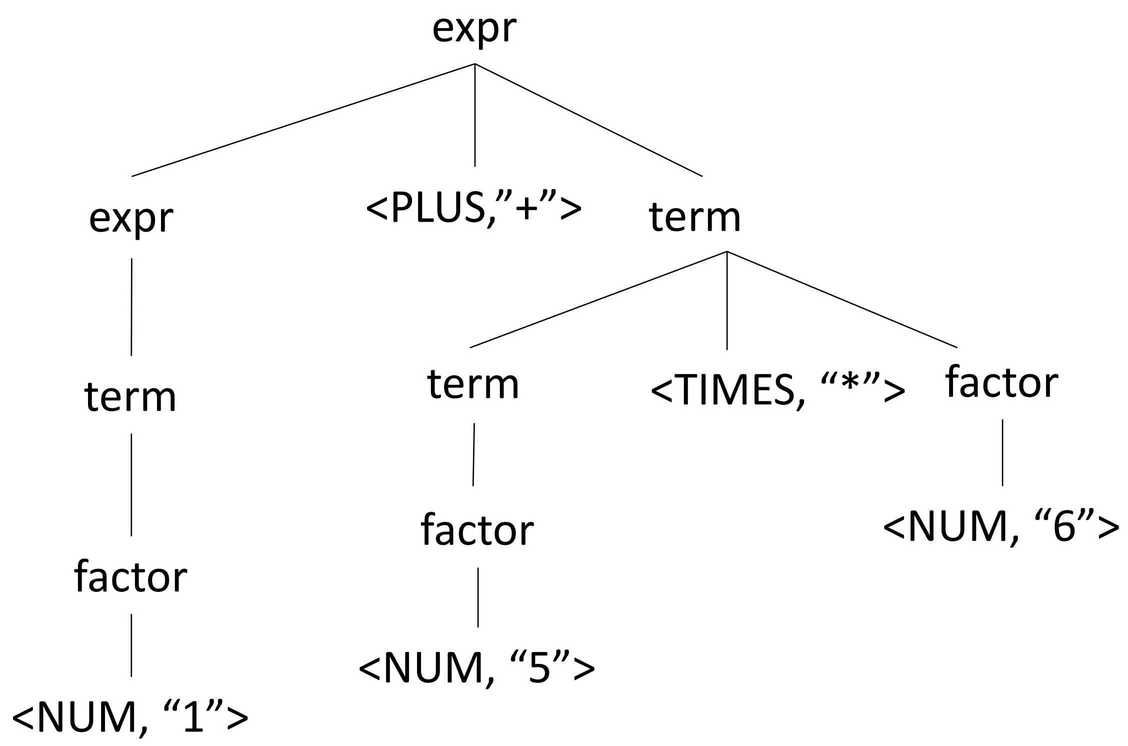
○ False

What is an AST?

`input: 1+5*6`



AST



What are some differences?

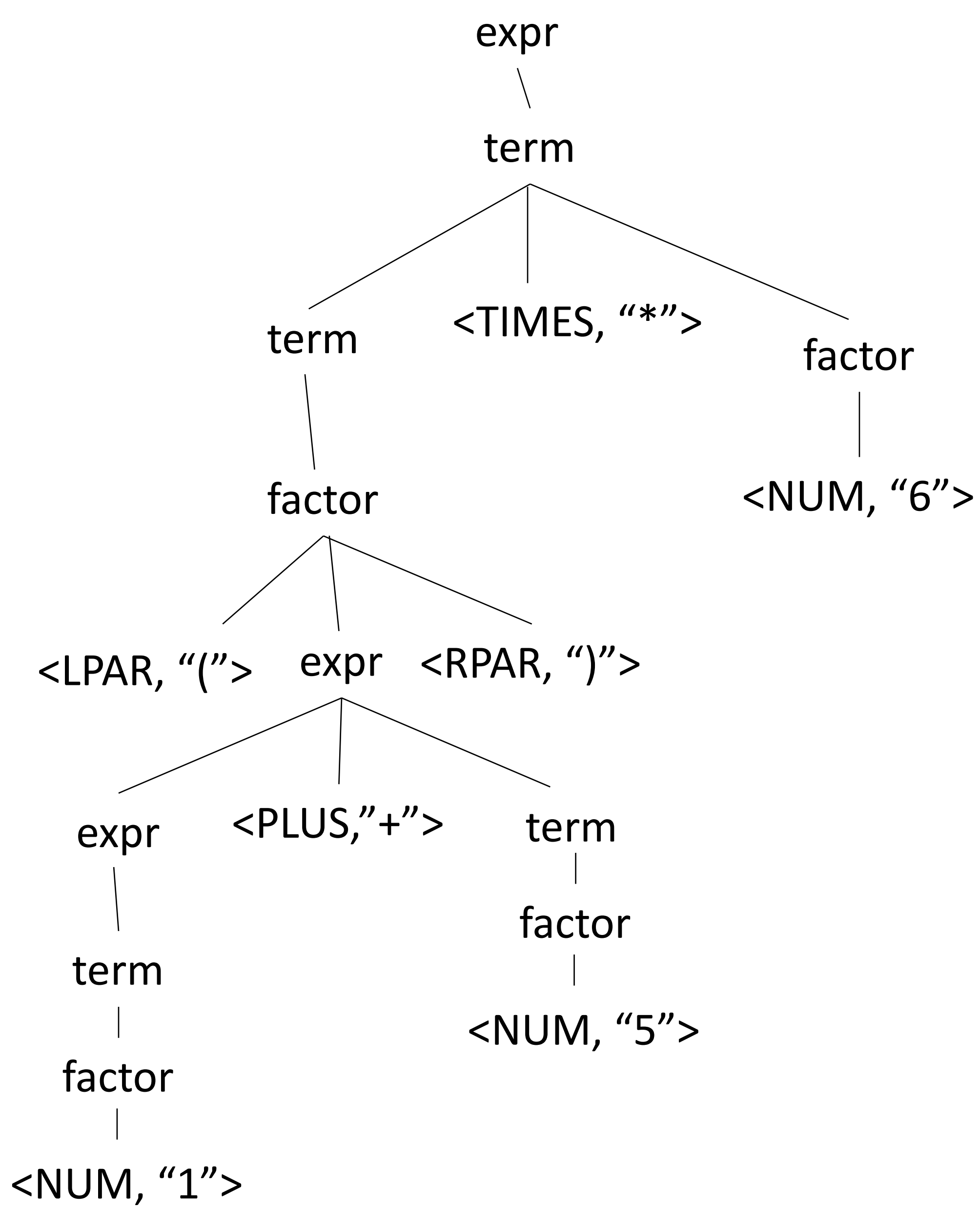
- disjoint from the grammar
- leaves are data, not lexemes
- nodes are operators, not non-terminals

Example

what happens to ()s in an AST?

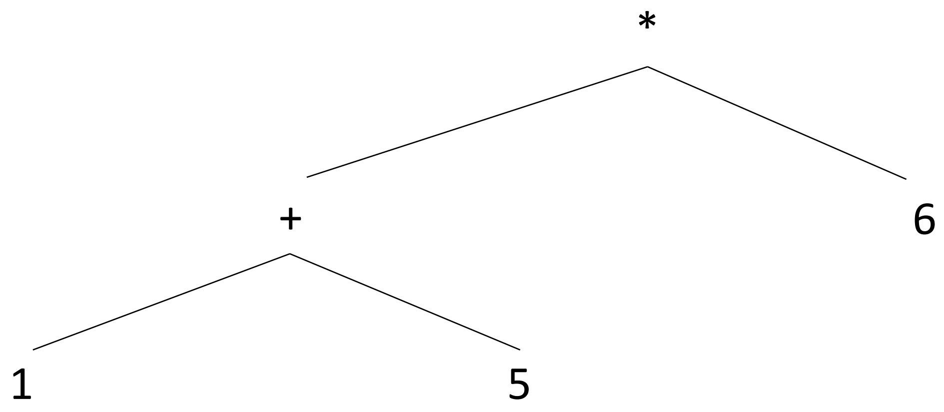
Operator	Name	Productions
+	`expr`	`: expr PLUS term` `| term`
*	`term`	`: term TIMES factor` `| factor`
()	`factor`	`: LPAR expr RPAR` `| NUM`

`input: (1+5)*6`

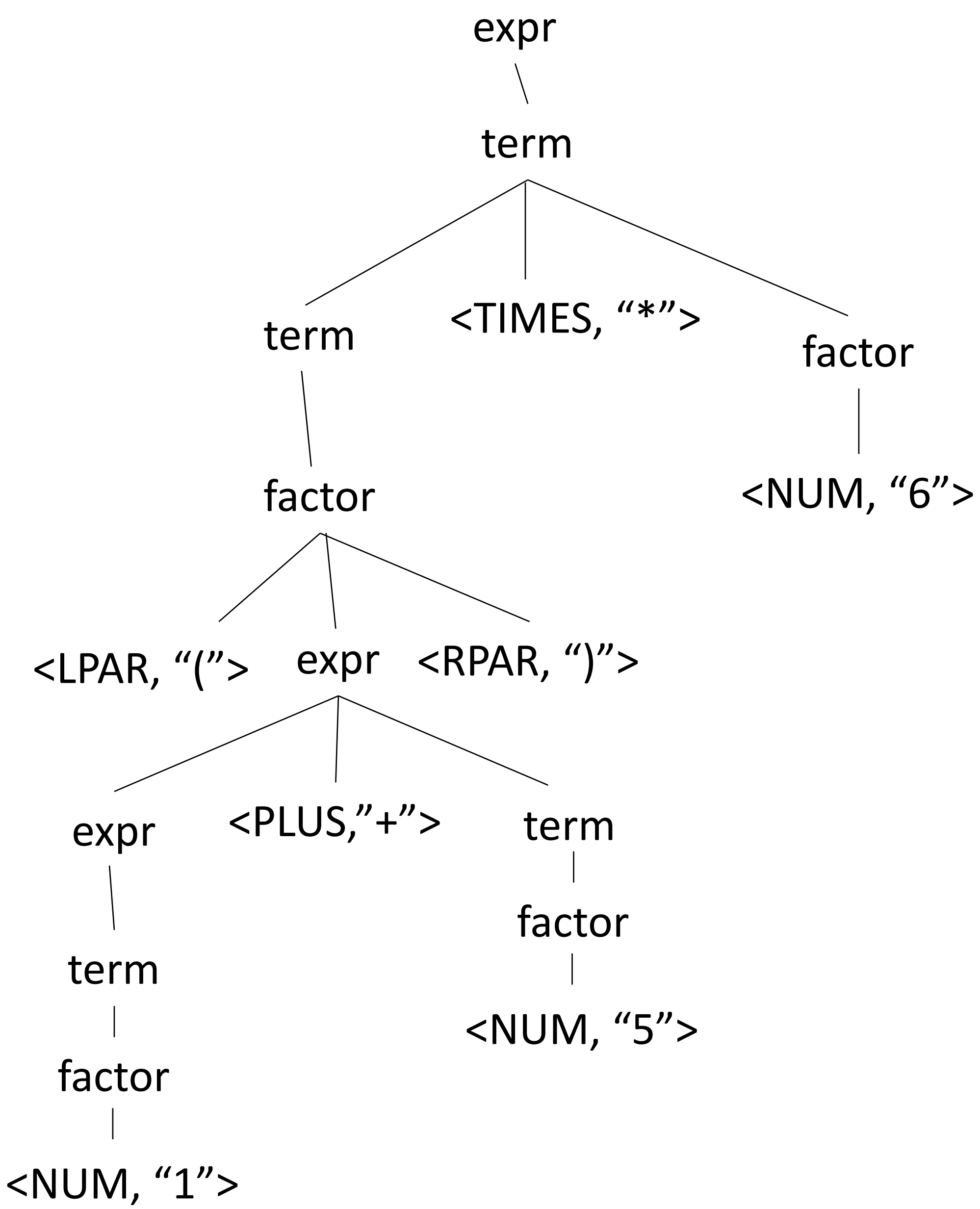


Example

what happens to ()s in an AST?



No need for (), they simply encode precedence. And now we have precedence in the AST tree structure



Quiz

If you are writing a compiler on M languages for N target architectures. How many components (front end or backend) will you need to write with the help of an IR? How about without an IR

- ○ M, N
- ○ MN, M+N
- ○ M+N, MN
- ○ MN, NM
- ○ M, NM
- ○ M, N + M

Quiz

Loop unrolling will ____ loop overhead and ____ program code size

- ○ increase, increase
- ○ increase, reduce
- ○ reduce, increase
- ○ reduce, reduce

Example: loop unrolling

```
for (i = 0; i < 102; i = i++) {
   x = x + 1;
}
```

Quiz

Name and discuss few Intermediate Representations you have seen in real life. If you have not used or seen any, then describe some that you might have been using without knowing.

ASTs

```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

```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 production rules

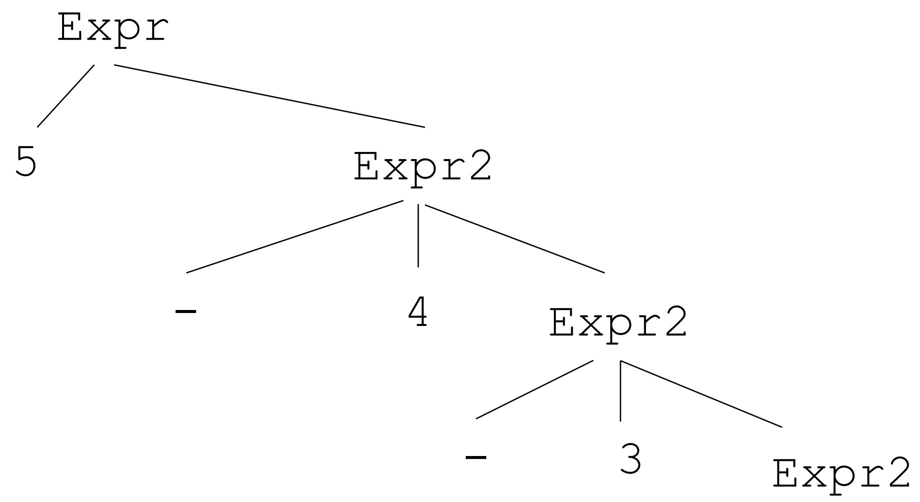
Operator	Name	Productions	Production action
+	`expr`	`: expr PLUS term` `| term`	`{return ASTAddNode($1,$3)}` `{return $1}`
*	`term`	`: term TIMES factor` `| factor`	`{return ASTMultNode($1,$3)}` `{return $1}`

()	`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
       |    ""
```

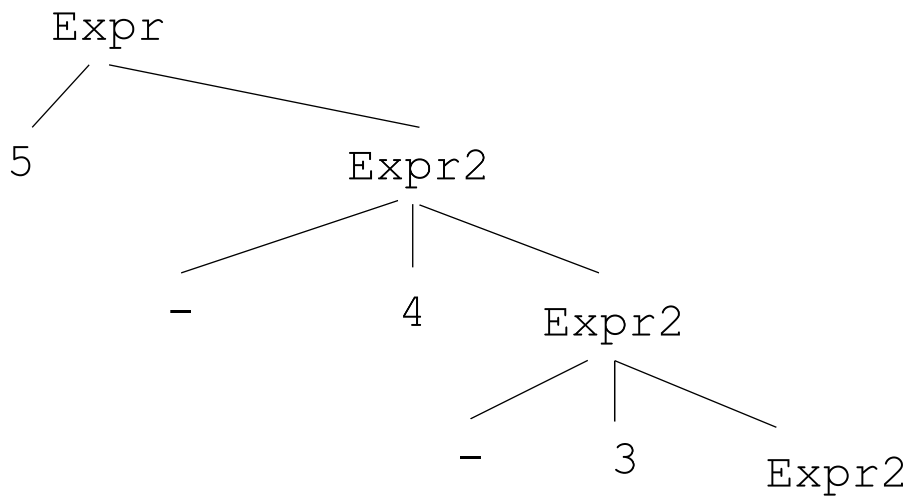
5 - 4 - 3

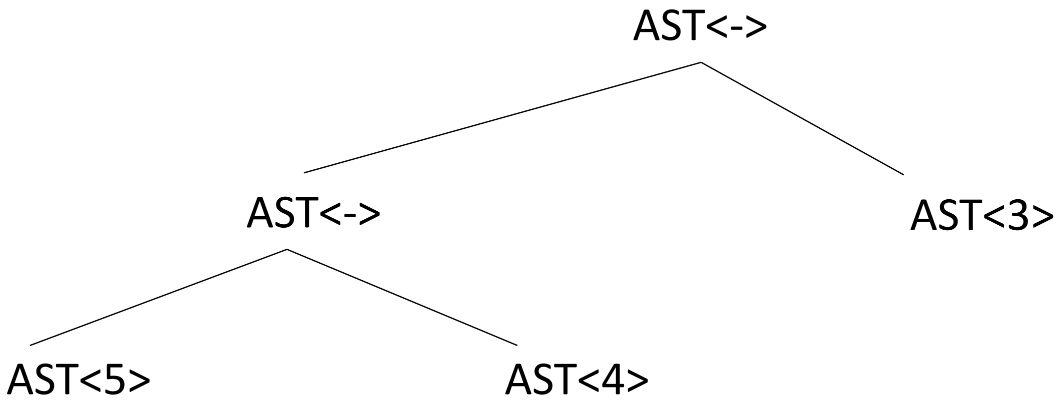


Creating an AST from top down grammar

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

5 - 4 - 3



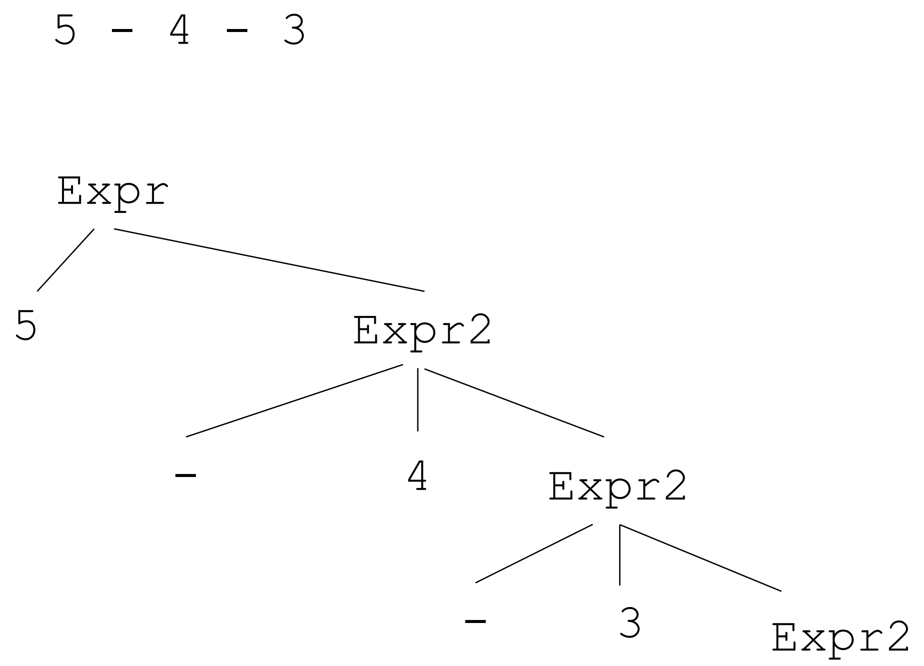


How do we get to the desired parse tree?

Creating an AST from top down grammar

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

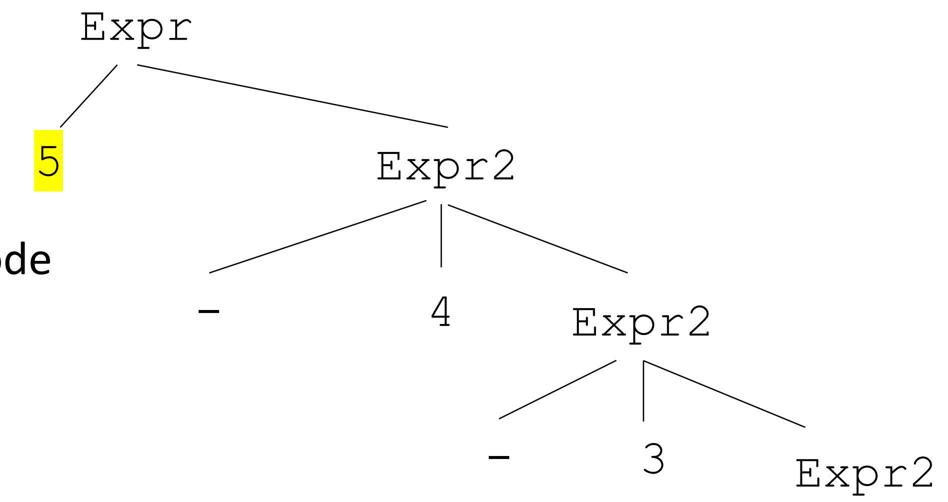
Keep in mind that because we wrote our own parser, we can inject code at any point during the parse.



Creating an AST from top down grammar

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

5 - 4 - 3



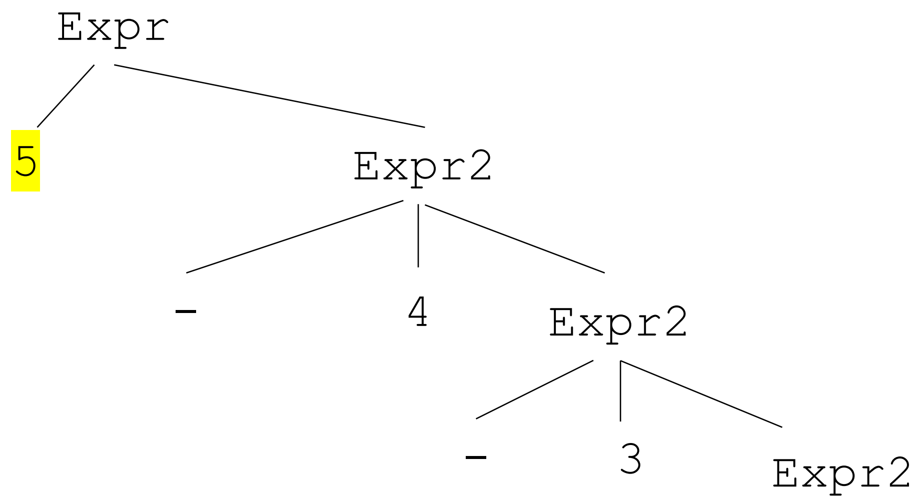
Get number node after we see a number

AST<5>

Creating an AST from top down grammar

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

5 - 4 - 3



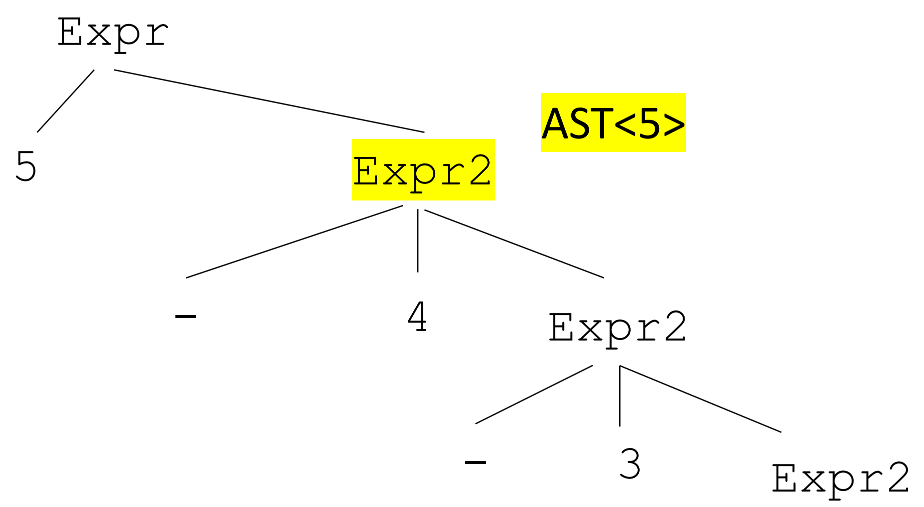
Pass the node down

AST<5>

Creating an AST from top down grammar

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

5 - 4 - 3



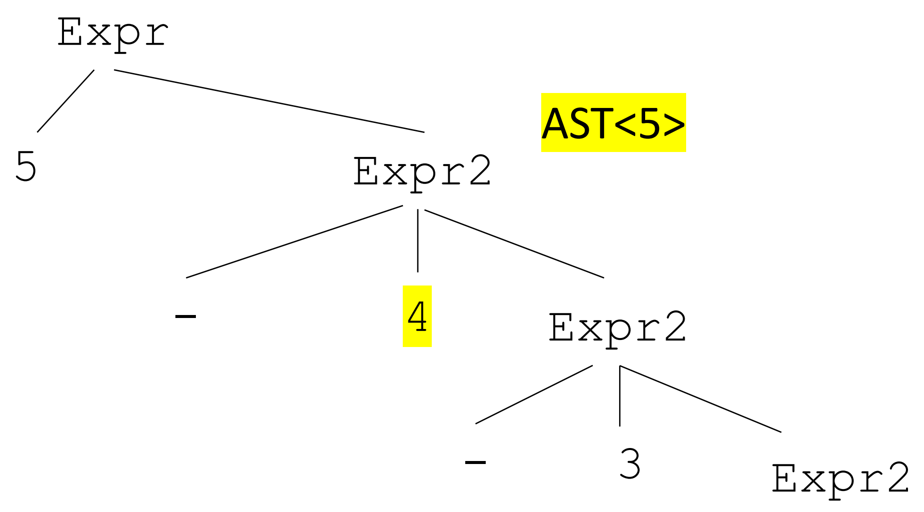
Pass the node down

AST<5>

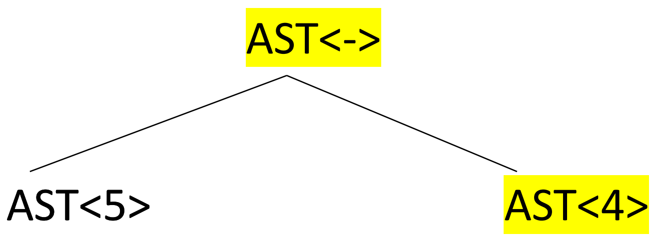
Creating an AST from top down grammar

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

5 - 4 - 3



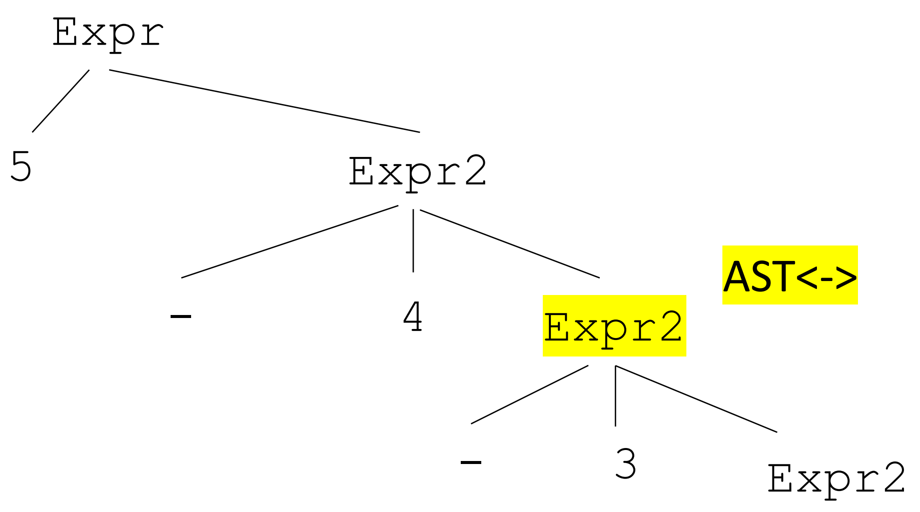
In Expr2, after 4 is parsed, create a number node and a minus node



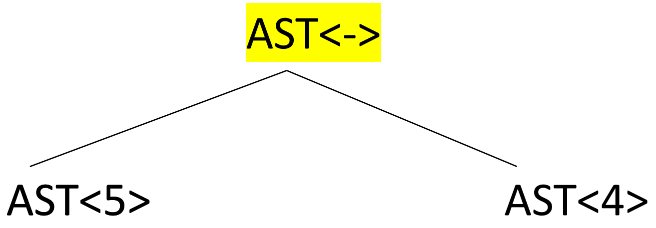
Creating an AST from top down grammar

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

5 - 4 - 3



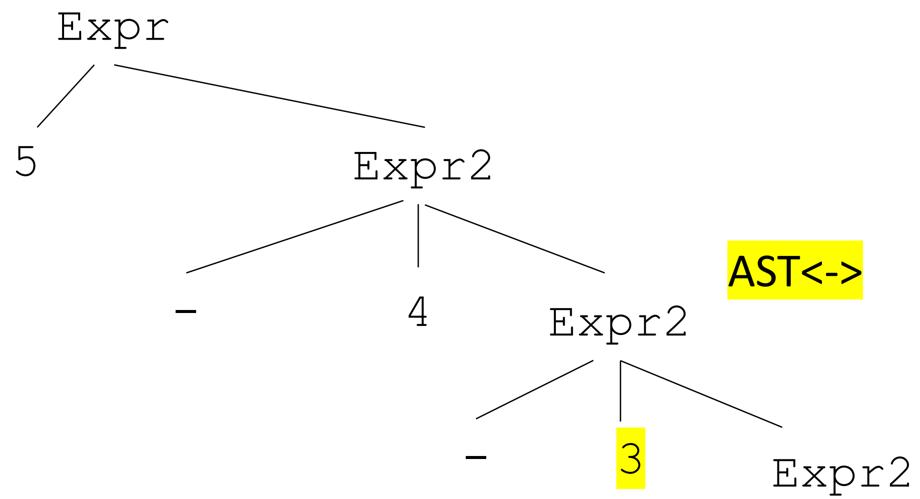
pass the new node down

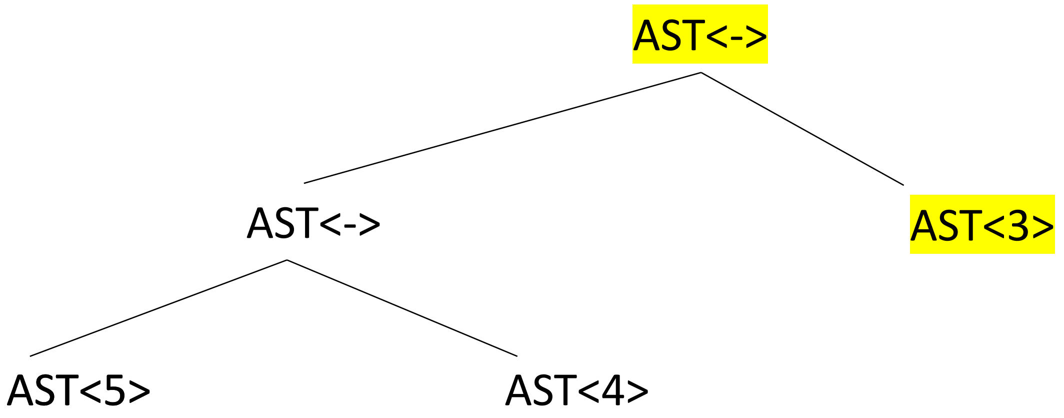


Creating an AST from top down grammar

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

5 - 4 - 3



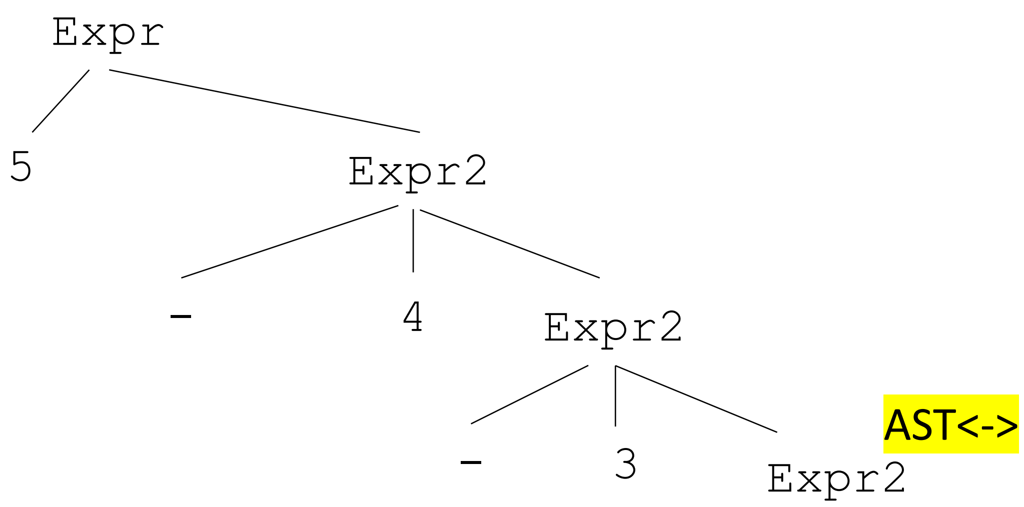


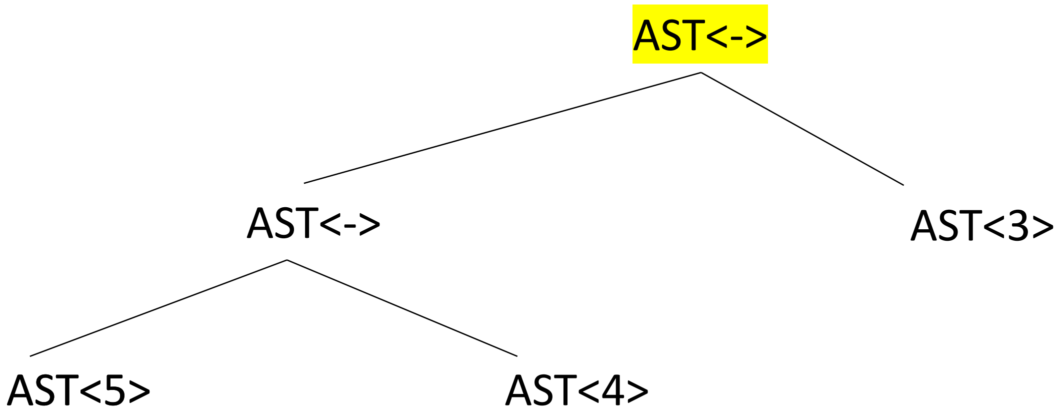
In Expr2, after 3 is parsed, create a number node and a minus node

Creating an AST from top down grammar

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

5 - 4 - 3



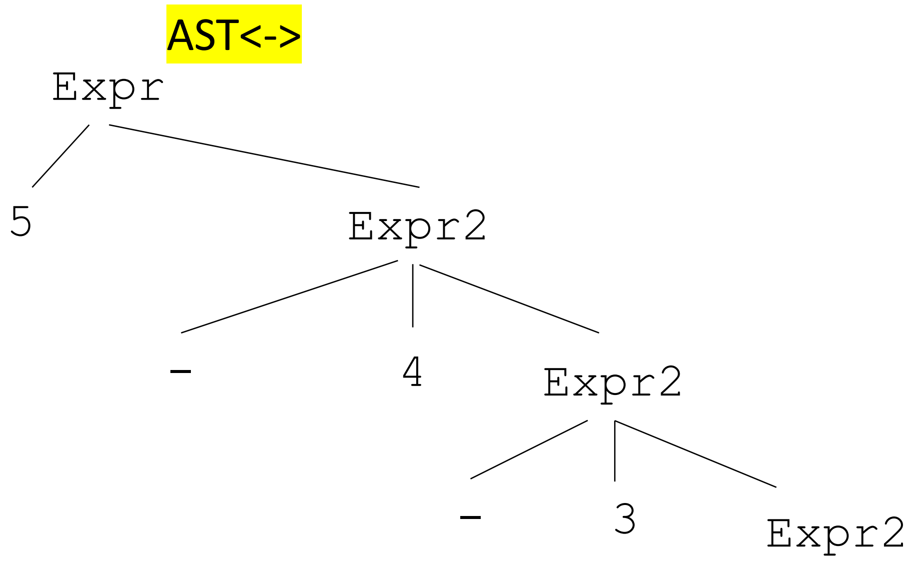


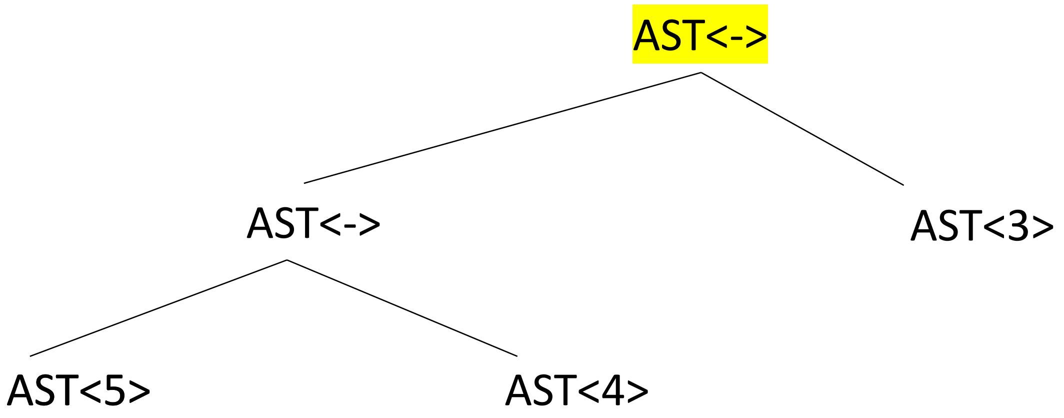
pass down the new node

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
        |    ""
```

```python
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)
```

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
        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  ::= 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
    return lhs_node
```

Creating an AST from top down grammar

```
Expr   ::= Term Expr2
Expr2  ::= MINUS Term 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)
```

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")
        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?

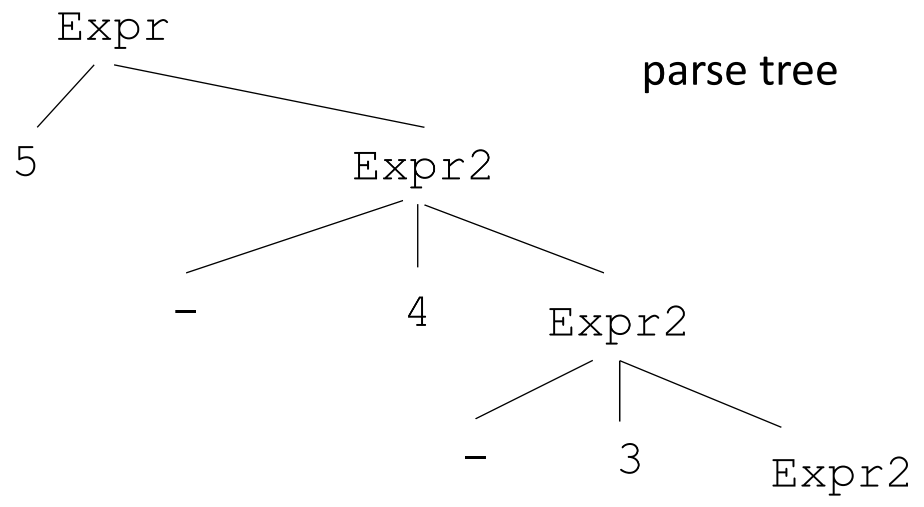
```
def parse_expr2(self, lhs_node):
        # ... for applying the first production rule
        self.eat("MINUS")
        rhs_node = self.parse_term()
        node = ASTMinusNode(lhs_node, rhs_node)
        return self.parse_expr2(node)
```

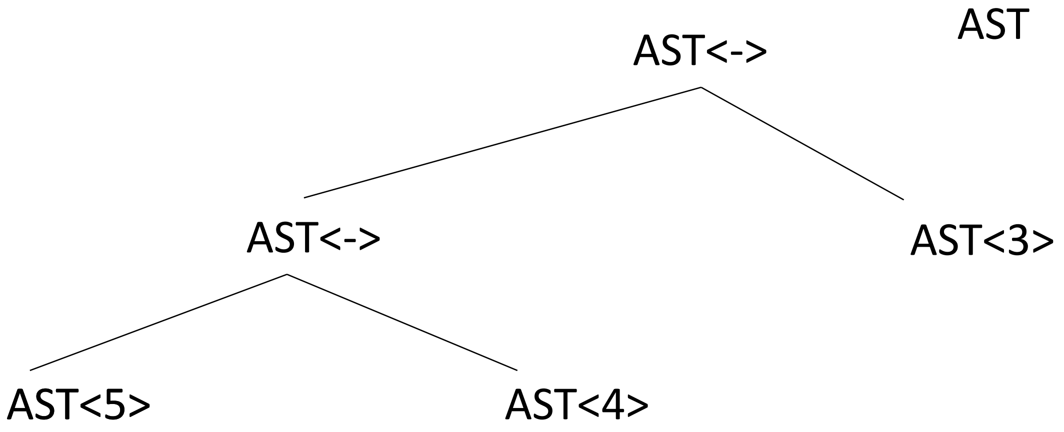
The parse_term will figure out how to get you an AST node for that term.

Evaluate an AST by doing a post order traversal

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

`5 - 4 - 3`





Parse 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')))
```

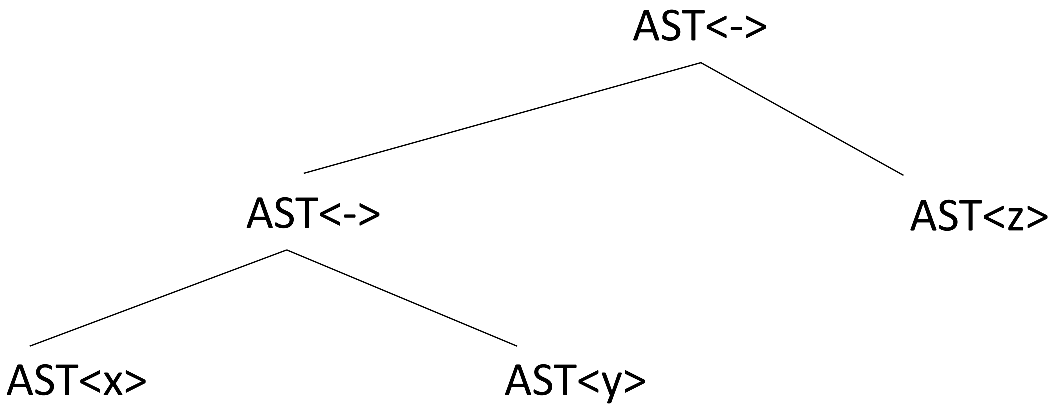
```
Expr(value=BinOp(left=BinOp(left=Num(n=5), op=Sub(), right=Num(n=4)), op=Sub(), right=Num(n=2)))
```

Evaluate an AST by doing a post order traversal

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

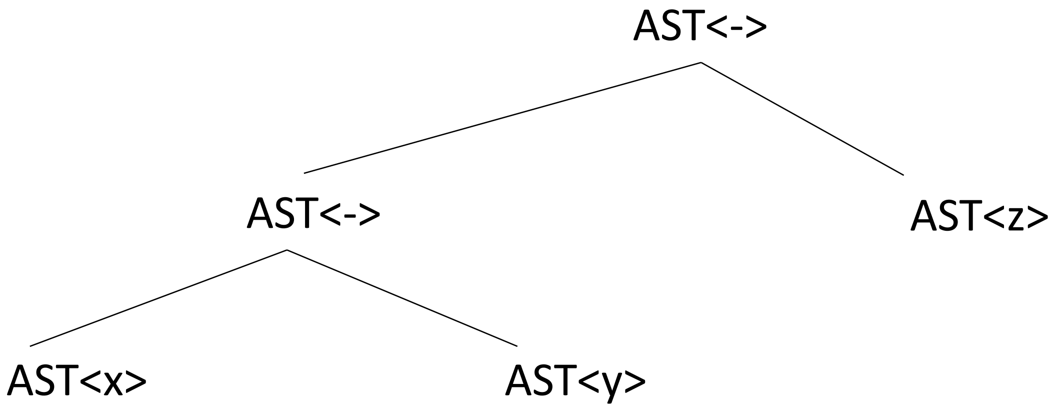
What if you cannot evaluate it?
What else might you do?

`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?
What else might you do?

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

How does this change things?

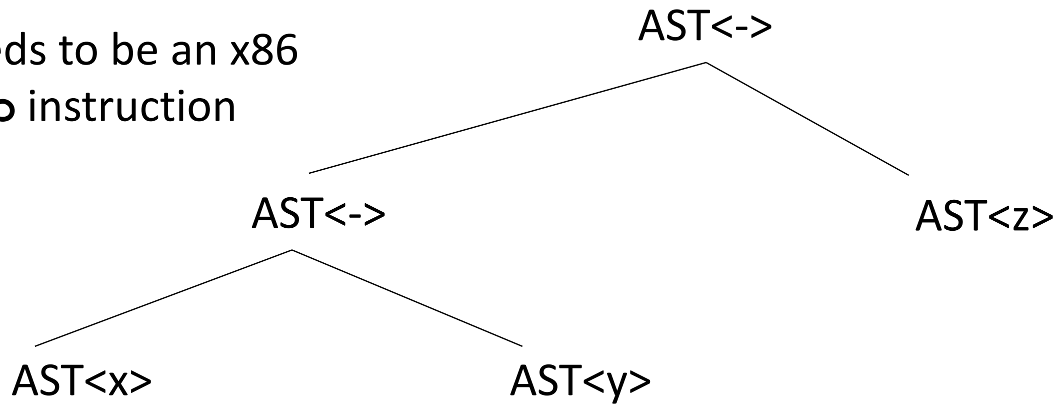
Evaluate an AST by doing a post order traversal

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

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

needs to be an x86 **`subss`** instruction

needs to be an x86 **`sub`** instruction




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

How does this change things?

Is this all?

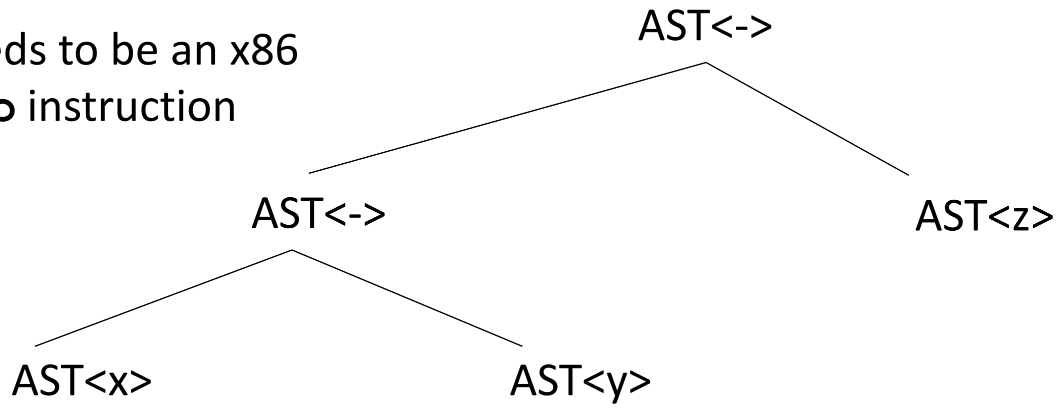
Evaluate an AST by doing a post order traversal

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

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

needs to be an x86 **subss** instruction

needs to be an x86 **sub** instruction



Lets do some experiments.

What should 5 - 5.0 be?

Is this all?

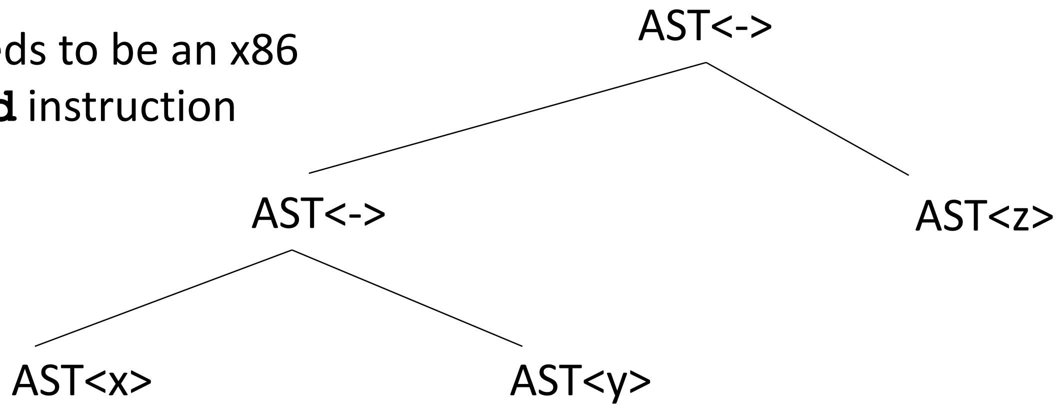
Evaluate an AST by doing a post order traversal

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

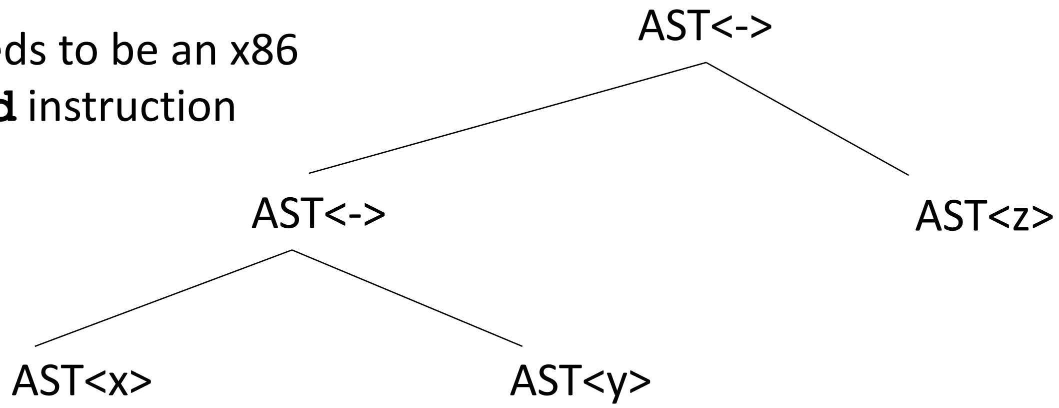
Evaluate an AST by doing a post order traversal

```
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?

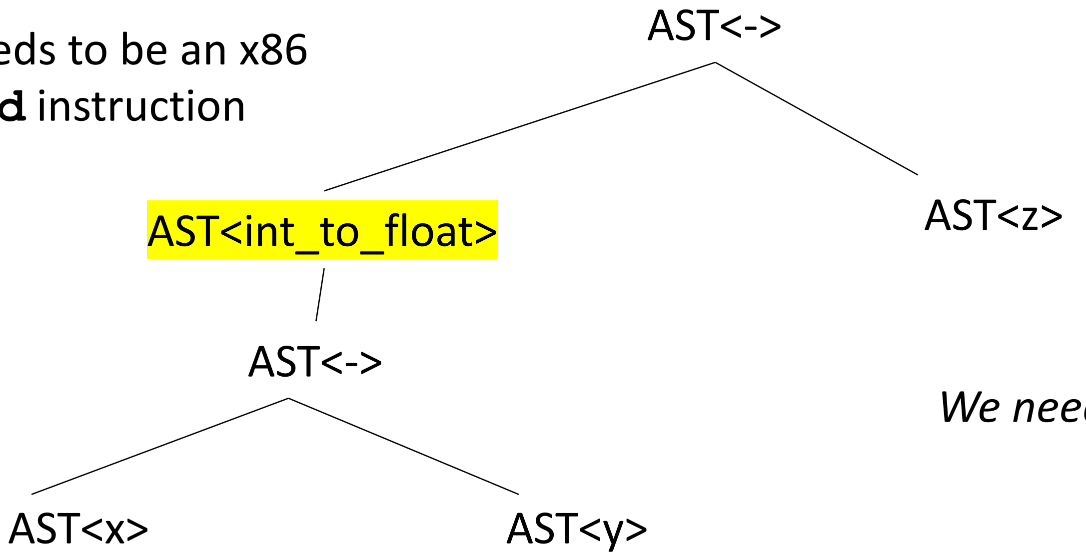
Evaluate an AST by doing a post order traversal

```
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

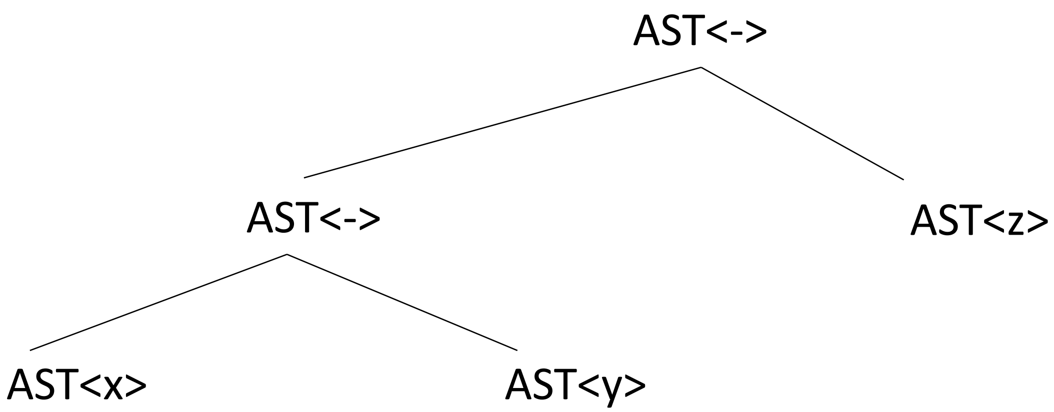
- Type inference
  - Determines a type for each AST node
  - Modifies the AST into a type-safe form

- Catches type-related errors

Type checking on an AST

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

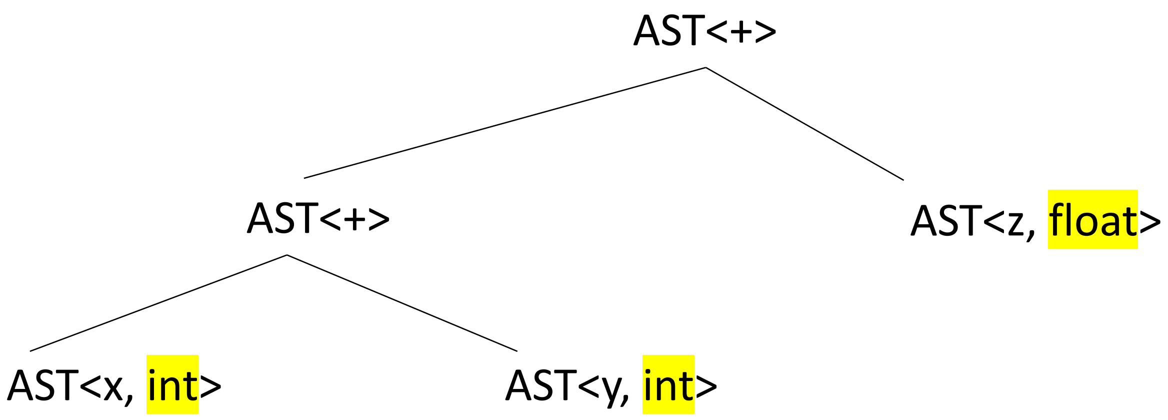
each node additionally gets a type



Type checking on an AST

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

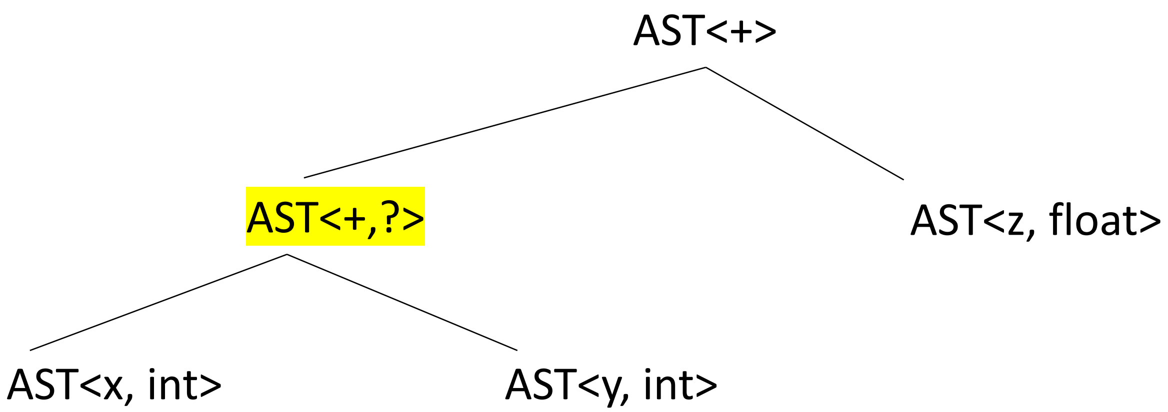
each node additionally gets a type
we can get this from the symbol table for the leaves or based on the input (e.g. 5 vs 5.0)



Type checking on an AST

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

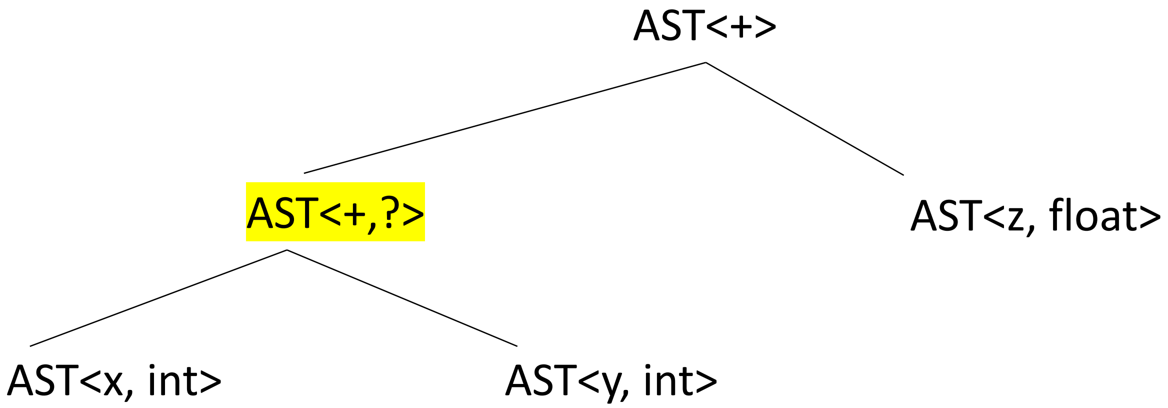
How do we get the type for this one?



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?



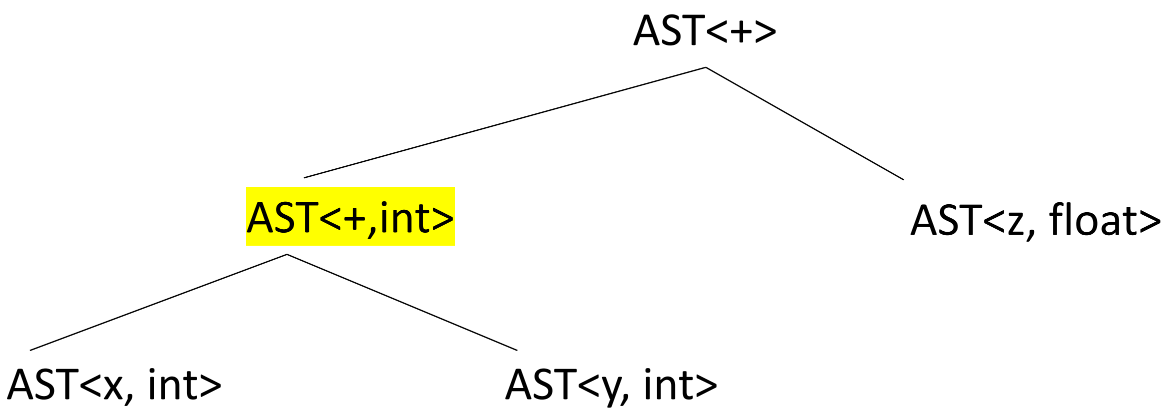
inference rules for addition:

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?



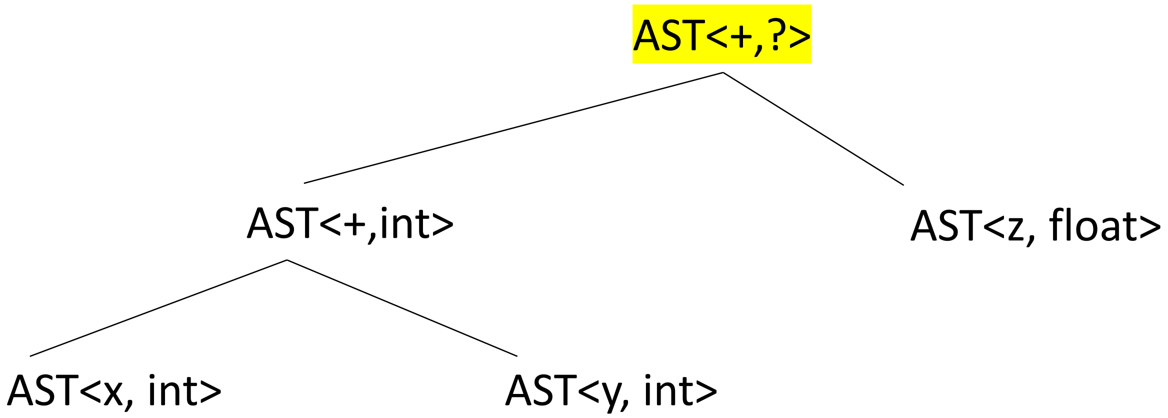
inference rules for addition:

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?



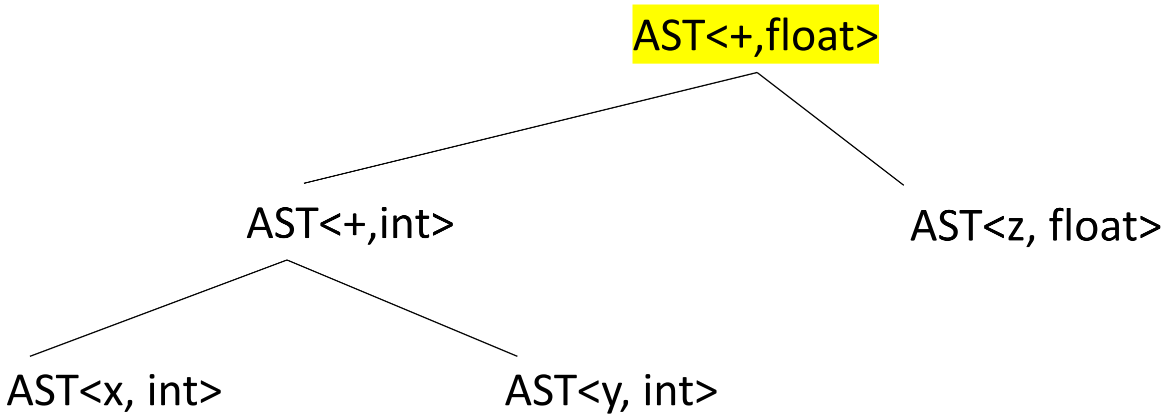
inference rules for addition:

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?



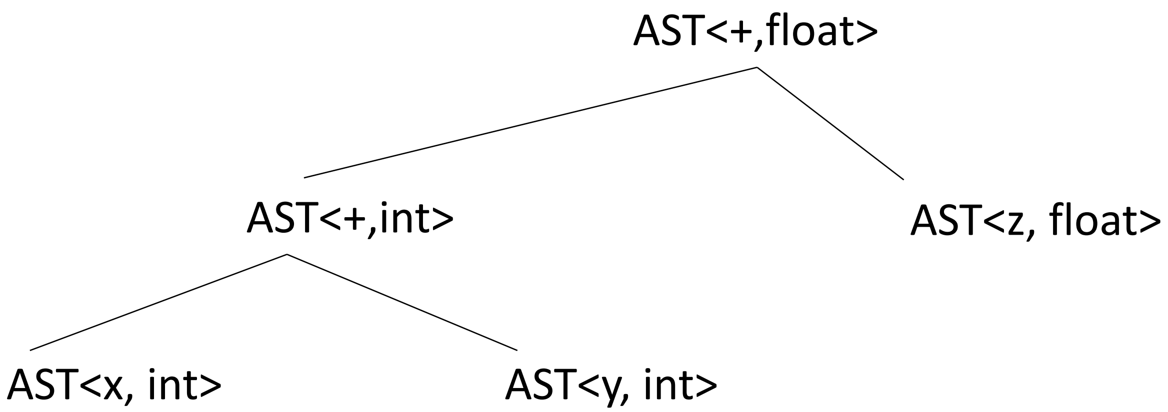
inference rules for addition:

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:

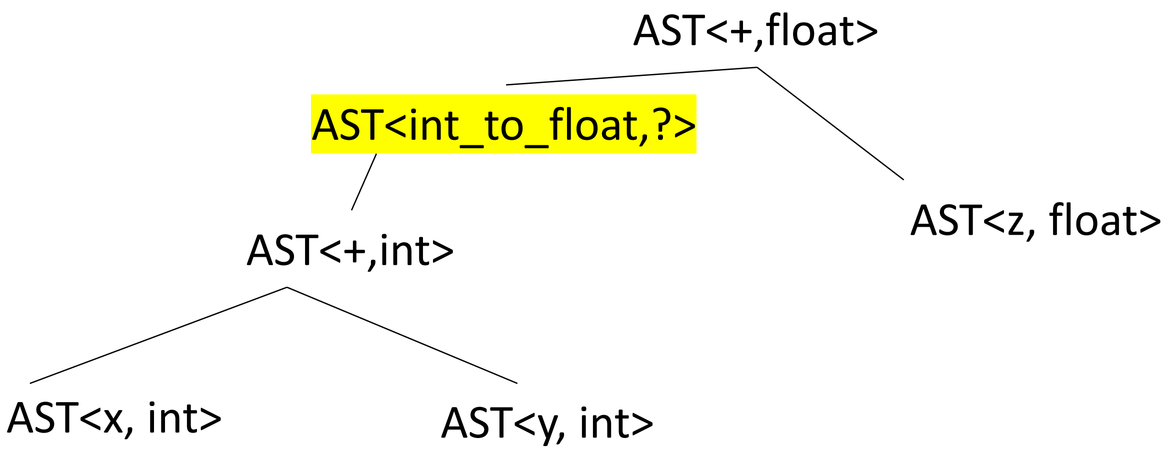
first	second	result
int	int	int
int	float	float
float	int	float
float	float	float

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

```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

```python
from enum import Enum

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

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)
```

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
```

```python
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

```python
from enum import Enum

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

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)
```

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
```

```python
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

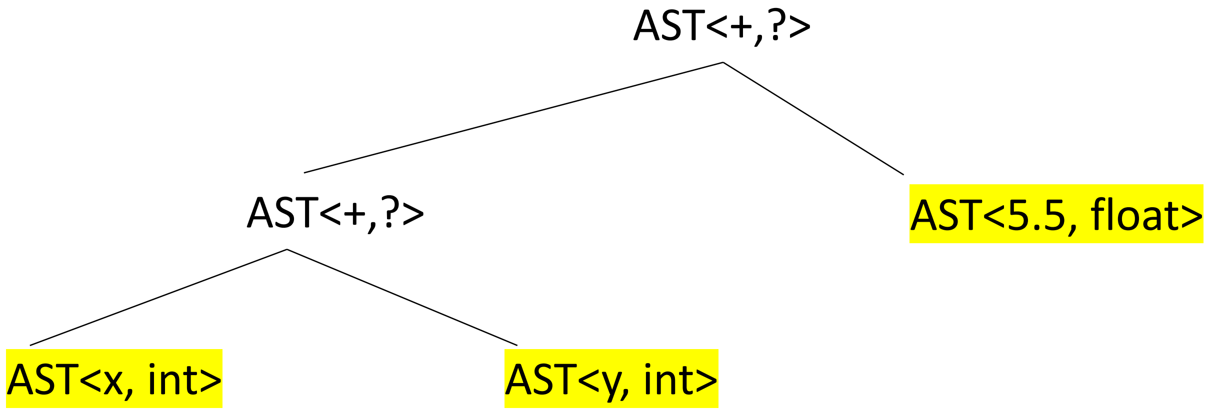
```
Unit ::= ID
     |   NUM
```

```
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

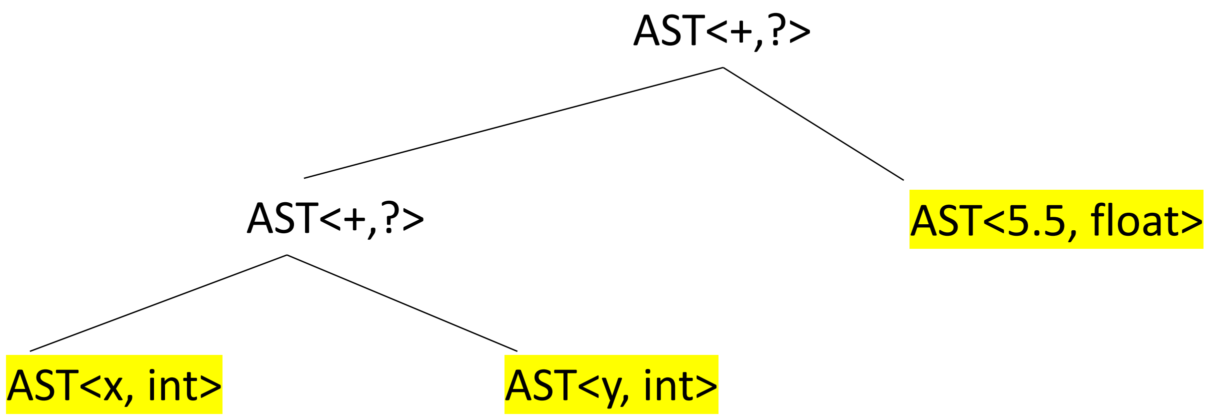
- We now have the types for the leaf nodes

```
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 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):
```

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):

 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):
```

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

```
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

Good luck with the test!

- Study for the test!