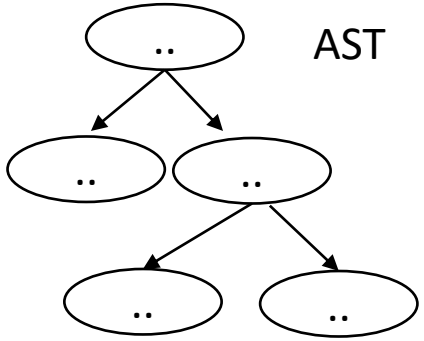
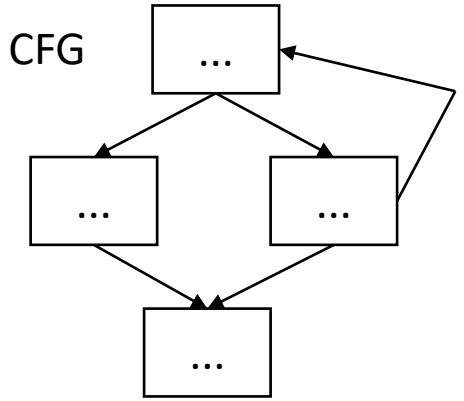
CSE110A: Compilers

May 10, 2023



Topics:

- *ASTs*
  - *parse trees into 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

- Midterm is over!
  - We will start grading early next week and aim to scores released by next Friday

Announcements

- HW 3
  - Released on Monday
  - 7 days to do the homework (due May 15)
    - Implementing a recursive descent parser for your grammar
    - Implementing a symbol table
  - You will need to implement a scanner
    - Use your HW 1 solution or the exact match scanner (given in HW 1)
  - You will need a grammar
    - Either use your grammar or the provided grammar (Rithik provided on Piazza)
- Come to office hours or post on Piazza for help!

Announcements

- HW 3 clarification: You do not need to return anything from your parser!
  - If the input program satisfies the grammar then you return without issue
  - If it does not, then you throw an exception
    - Scanner exception if you cannot create a token
    - Parser exception if the input violations the grammar
    - Symbol table exception if a variable is used outside of a scope it is declared
- HW 4 will be creating an IR inside your parser.

Quiz

Quiz

Parse tree is an Abstract Syntax Tree

- ◯ True
- ◯ False

Quiz

If you are writing a compiler on M languages for N target architectures. How many compilers will you need to write with and without the help of Intermediate Representation?

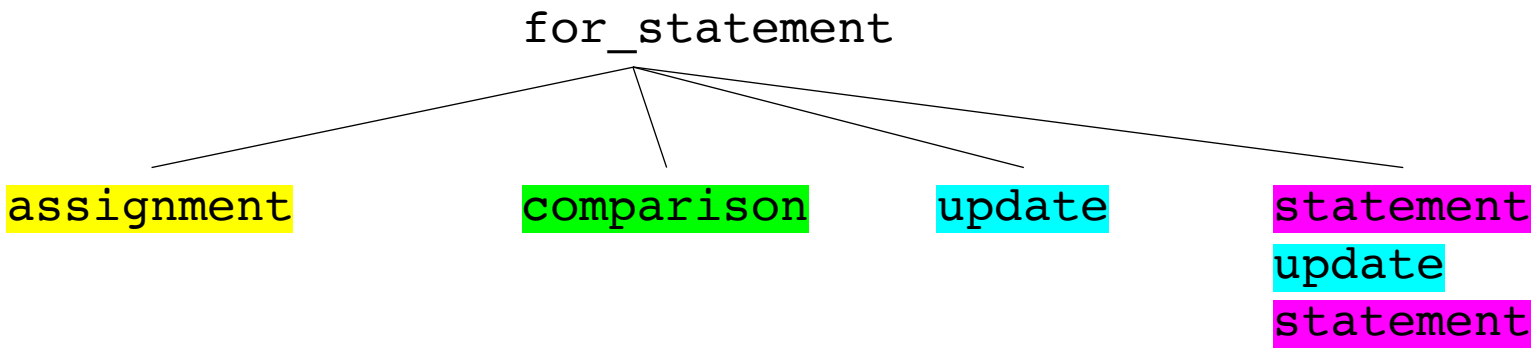
- ○ M, N
- ○ MN, M+N
- ○ M+N, MN
- ○ MN, NM
- ○ M, NM
- ○ M, N + M

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 < 100; i = i + 1) {
   x = x + 1;
}
```

Check:

1. Find iteration variable by examining assignment, comparison and update.

2. found i

3. check that statement doesn't change i.

4. check that comparison goes around an even number of times.

Perform optimization

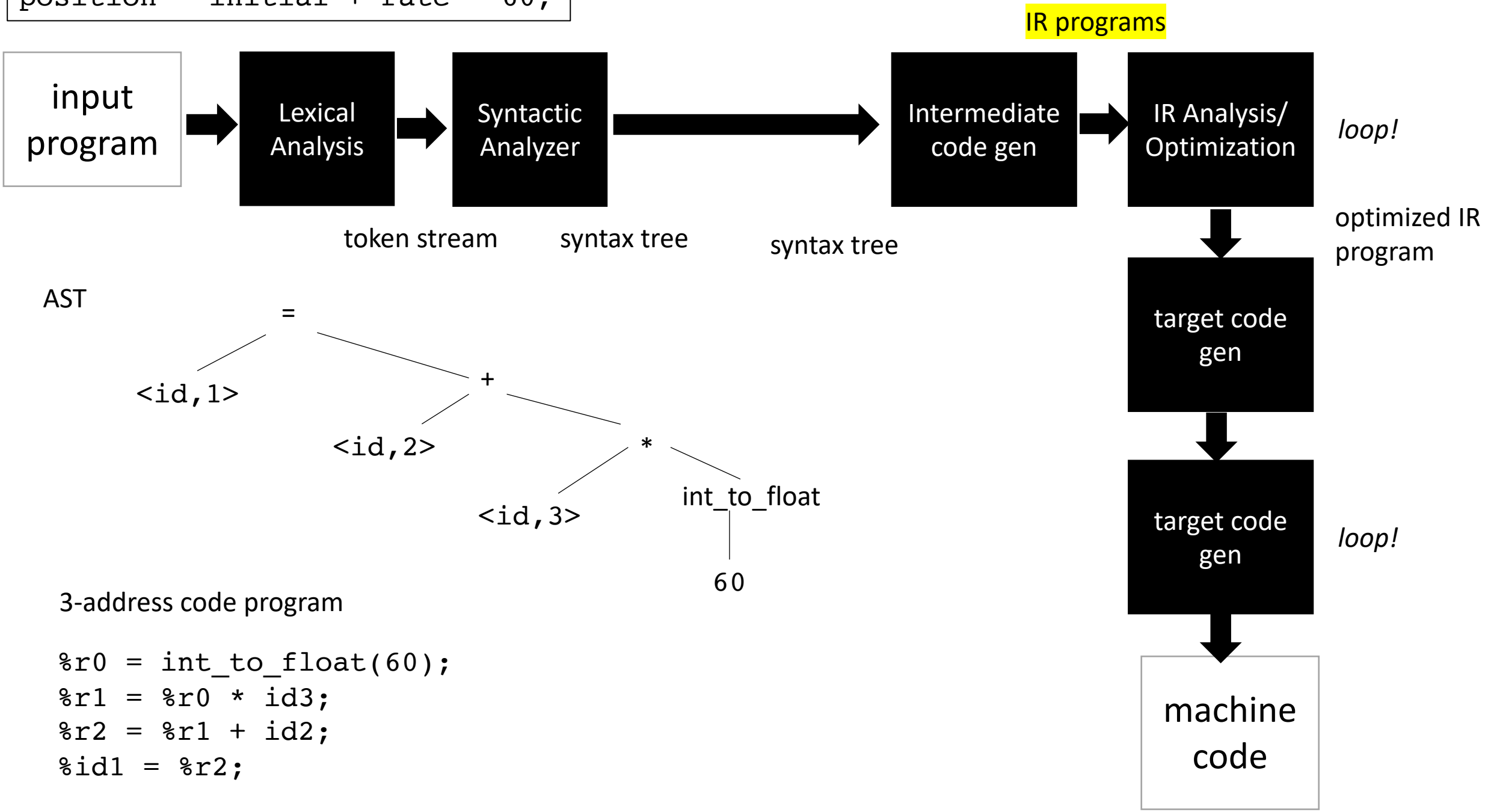
copy statement and put an update before it

Quiz

Name a few Intermediate Representations you have seen in real life

Review

```
position = initial + rate * 60;
```



AST

3-address code program

```
%r0 = int_to_float(60);
%r1 = %r0 * id3;
%r2 = %r1 + id2;
%id1 = %r2;
```

Intermediate representations

- Several forms:
  - tree - abstract syntax tree
  - graphs - control flow graph
  - linear program - 3 address code

- Often times the program is represented as a hybrid
  - graphs where nodes are a linear program
  - linear program where expressions are ASTs

- Progression:
  - start close to a parse tree
  - move closer to an ISA

Our first IR: abstract syntax tree

- One step away from parse trees
- Great representation for expressions
- Natural representation to apply type checking

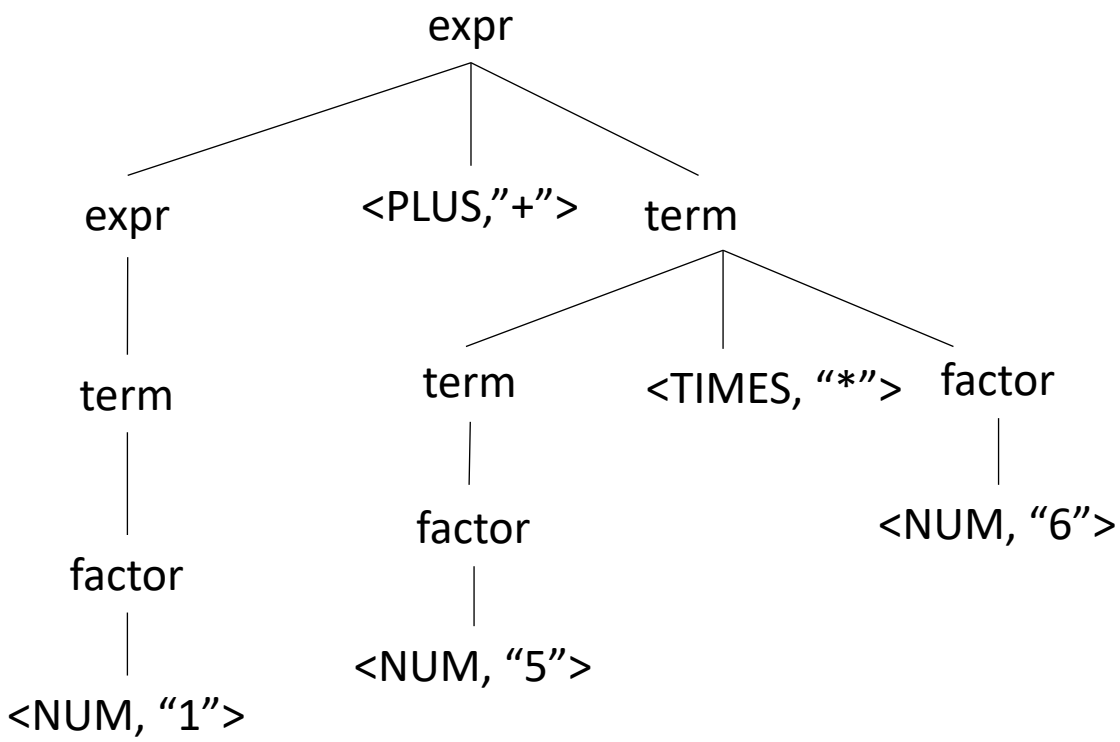
What is an AST?

Parse trees are defined entirely by the grammar

- **Tokens**
- **Production rules**

Parse trees are often not explicitly constructed. We use them to visualize the parsing computation

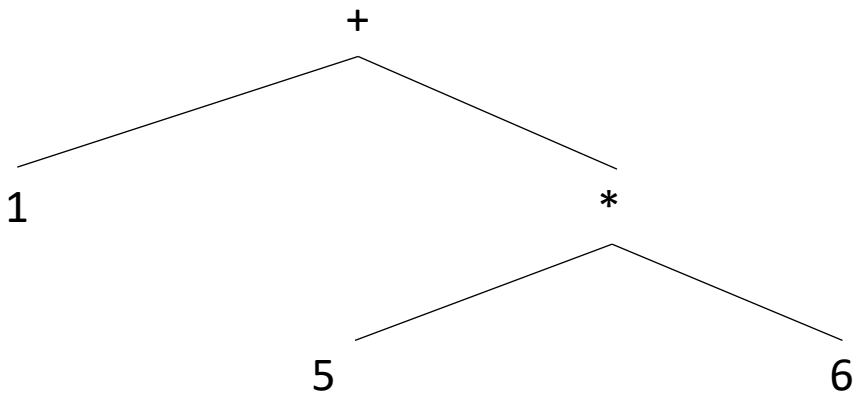
`input: 1+5*6`



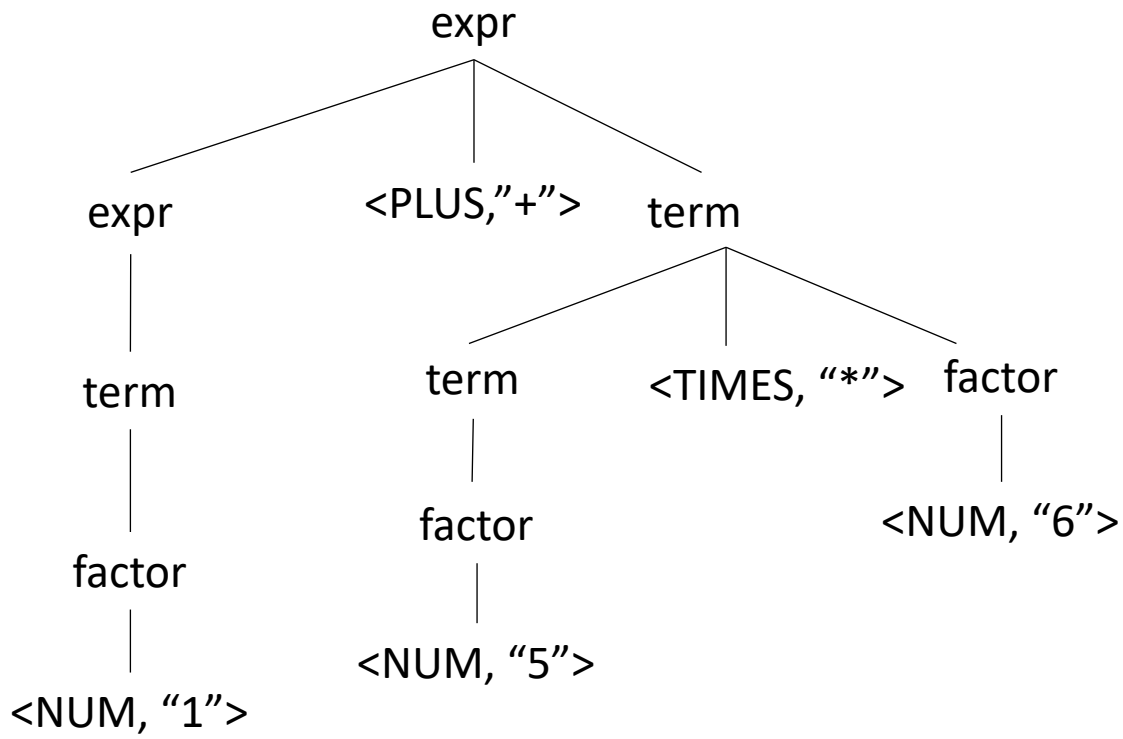
`input: 1+5*6`

What is an AST?

What are some differences?



AST

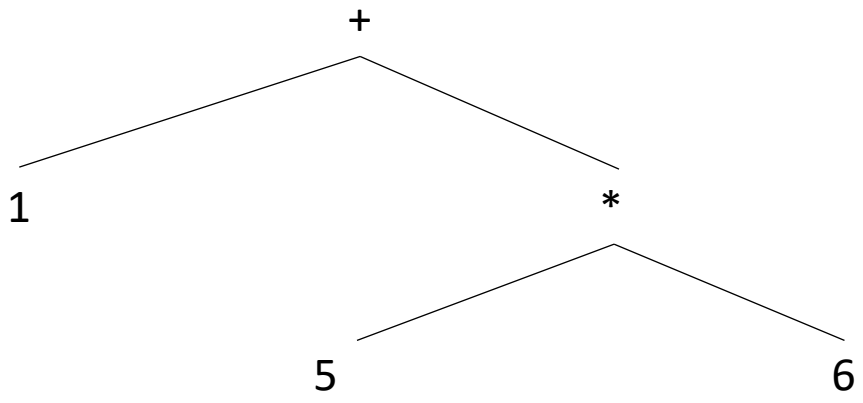


Parse Tree

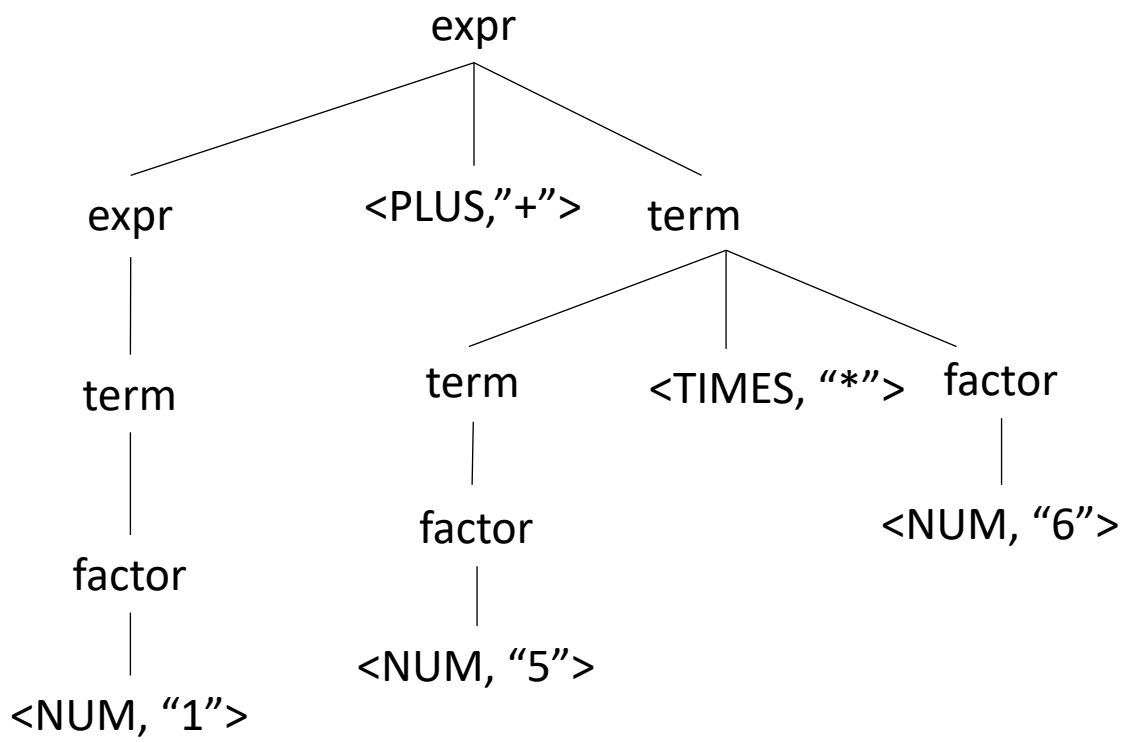
input: 1+5*6

What is an AST?

What are some differences?



AST



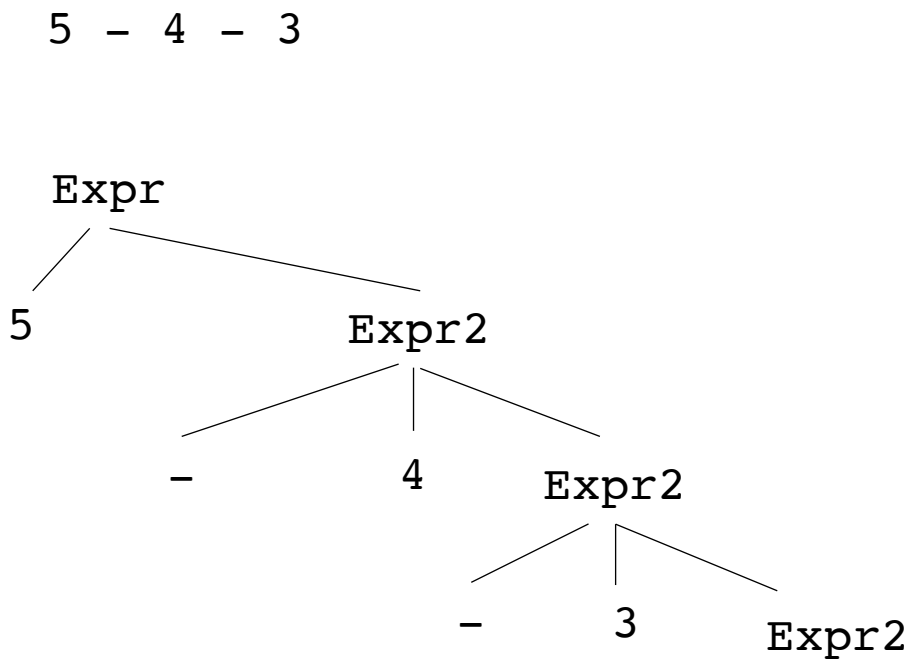
- decoupled from the grammar
- leaves are data, not lexemes
- nodes are operators, not non-terminals

Parse Tree

Creating an AST from predictive grammar

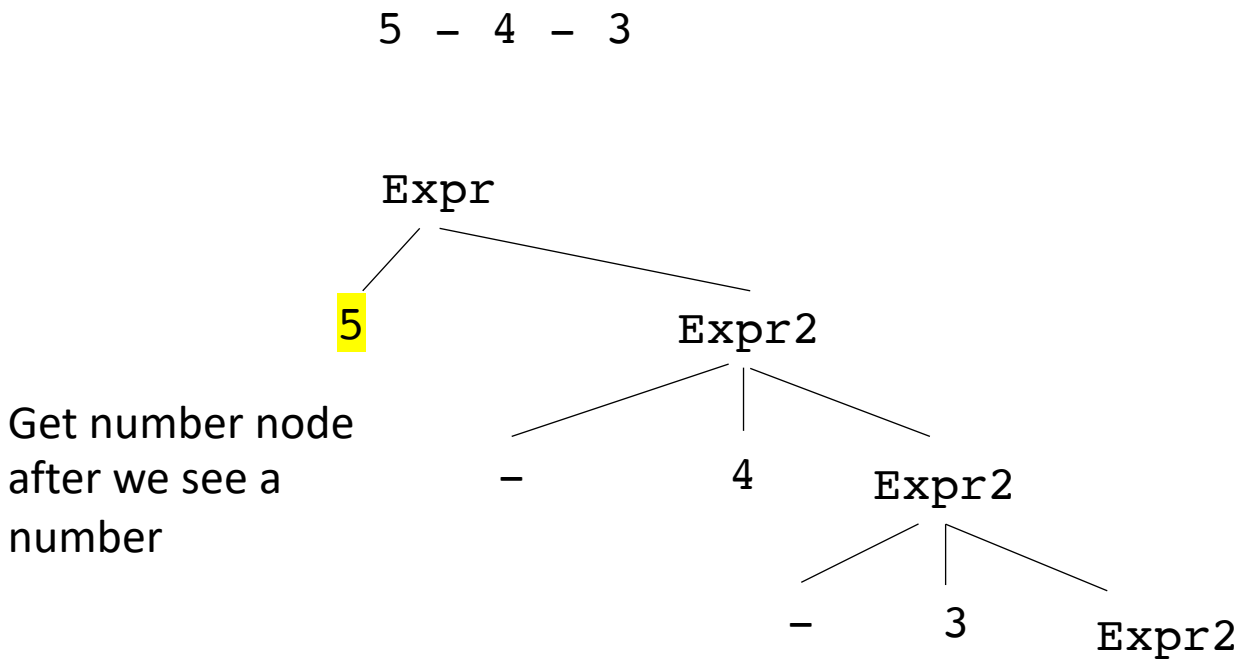
```
Expr  ::= NUM Expr2
Expr2 ::= MINUS NUM Expr2
        |   “”
```

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



Creating an AST from predictive grammar

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

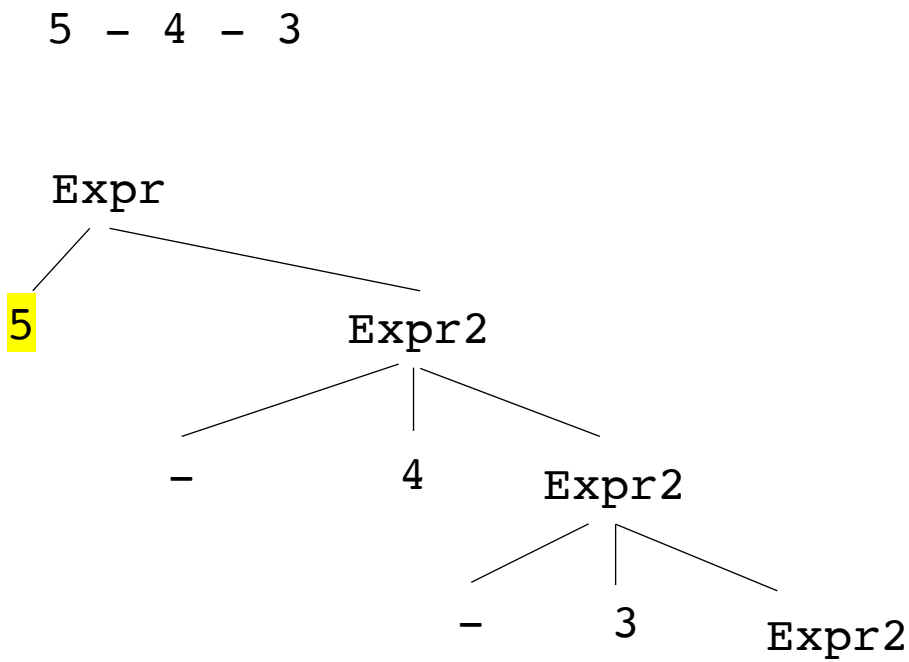


Get number node after we see a number

AST<5>

Creating an AST from predictive grammar

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

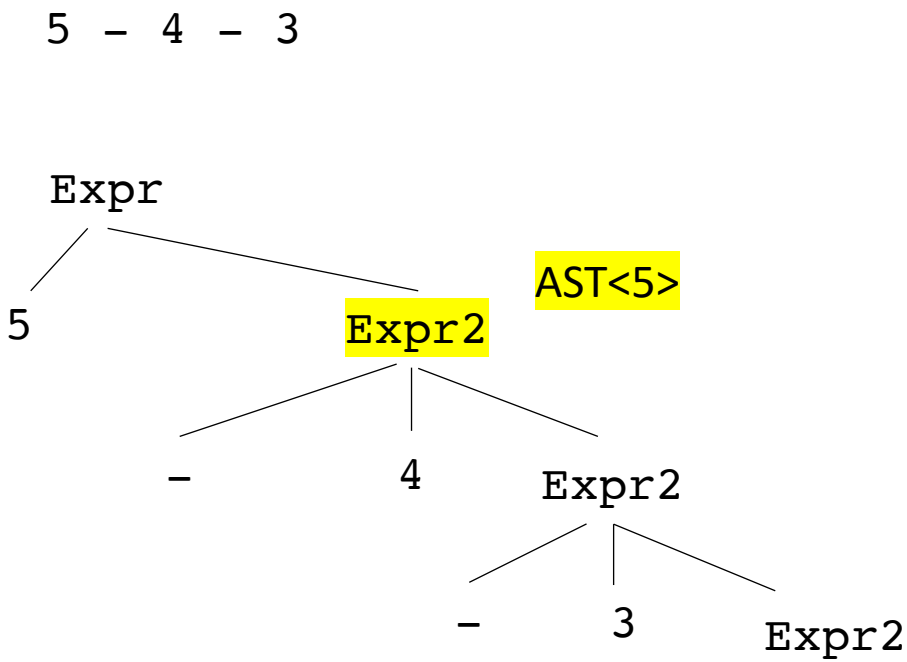


Pass the node down

AST<5>

Creating an AST from predictive grammar

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

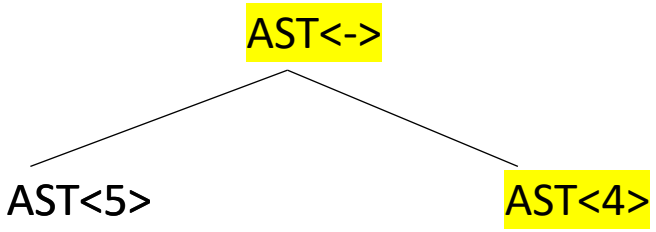


Pass the node down

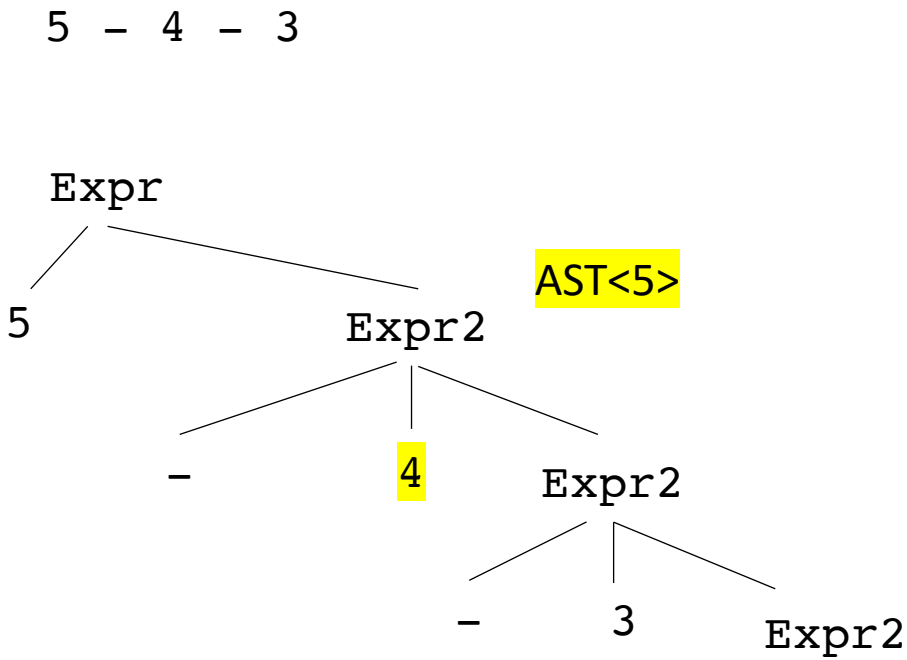
AST<5>

Creating an AST from predictive grammar

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

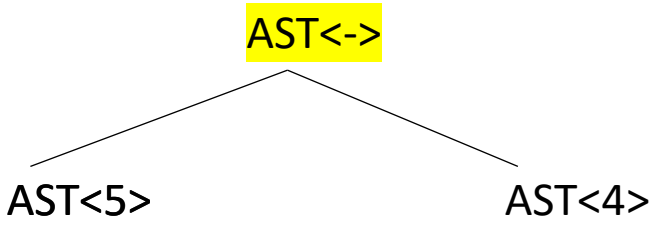


In Expr2, after 4 is parsed, create a number node and a minus node

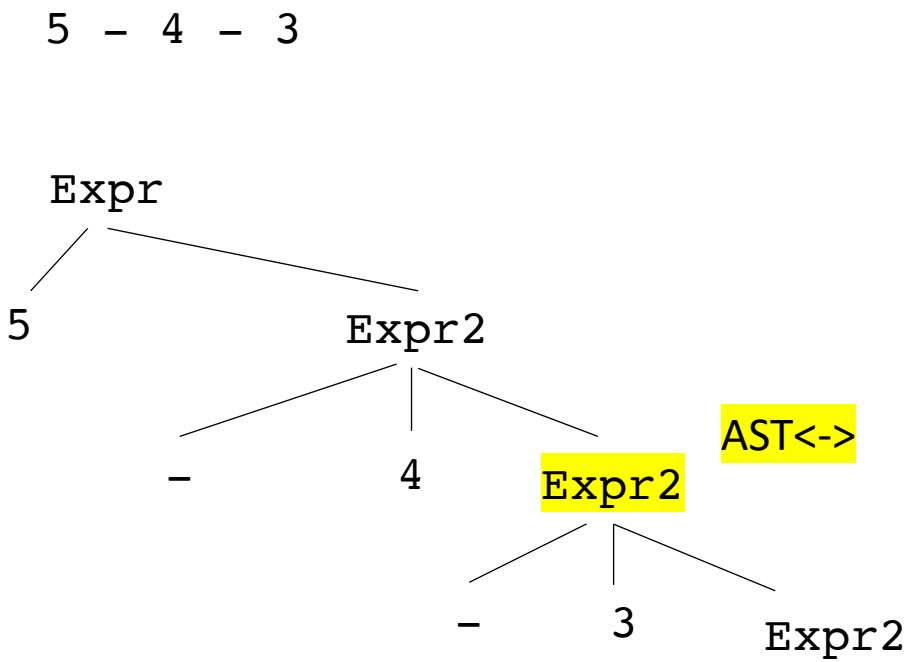


Creating an AST from predictive grammar

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

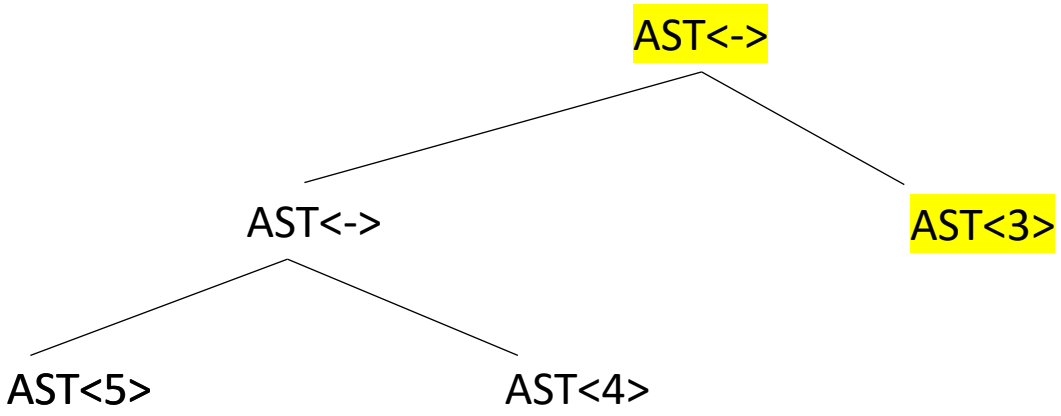


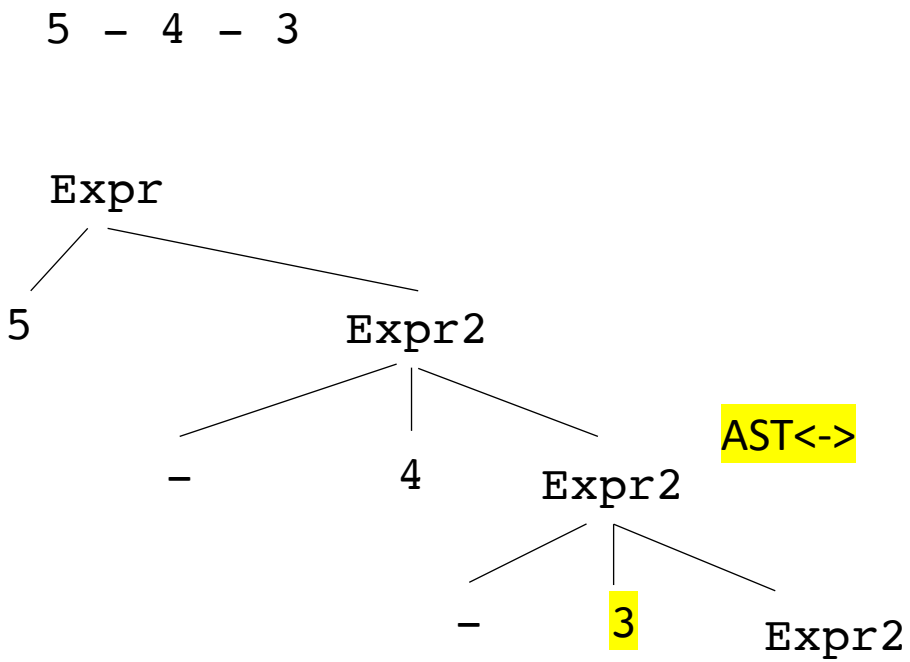
pass the new node down



Creating an AST from predictive grammar

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

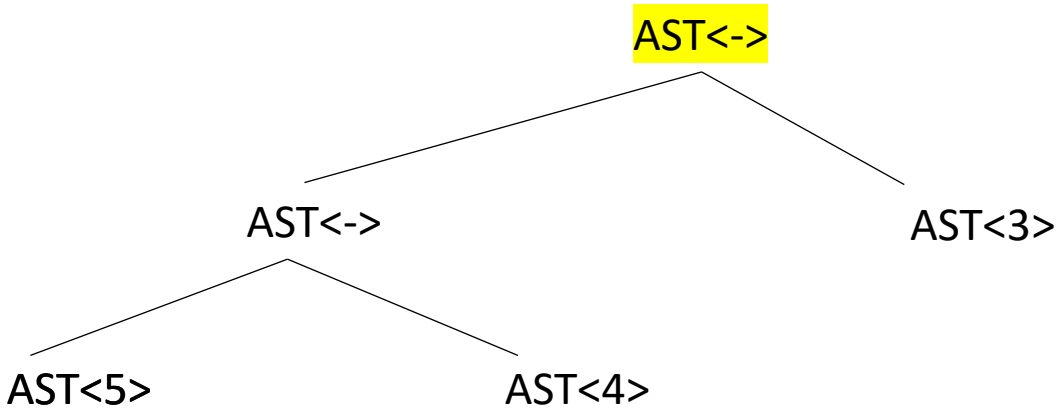




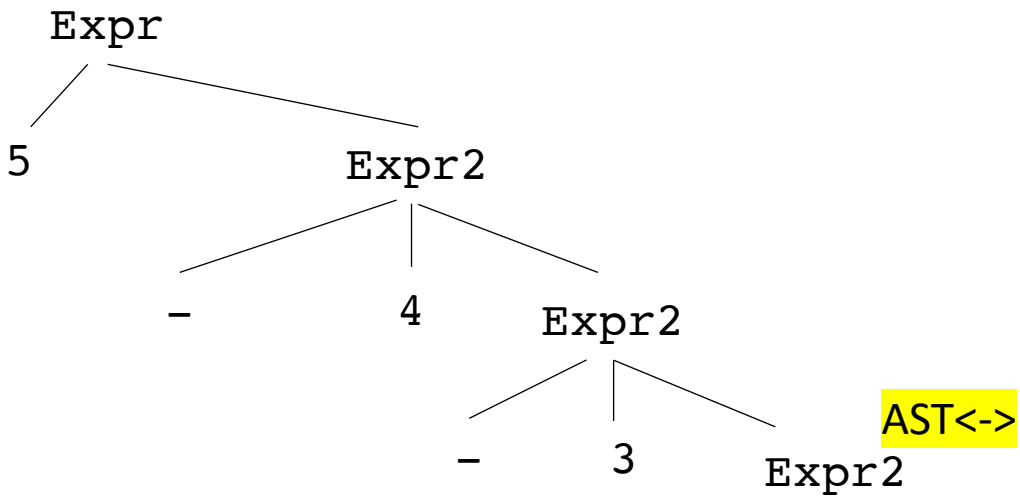
In Expr2, after 3 is parsed, create a number node and a minus node

Creating an AST from predictive grammar

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



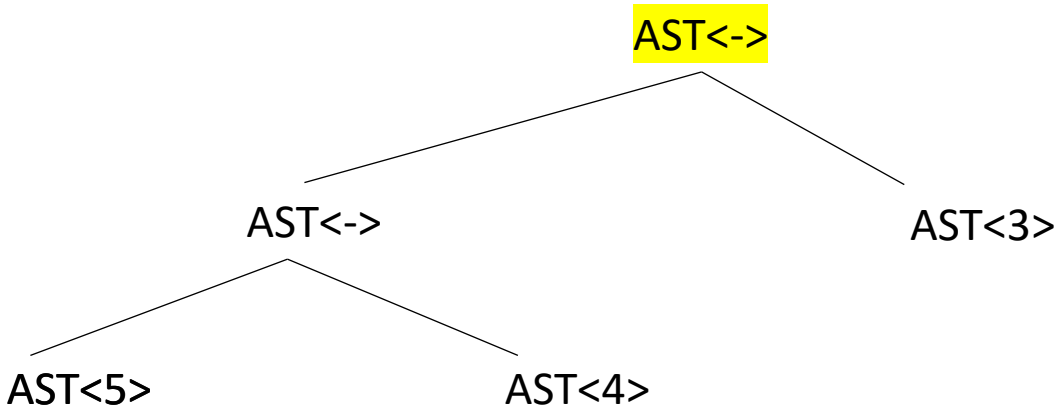
5 – 4 – 3

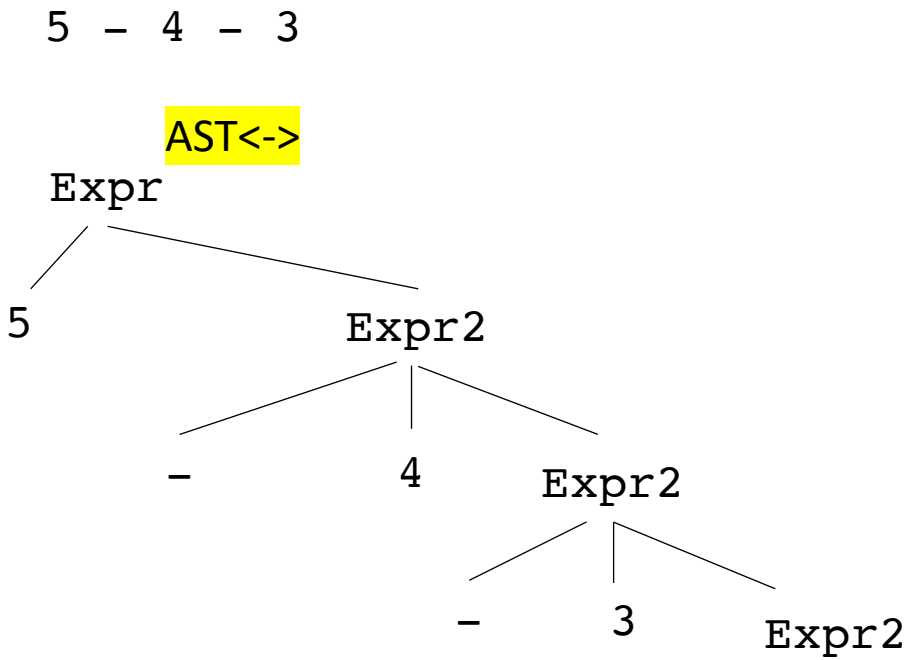


pass down the new node

Creating an AST from predictive grammar

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





return the node when there is nothing left to parse

Creating an AST from predictive grammar

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

```
def parse_expr(self):
    #lexemes second field is the value
    value = self.next_word.value
    node = ASTNumNode(value)
    self.eat(Token.NUM)
    return self.parse_expr2(node)
```

Creating an AST from predictive grammar

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

```
def parse_expr(self):
    #lexemes second field is the value
    value = self.next_word.value
    node = ASTNumNode(value)
    self.eat(Token.NUM)
    return self.parse_expr2(node)
```

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

Creating an AST from predictive grammar

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

```
def parse_expr(self):
    #lexemes second field is the value
    value = self.next_word.value
    node = ASTNumNode(value)
    self.eat(Token.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 predictive 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):
    #lexemes second field is the value
    value = self.next_word.value
    node = ASTNumNode(value)
    self.eat(Token.NUM)
    return self.parse_expr2(node)
```

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

Creating an AST from predictive 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(Token.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.

Example

- Python AST

```
import ast

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

Example

- Python AST

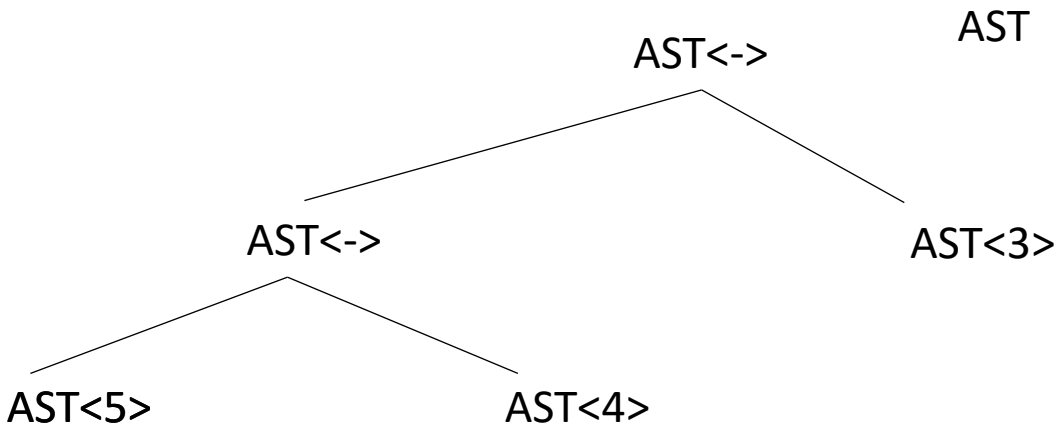
```
import ast

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

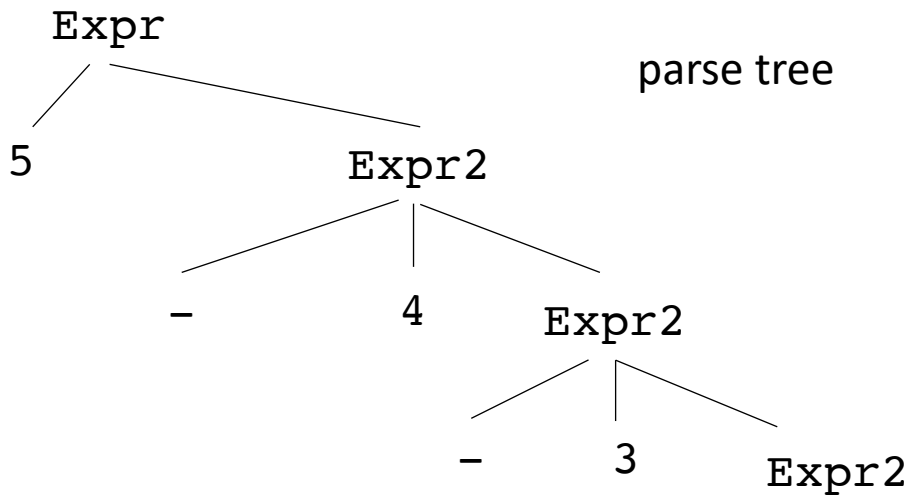
```
Expr(value=BinOp(left=BinOp(left=Num(n=5), op=Sub(), right=Num(n=4)), op=Sub(), right=Num(n=2)))
```

Evaluate an AST by doing a post order traversal

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



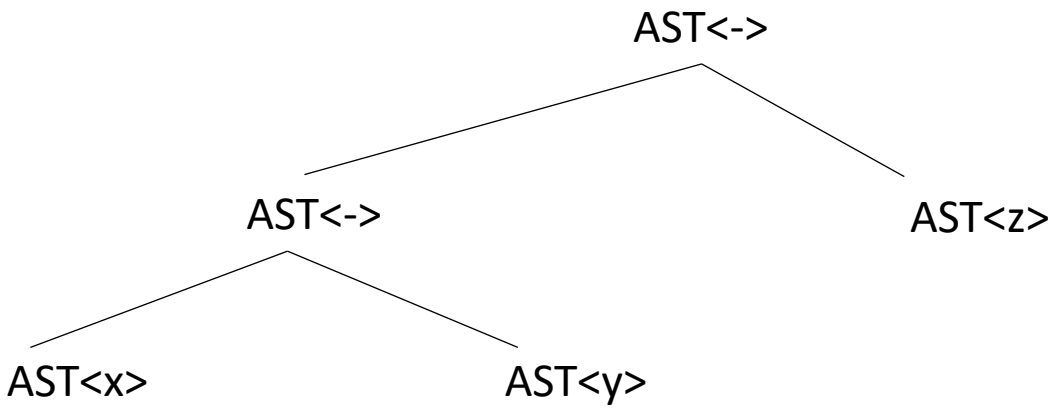
5 – 4 – 3



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

Evaluate an AST by doing a post order traversal

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

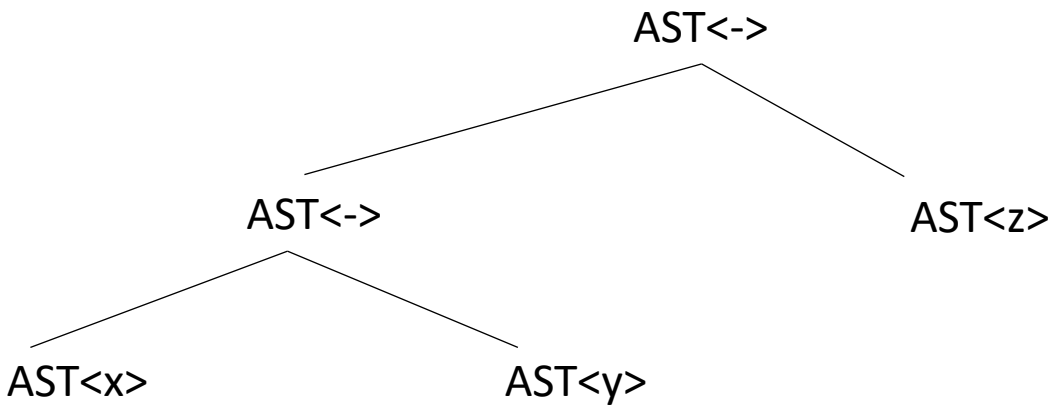


What if you cannot evaluate it?
What else might you do?

```
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
```

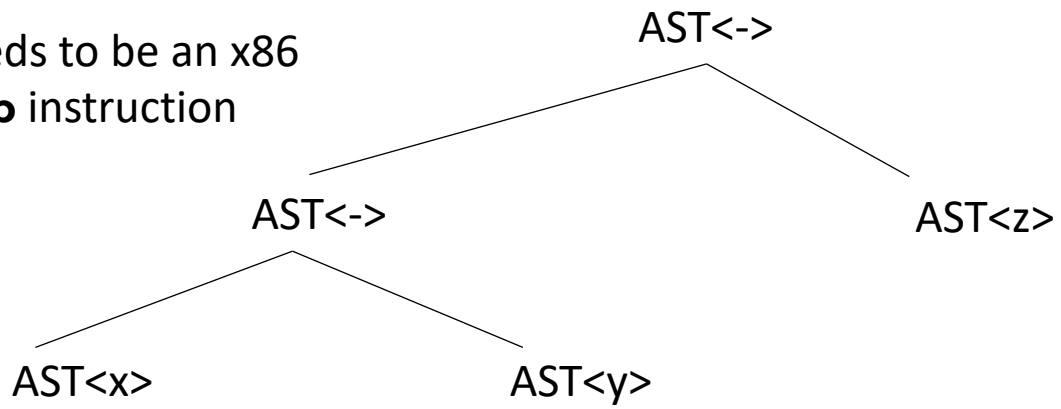
How does this change things?

Evaluate an AST by doing a post order traversal

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

needs to be an x86 **subss** instruction

needs to be an x86 **sub** instruction



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?

Is this all?

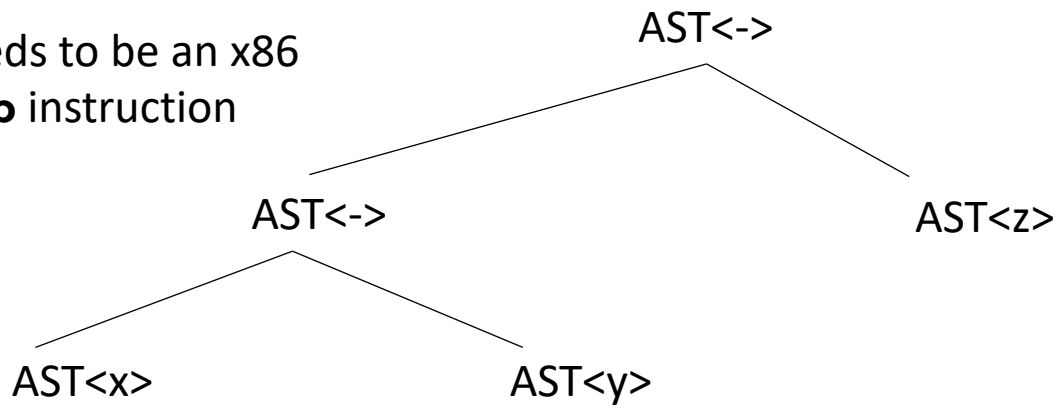
Evaluate an AST by doing a post order traversal

```
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?

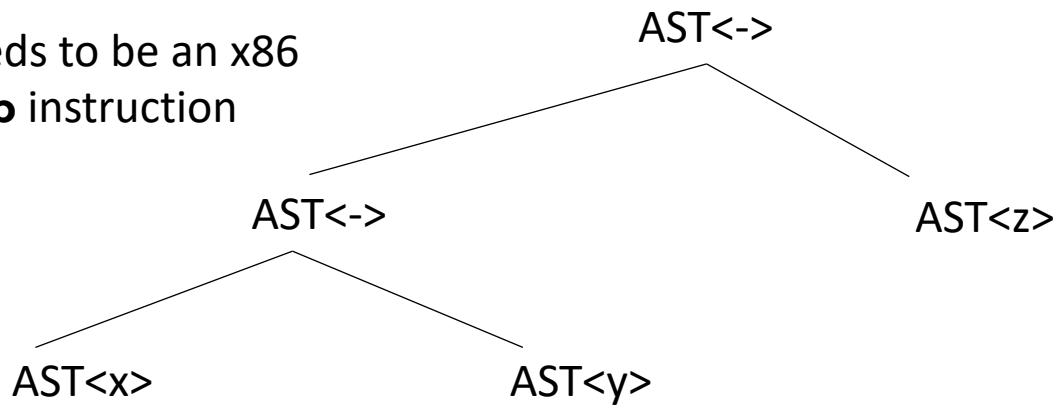
Evaluate an AST by doing a post order traversal

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

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

needs to be an x86 **subss** instruction

needs to be an x86 **sub** instruction



Is this all?

Lets do some experiments.

What should 5 - 5.0 be?

but

subss r1 r2

interprets both registers as floats

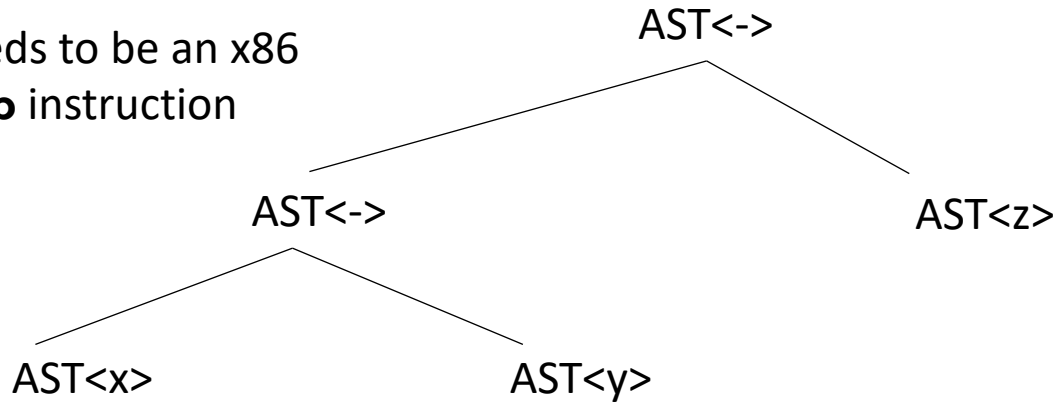
Evaluate an AST by doing a post order traversal

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

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

needs to be an x86 **subss** instruction

needs to be an x86 **sub** 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?

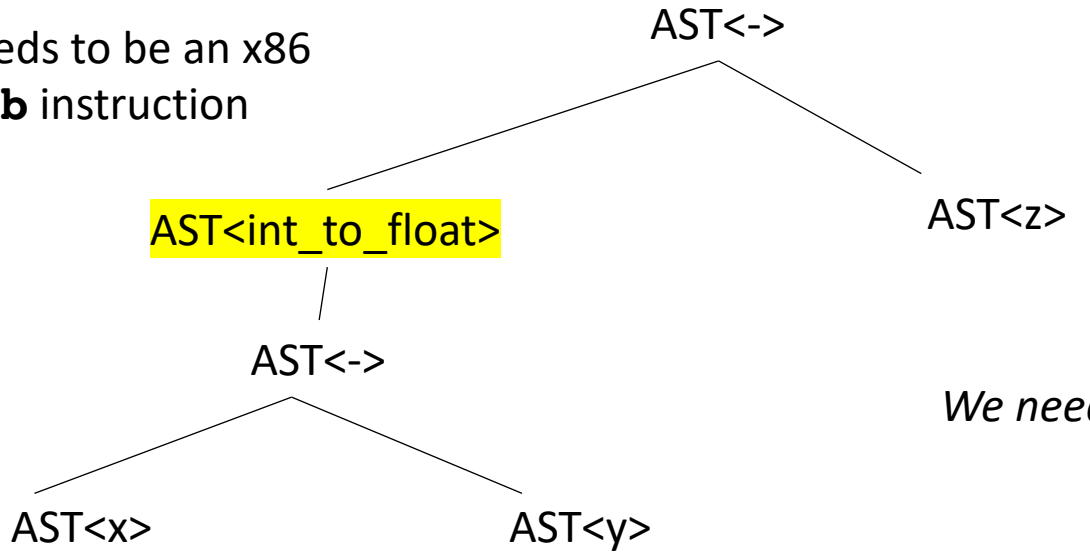
Evaluate an AST by doing a post order traversal

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

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

needs to be an x86 **subss** instruction

needs to be an x86 **sub** 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

- 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?
- In this class, we will be:
  - Compiling a statically typed language (similar to C)
  - into an untyped language (similar to an ISA)
  - using a dynamically typed language (python)

Type systems

Considerations:

- common base types in a language:
  - ints
  - chars
  - strings
  - floats
  - bool

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

Type systems

Considerations:

- common base types in a language:
  - ints
  - chars
  - strings
  - floats
  - bool

size of ints?
How does C do it?
How does Python do it?
Pros and cons?

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

Type systems

Considerations:

- common base types in a language:
  - ints
  - chars
  - strings
  - floats
  - bool

Are strings a base type? In C? In Python?

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

Type systems

Considerations:

- common base types in a language:
  - ints
  - chars
  - strings
  - floats
  - bool

How are bools handled? in C? in Python

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

Type systems

Considerations:

- common base types in a language:
  - ints
  - chars
  - strings
  - floats
  - bool

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

Type systems

Considerations:

- common base types in a language:
  - ints
  - chars
  - strings
  - floats
  - bool

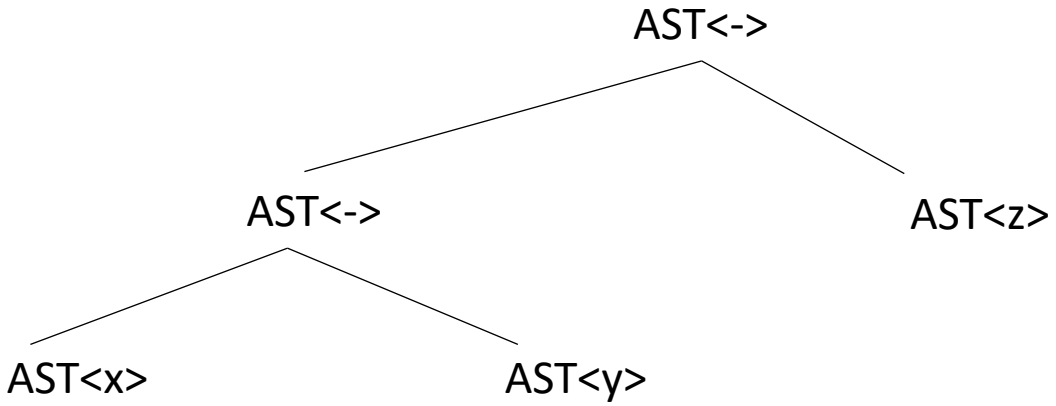
- How to combine types in expressions:
  - int and float?
  - int and char?
  - int and bool?

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

Type checking on an AST

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

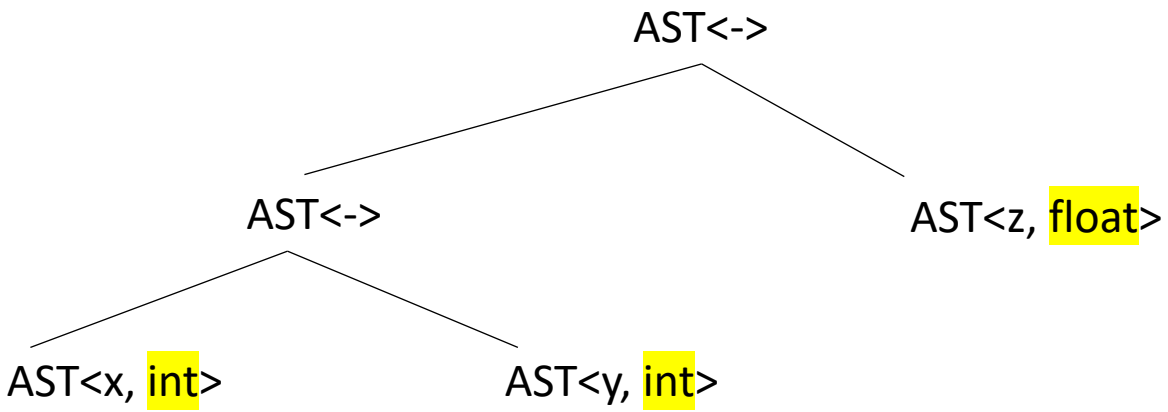
each node additionally gets a type



Type checking on an AST

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

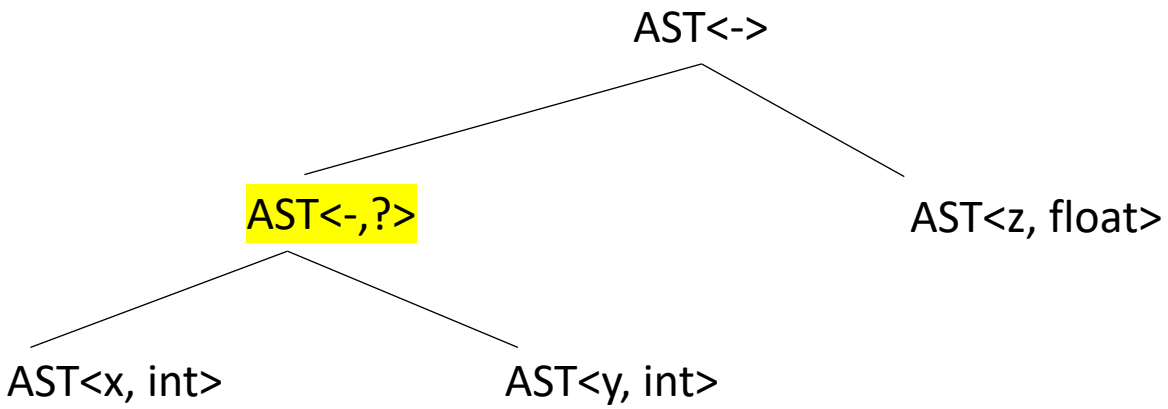
each node additionally gets a type
we can get this from the symbol table for the leaves



Type checking on an AST

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

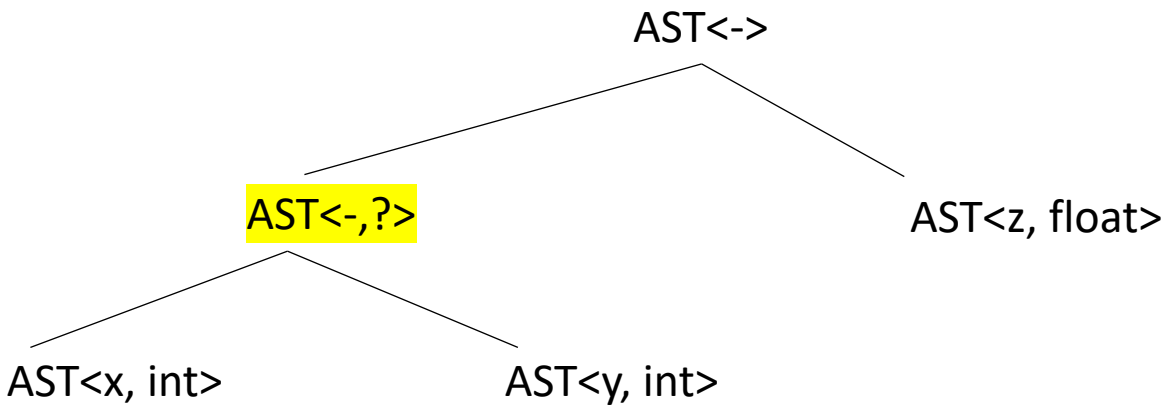
How do we get the type for this one?



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?



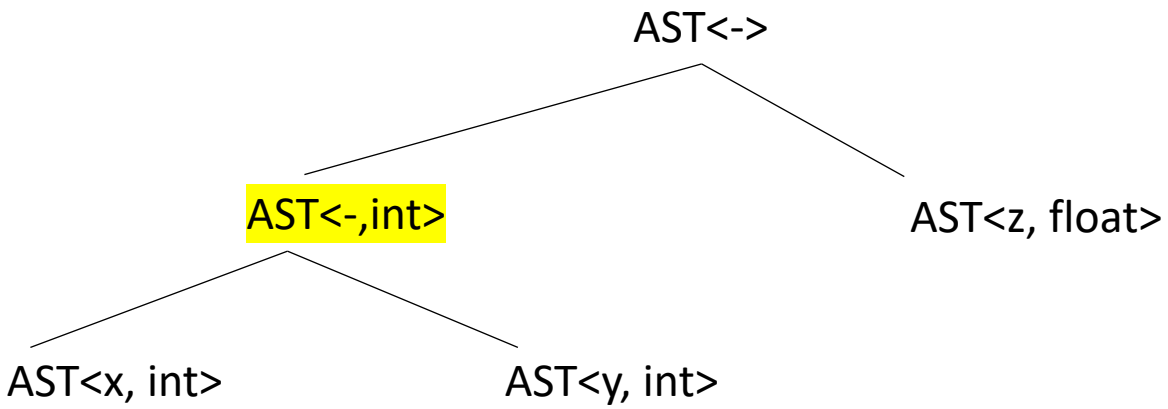
combination rules for subtraction:

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?



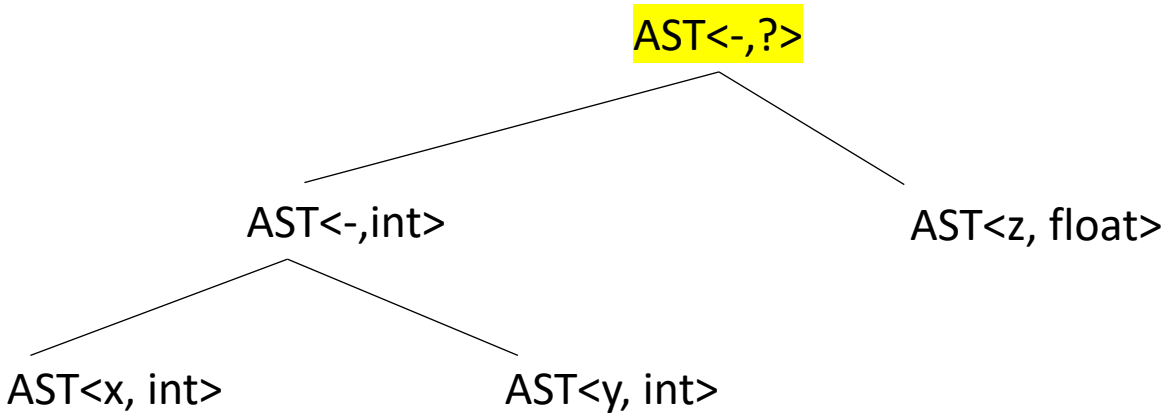
inference rules for subtraction:

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?



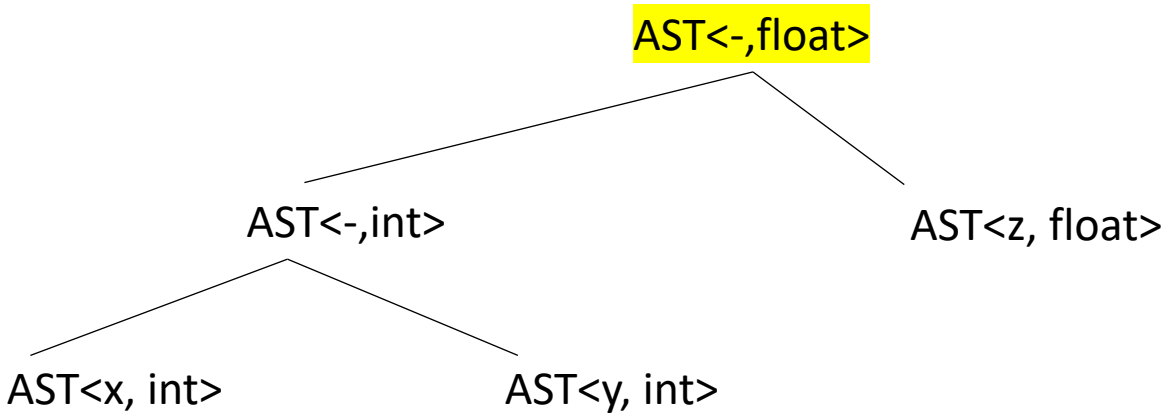
inference rules for subtraction:

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?



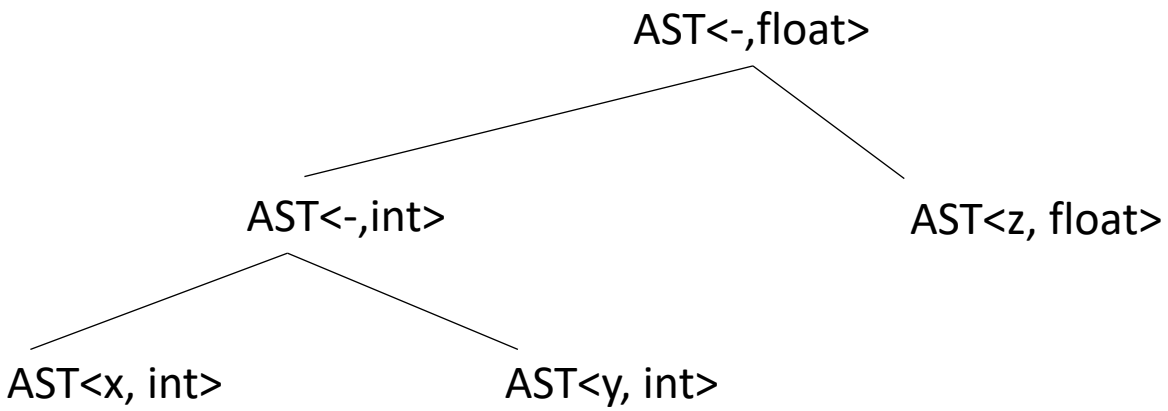
inference rules for subtraction:

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 subtraction:

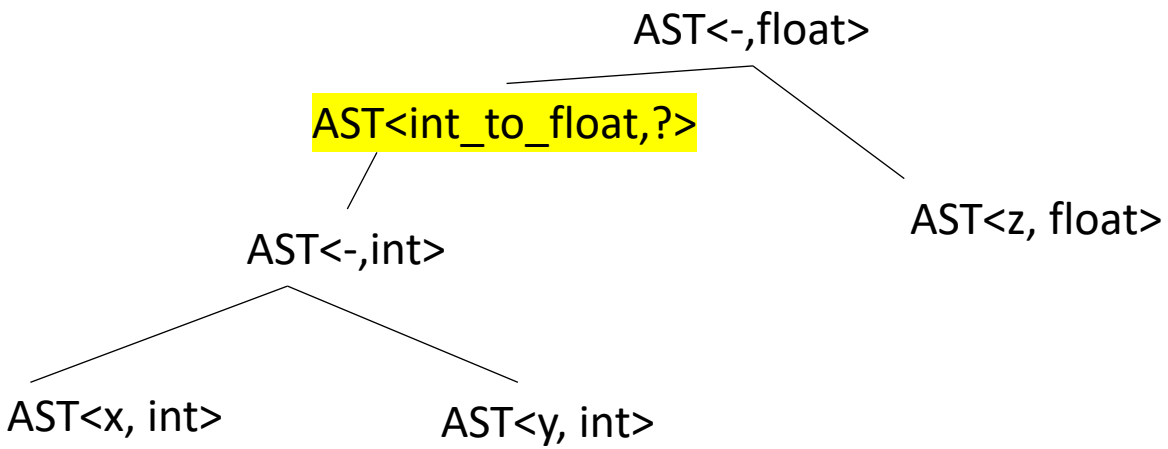
first	second	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

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 subtraction:

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 previous lectures 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)
}
```

in previous lectures we didn’t record any information in the symbol table

record the type in the symbol table

Enum for types

```
from enum import Enum

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

Our base AST Node needs a type

```
class ASTNode():
    def __init__(self):
        self.node_type = None
        pass

    def set_type(self, t):
        self.node_type = t

    def get_type(self):
        return self.node_type
```

Now we need to set the types for the leaf nodes

```
class ASTNumNode(ASTLeafNode):
    def __init__(self, value):
        super().__init__(value)
        if is_int(value):
            self.set_type(Types.INT)
        else:
            self.set_type(Types.FLOAT)
```

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

Where can we get the value type for an ID?

But that doesn't get us here yet...

add the type at parse time

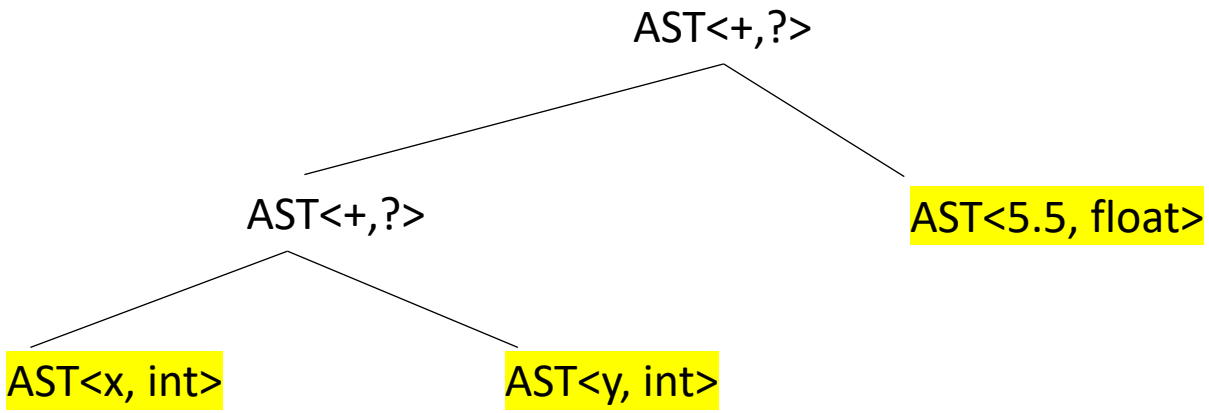
```
Unit ::= ID
       |   NUM
```

```python
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

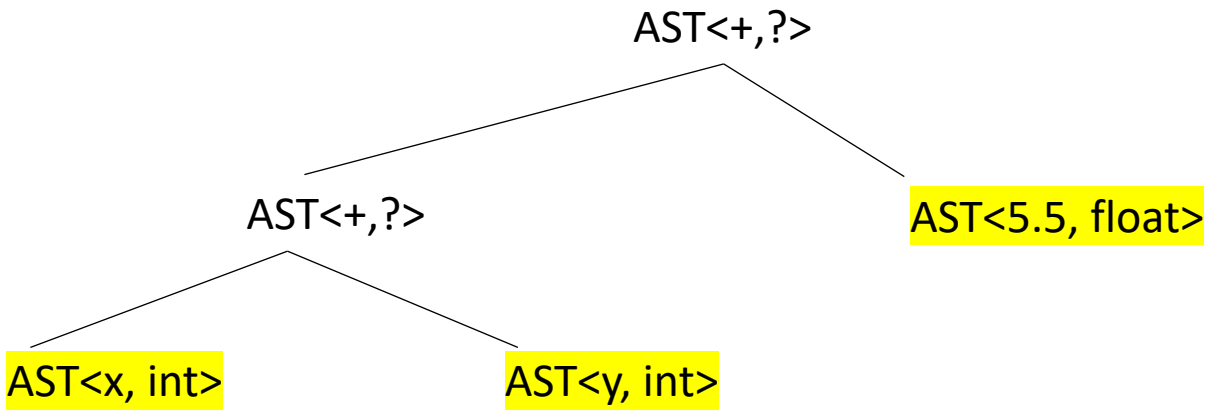
- We now have the types for the leaf nodes

```
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):

  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
```

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

we need to make sure the children have types!

inference rules for plus

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

Type inference

```
def type_inference(n):

  case split on n:

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

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

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

we should record our type

inference rules for plus

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

Type inference

```
def type_inference(n):

  case split on n:

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

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

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

is this just for plus?

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

Type inference

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

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

```
case split on n:

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

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

is this just for plus?

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

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

Type inference

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

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

```
case split on n:

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

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

is this just for plus?

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

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

Type inference

```
def type_inference(n):

 case split on n:

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

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

What about for assignments?

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

What does this return?

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

Type inference

```
def type_inference(n):

 case split on n:

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

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

What about for assignments?

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

What does this return?

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

whatever the left is

Type 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):

  case split on n:

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

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

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

we should record our type

inference rules for plus

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

Type inference

```
def type_inference(n):

  case split on n:

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

  if n is a plus node:
     do type inference on children
     t = lookup type from table
     if t is None:
        throw type exception
     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
string	int	None

like in Python

See everyone on Friday!

- We will discuss implementing type inference on Monday