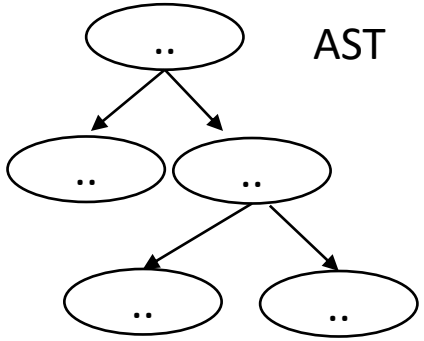
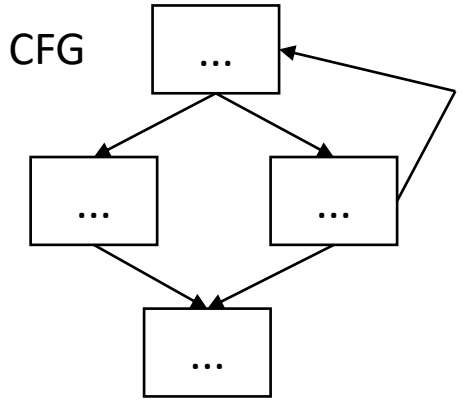
CSE110A: Compilers

April 29, 2022



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

- HW 2
  - Due on Monday by midnight
  - There is no guarantee of help on piazza off business hours and weekends
  - Neal wrote a recursive descent document you should read for extra help

- HW 3 will be assigned a week from Monday (May 9) so that you only have the midterm to do next week.

Announcements

HW2 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 3 will be creating an IR inside your parser.

Announcements

- Midterm will be given on May 2 (Monday)
  - Take home midterm.
  - Assigned on Monday morning and due on Friday by midnight
  - No late midterms are accepted - ***start early*** so that you can absorb any issues
  - No help off of business hours. Do not discuss the midterm at all with classmates, including conceptual, programming, or framework questions.
  - Open
    - book
    - notes
    - slides
    - lectures
  - You can use the internet for concepts. You cannot use it to ask questions or google answers to questions.

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?

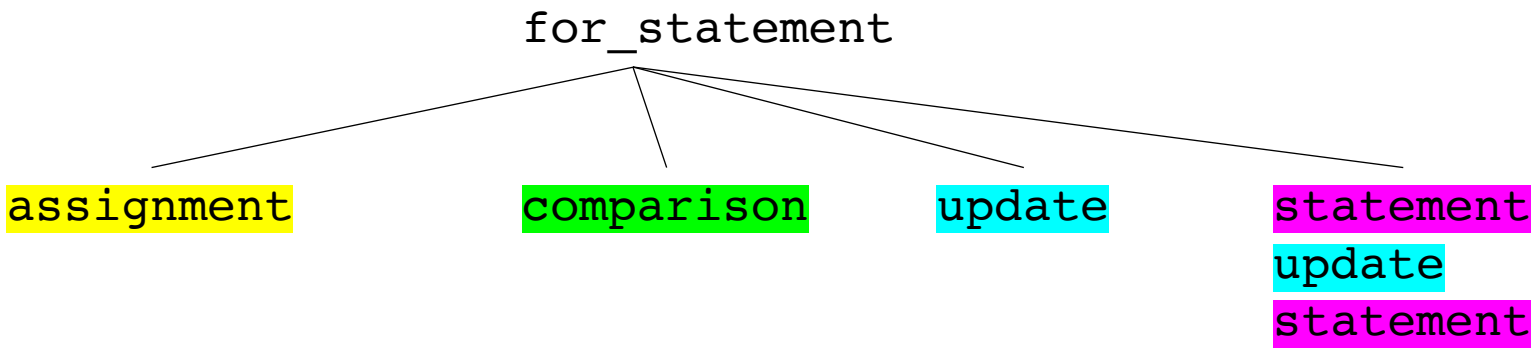
- ○ M, N
- ○ MN, M+N
- ○ M+N, MN
- ○ MN, NM
- ○ M, NM
- ○ M, N + M

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

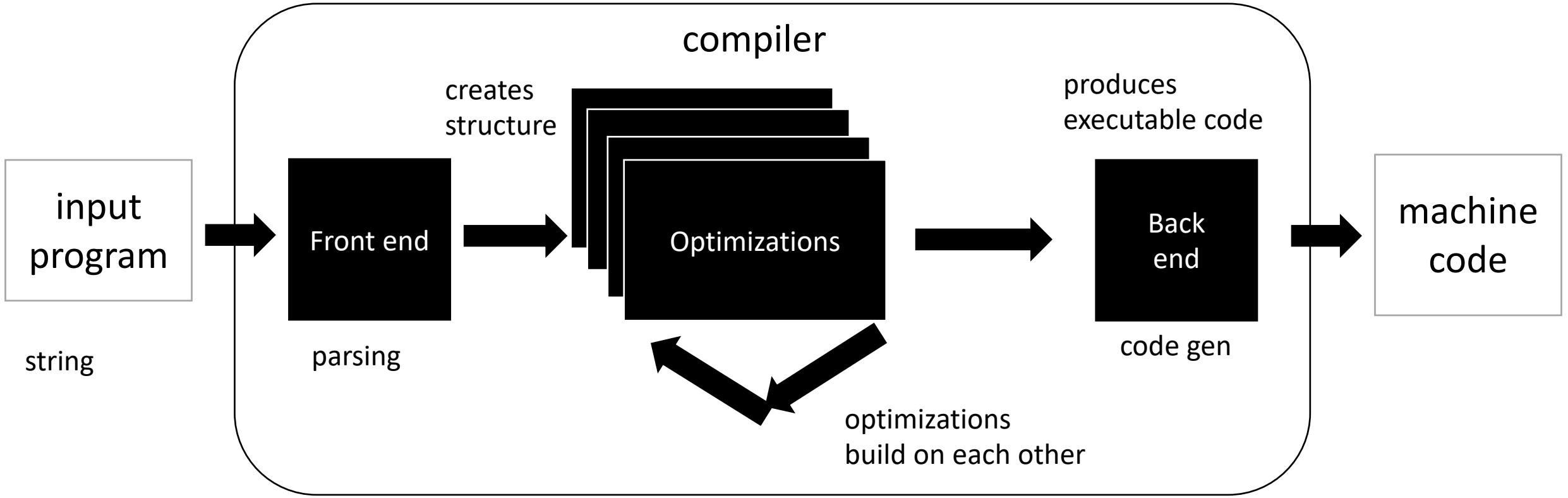
copy statement and put an update before it

Quiz

Name a few Intermediate Representations you have seen in real life

Review

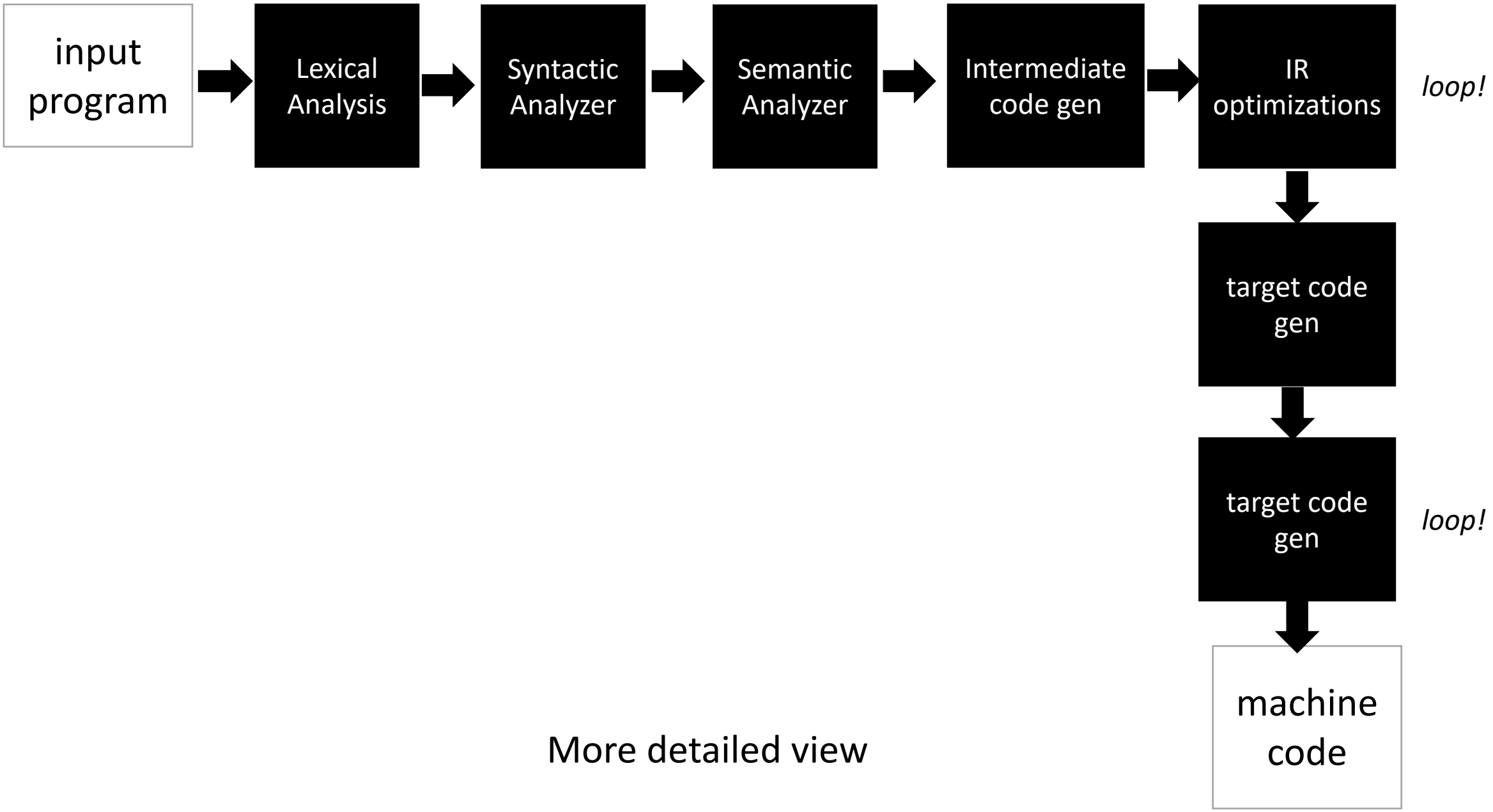
Compiler Architecture



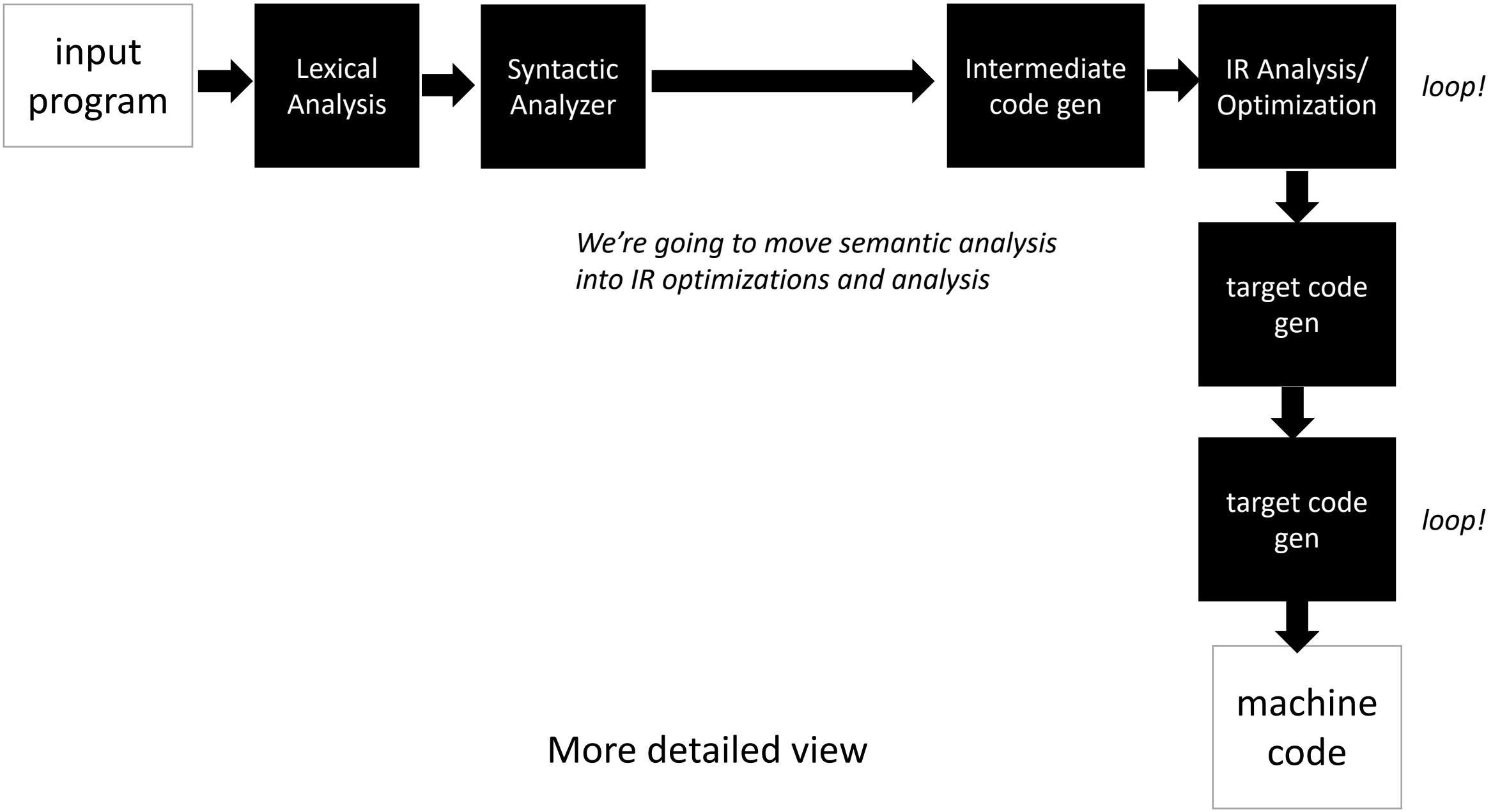
Medium detailed view

more about optimizations: https://stackoverflow.com/questions/15548023/clang-optimization-levels

More detailed view

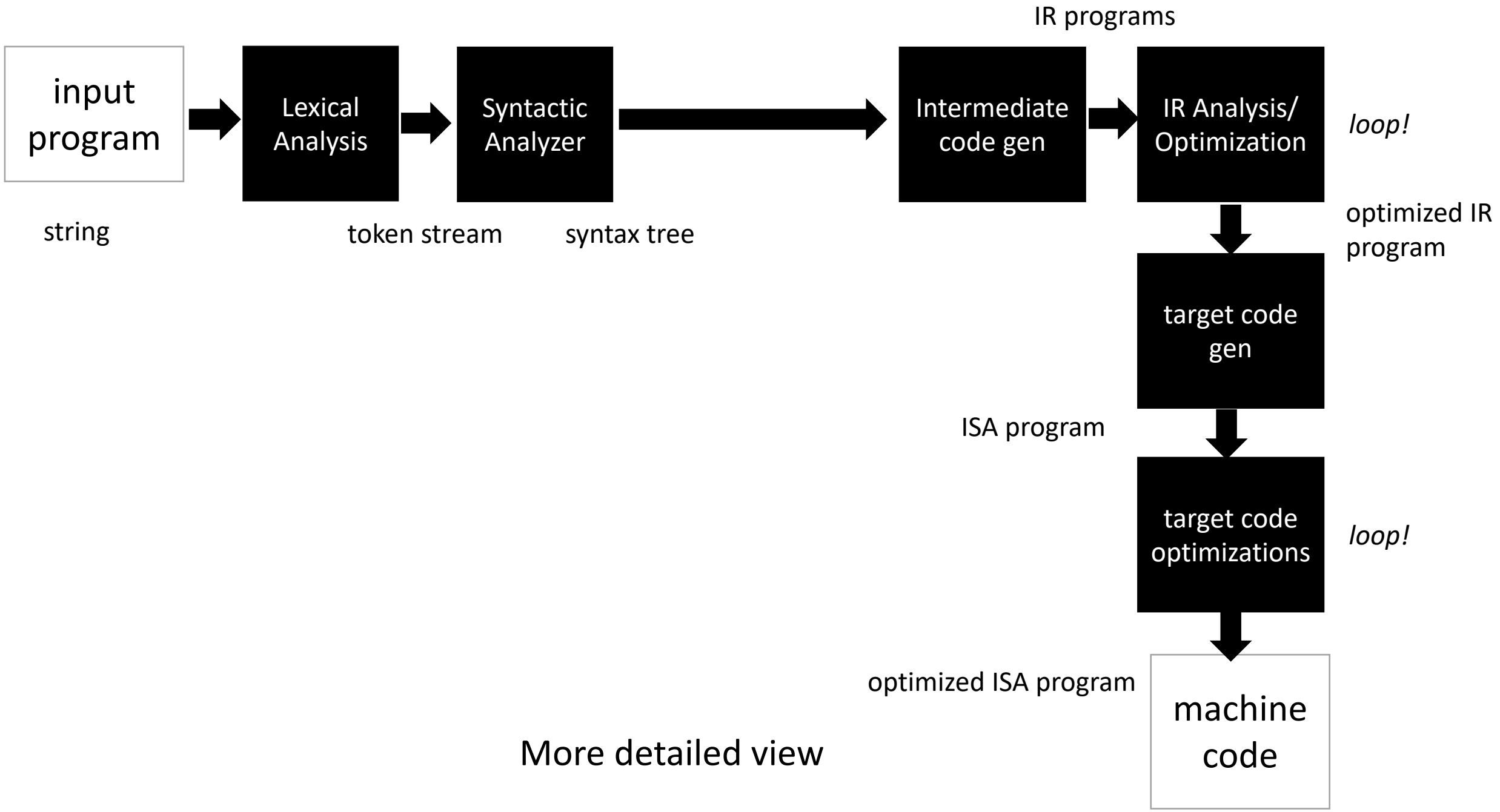


More detailed view

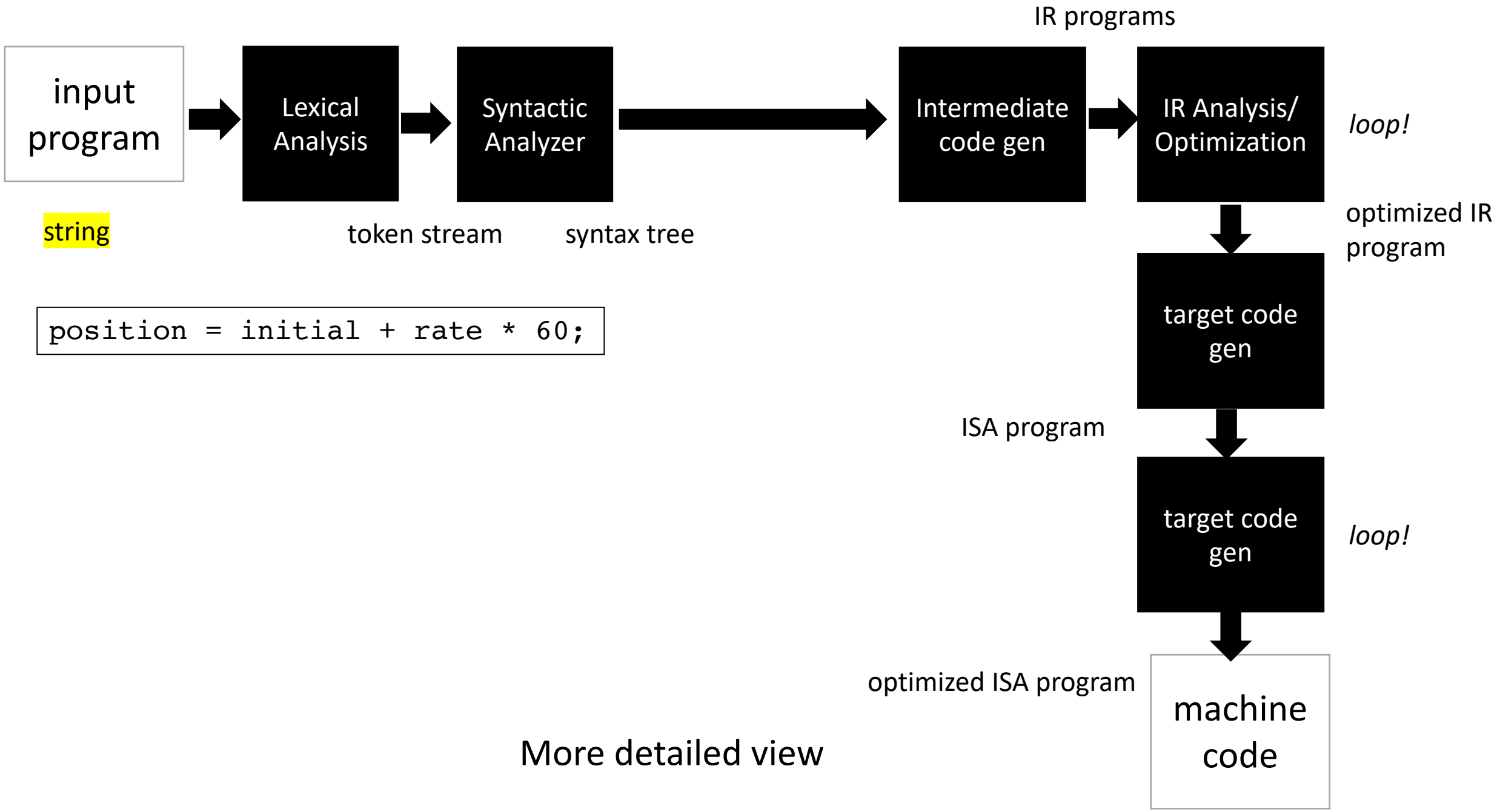


We're going to move semantic analysis into IR optimizations and analysis

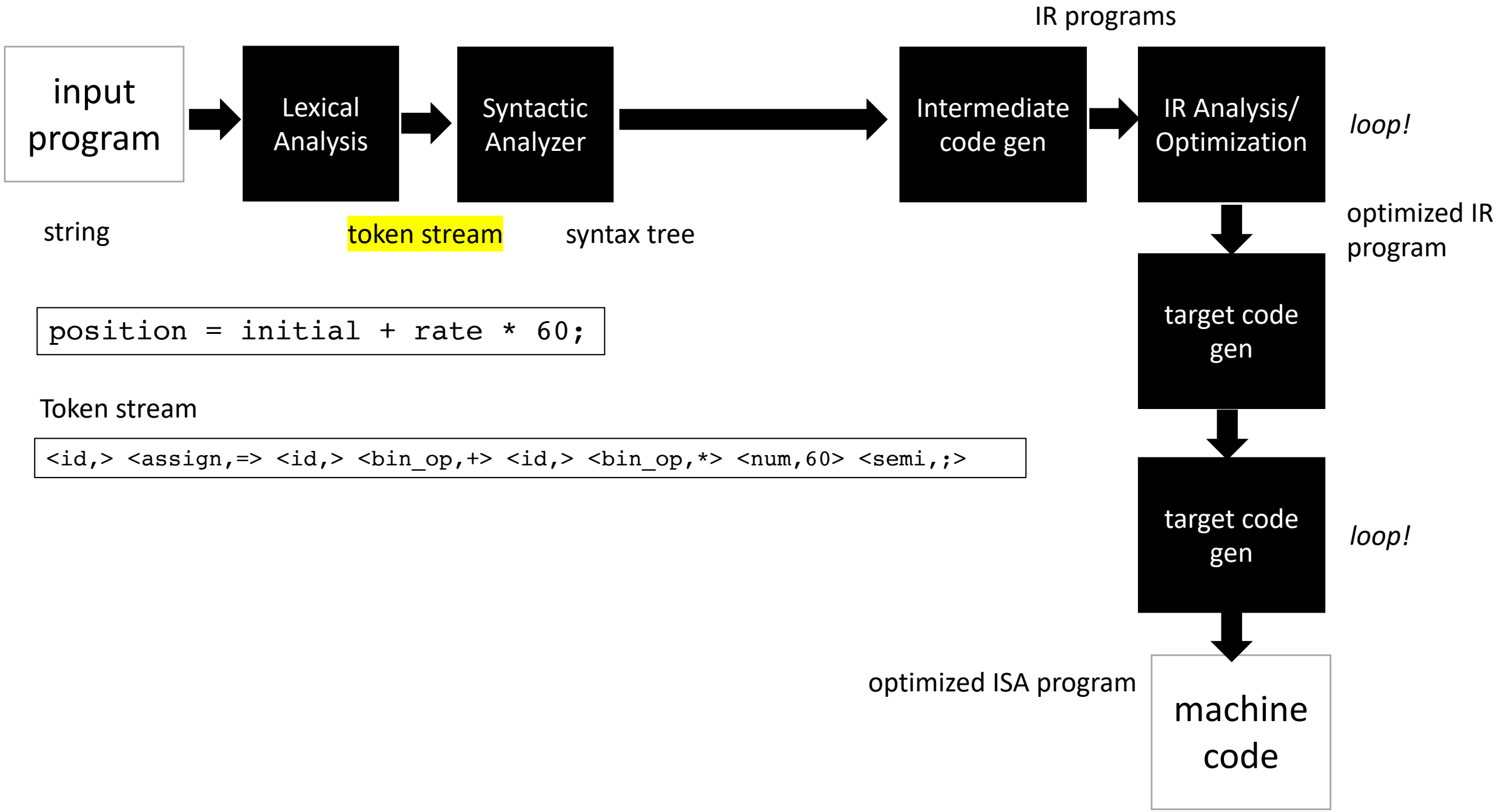
More detailed view



More detailed view



More detailed view

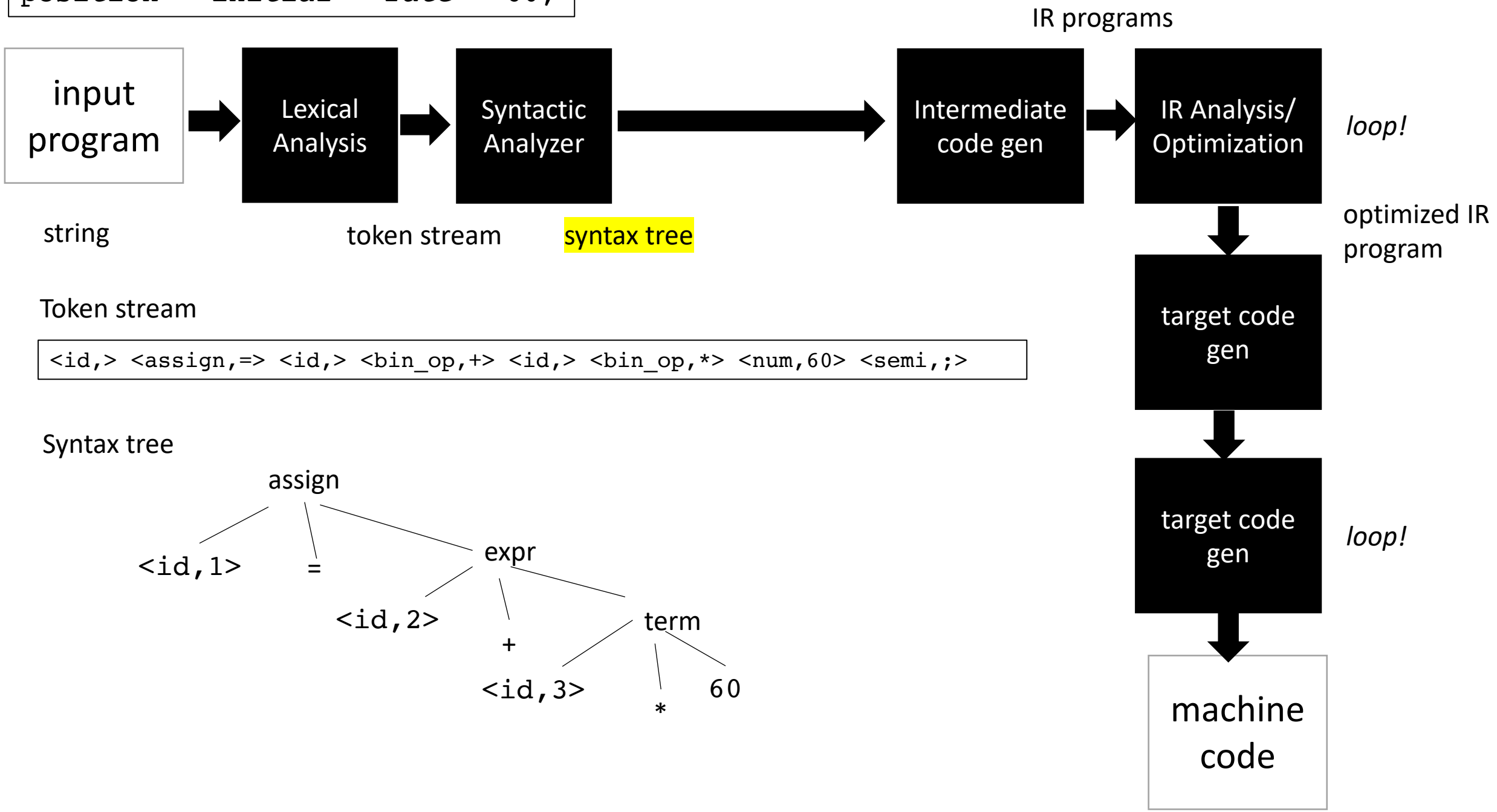



```
position = initial + rate * 60;
```

Token stream

```
<id,> <assign,=> <id,> <bin_op,+> <id,> <bin_op,*> <num,60> <semi,;>
```

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

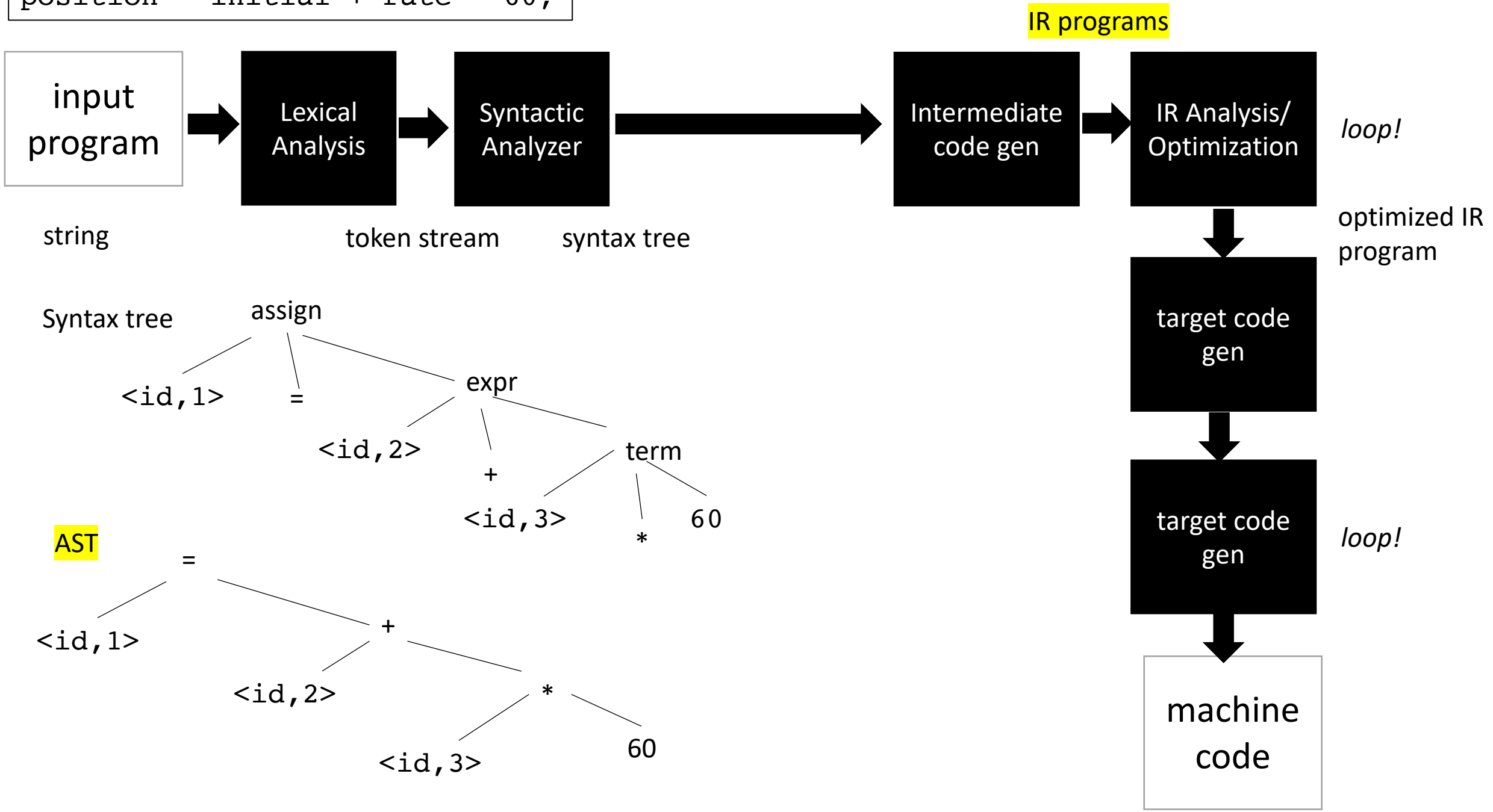


Token stream

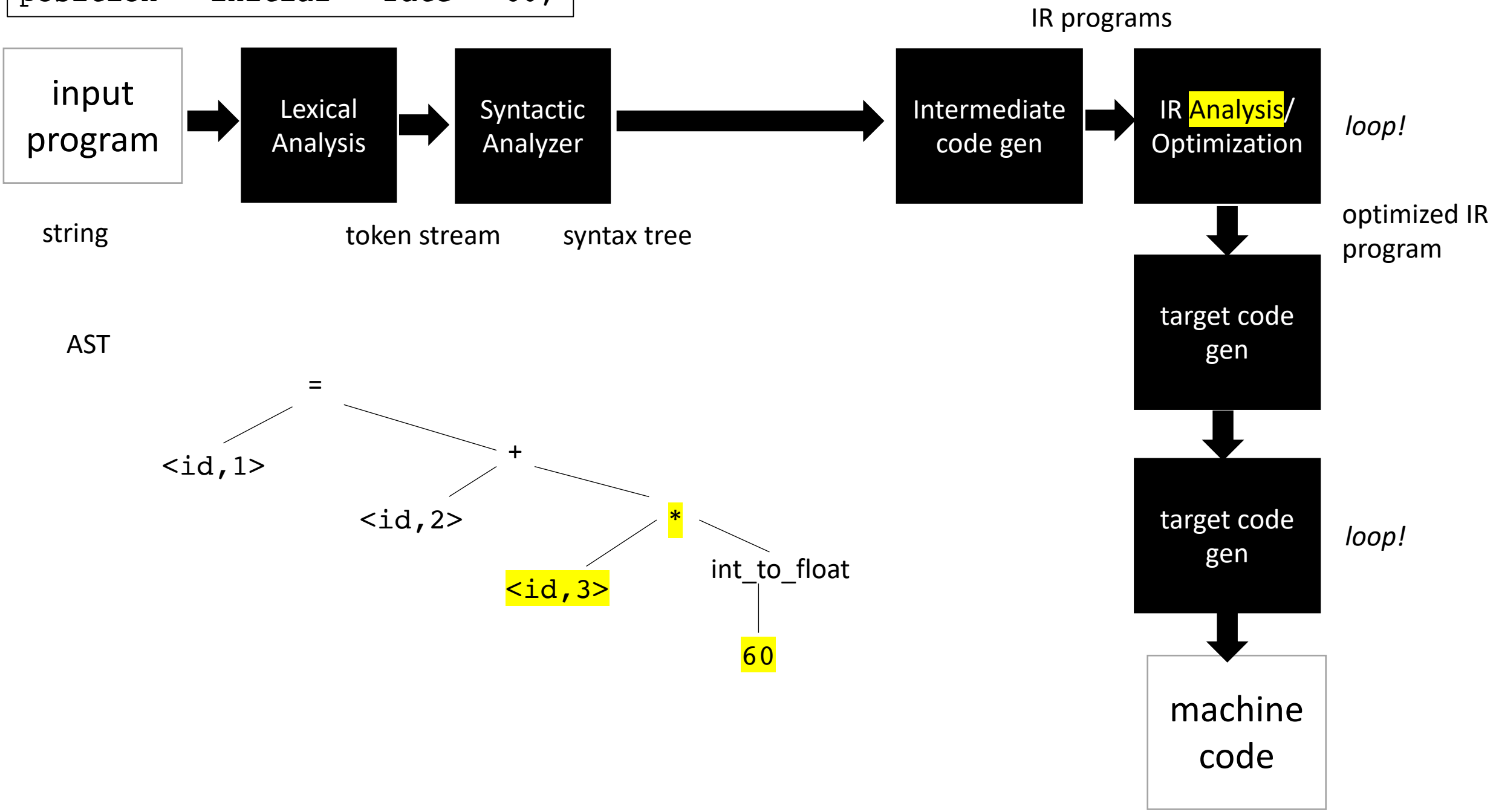
```
<id,> <assign,=> <id,> <bin_op,+> <id,> <bin_op,*> <num,60> <semi,;>
```

Syntax tree

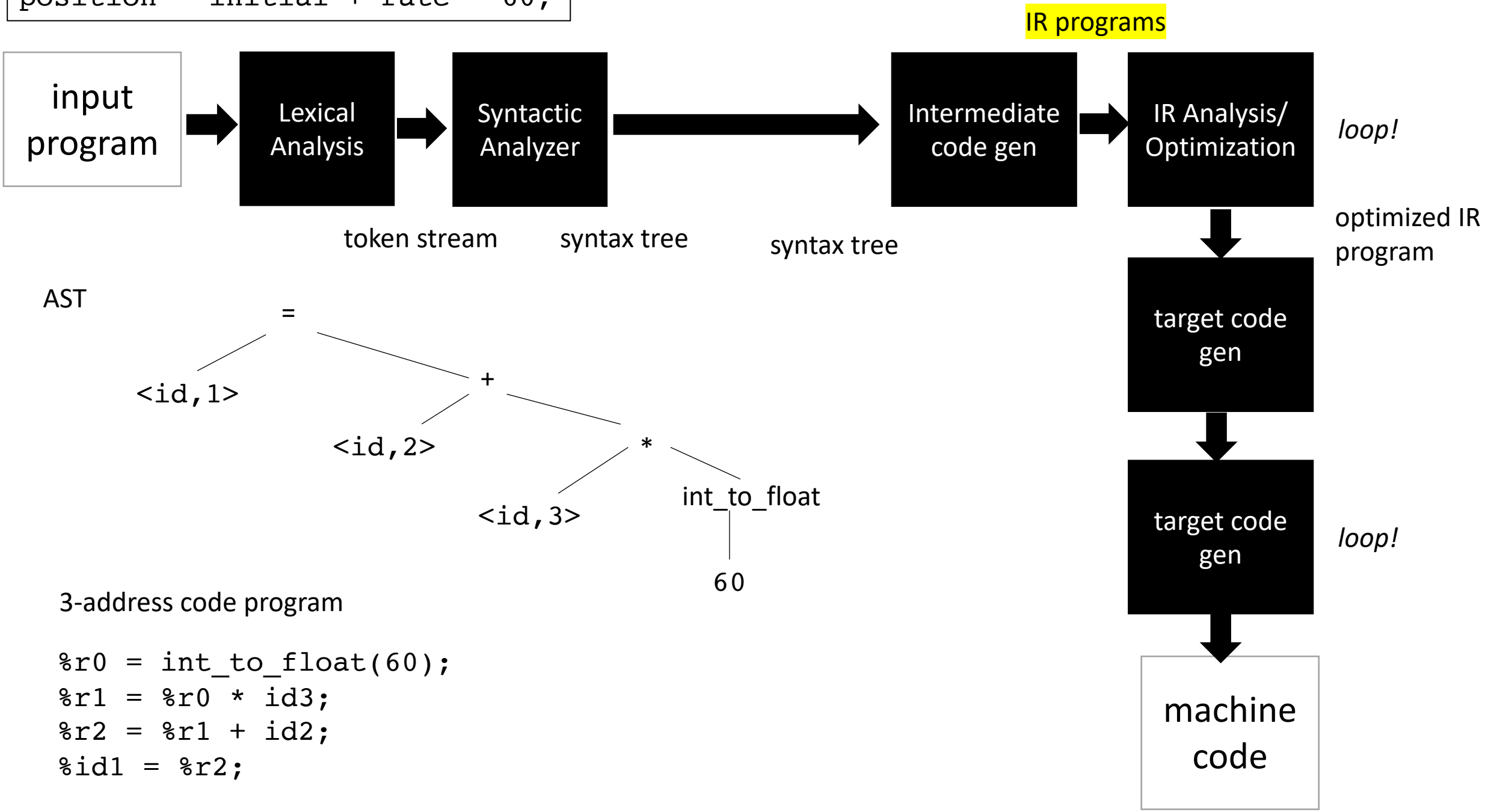
```
position = initial + rate * 60;
```



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



```
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

Intermediate representations

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

- Different optimizations and analysis are more suitable for IRs in different forms.

Our first IR: abstract syntax tree

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

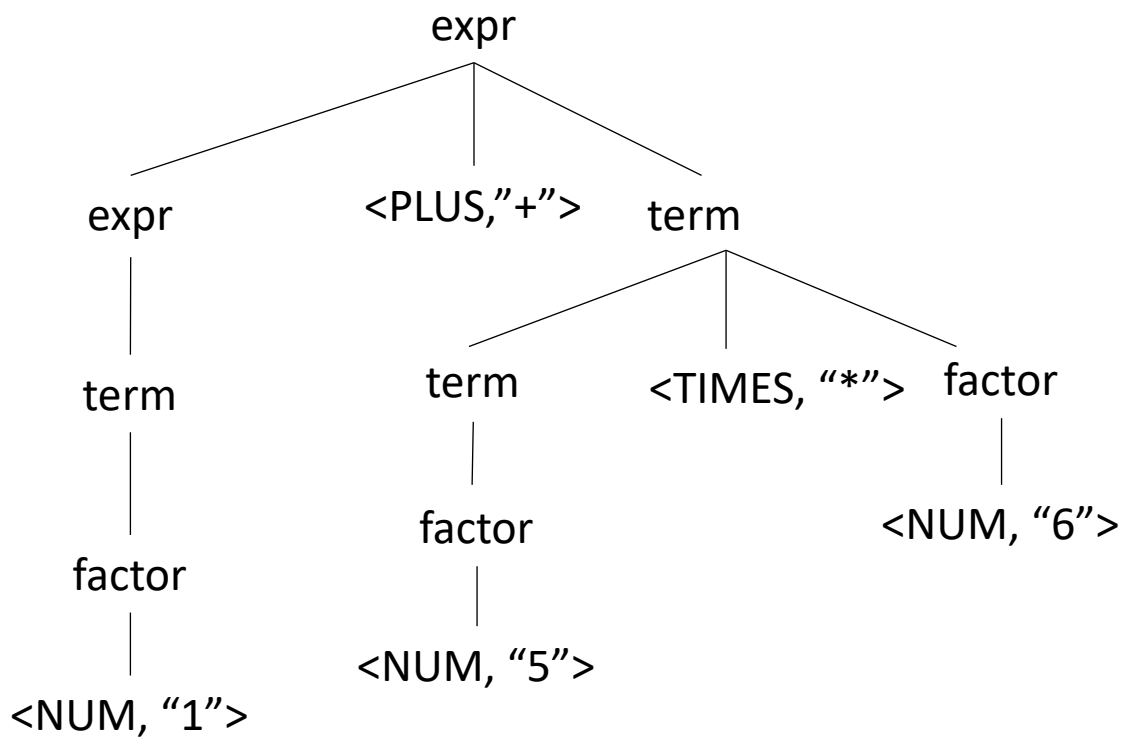
What is an AST?

We'll start by looking at a parse tree:

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

```
tokens = [
          (NUM, "[0-9]+"),
          (TIMES, "\*"),
          (PLUS, "\+"),
          (LPAR, "\(",
          (RPAR, "\)"
         ]
```

`input: 1+5*6`



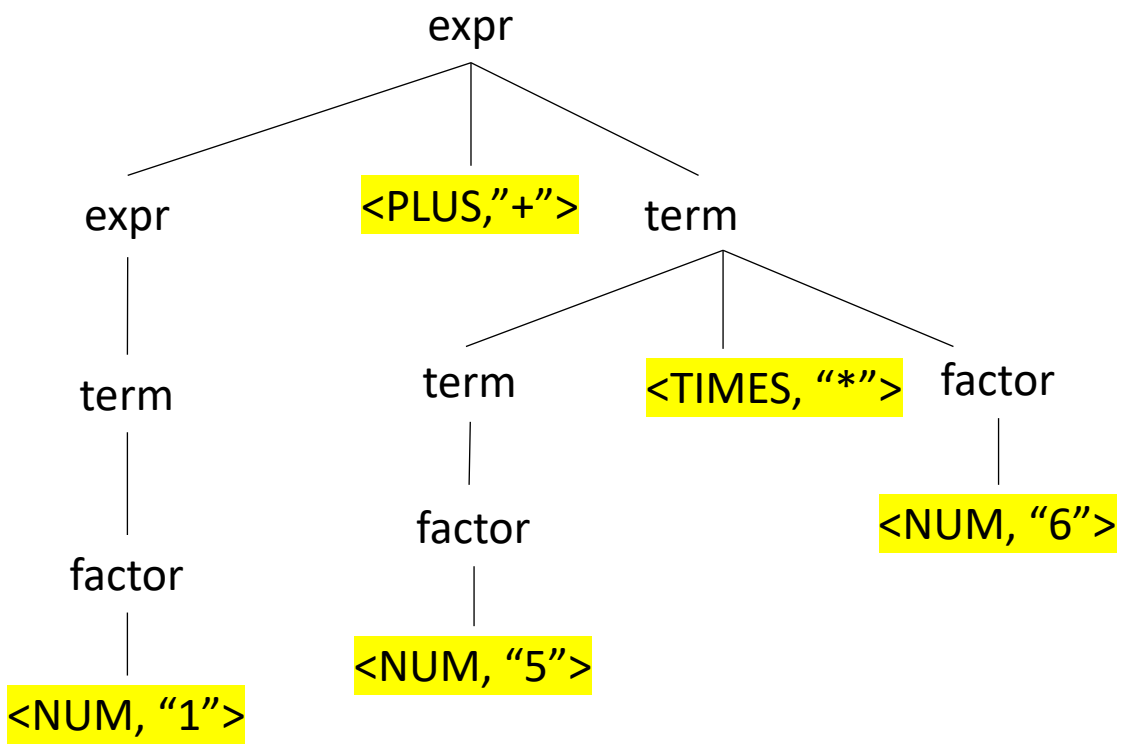
What is an AST?

We'll start by looking at a parse tree:

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

```
tokens = [
          (NUM, "[0-9]+"),
          (TIMES, "\*"),
          (PLUS, "\+"),
          (LPAR, "\(",
          (RPAR, "\)"
         ]
```

`input: 1+5*6`



What are leaves?

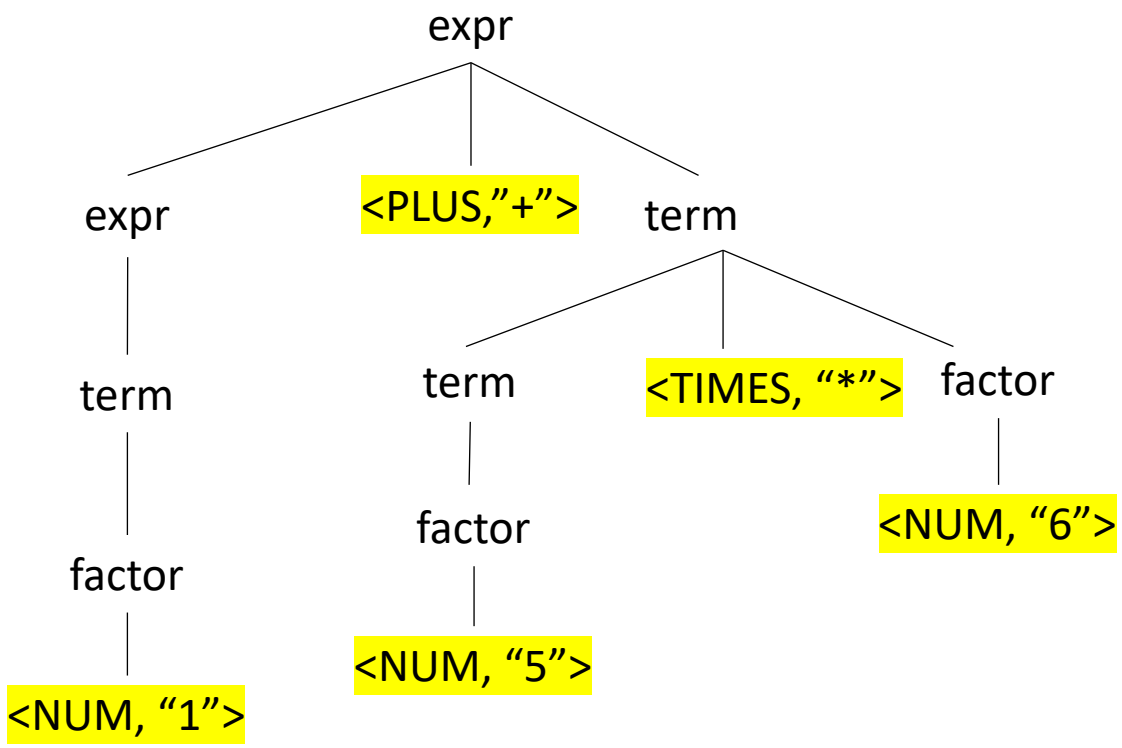
What is an AST?

We'll start by looking at a parse tree:

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

```
tokens = [
          (NUM, "[0-9]+"),
          (TIMES, "\*"),
          (PLUS, "\+"),
          (LPAR, "\(",
          (RPAR, "\)"
         ]
```

`input: 1+5*6`



What are leaves? lexemes

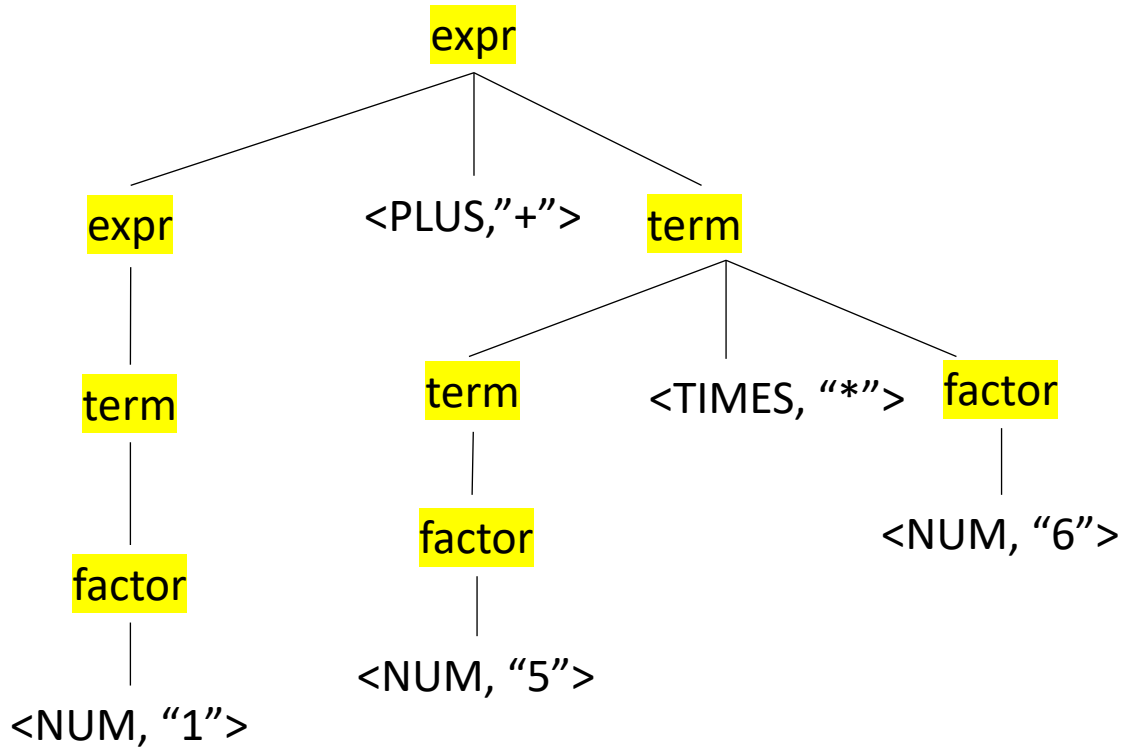
What is an AST?

We'll start by looking at a parse tree:

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

```
tokens = [
          (NUM, "[0-9]+"),
          (TIMES, "\*"),
          (PLUS, "\+"),
          (LPAR, "\(",
          (RPAR, "\)"
         ]
```

`input: 1+5*6`



What are nodes?

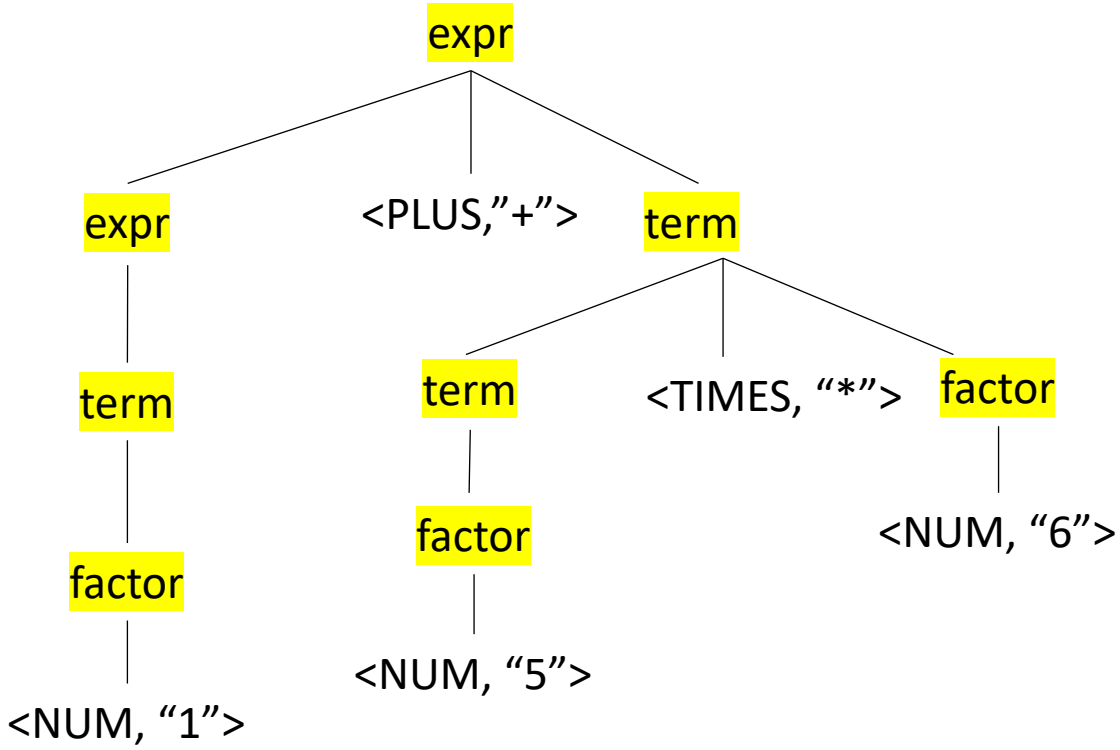
What is an AST?

We'll start by looking at a parse tree:

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

```
tokens = [
          (NUM, "[0-9]+"),
          (TIMES, "\*"),
          (PLUS, "\+"),
          (LPAR, "\(",
          (RPAR, "\)"
         ]
```

```
input: 1+5*6
```



What are nodes? non-terminals

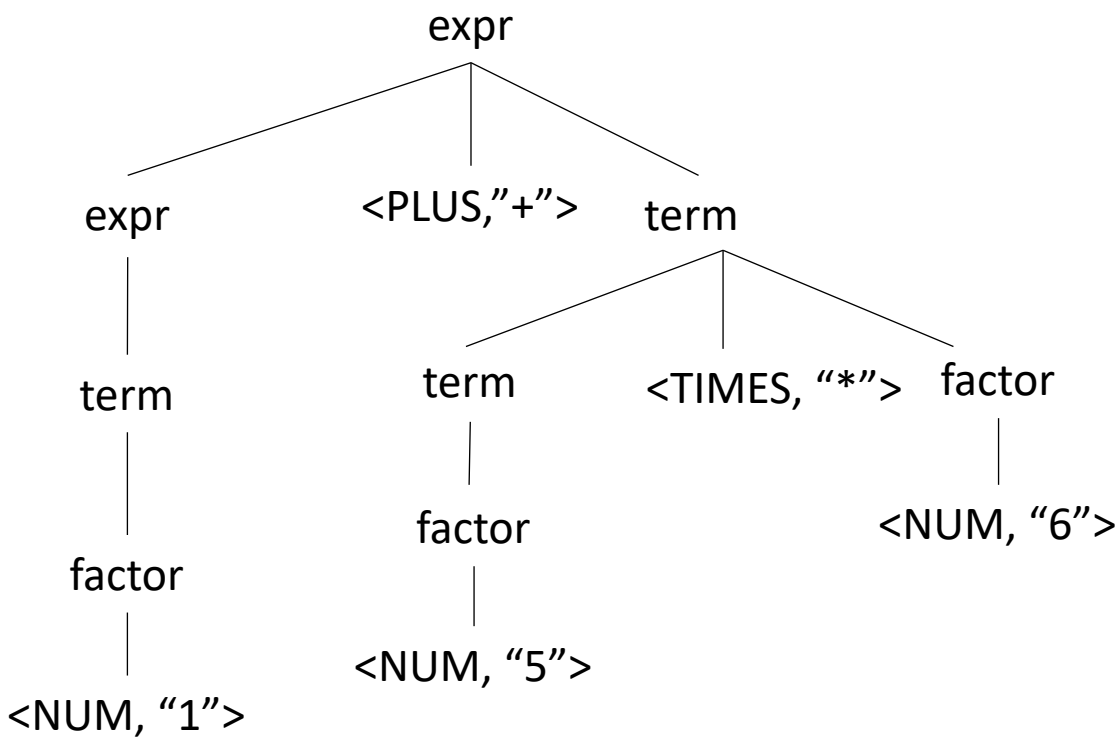
What is an AST?

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

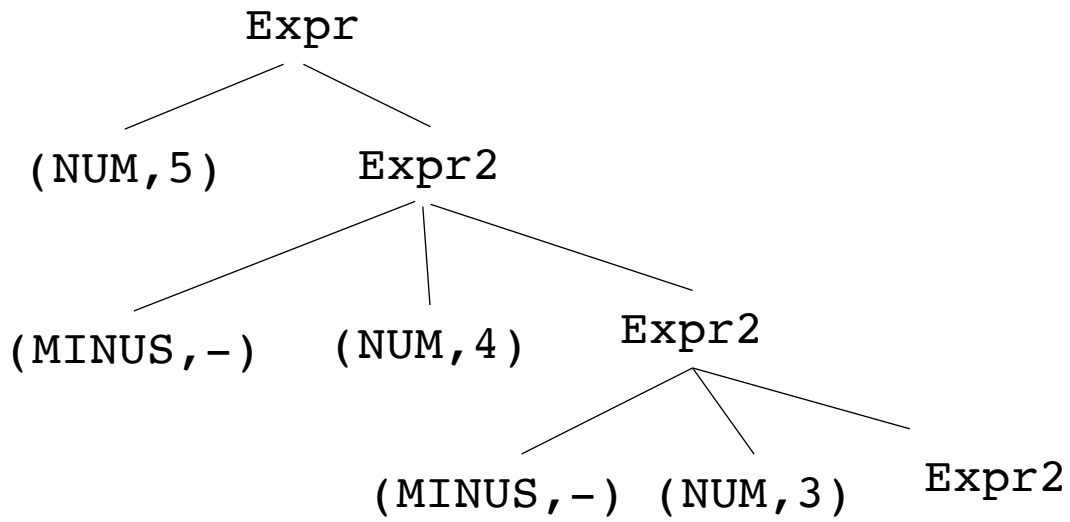
`input: 1+5*6`



in your homework, do you actually make a parse tree?

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

5 - 4 - 3



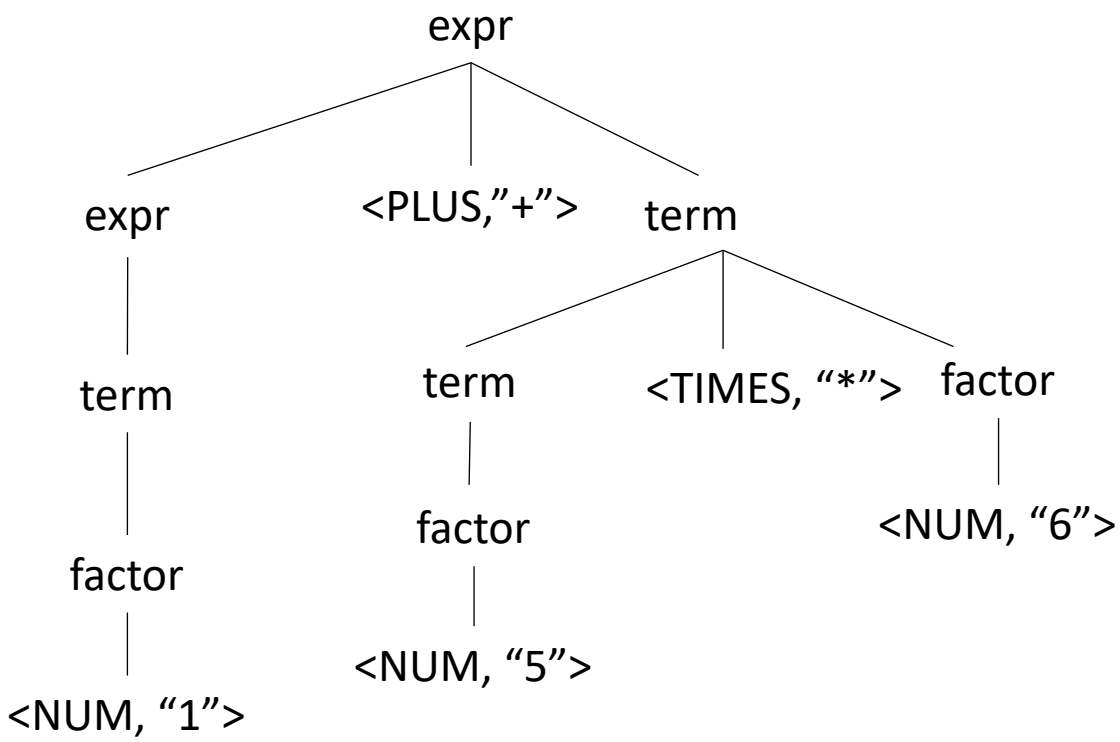
What is an AST?

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

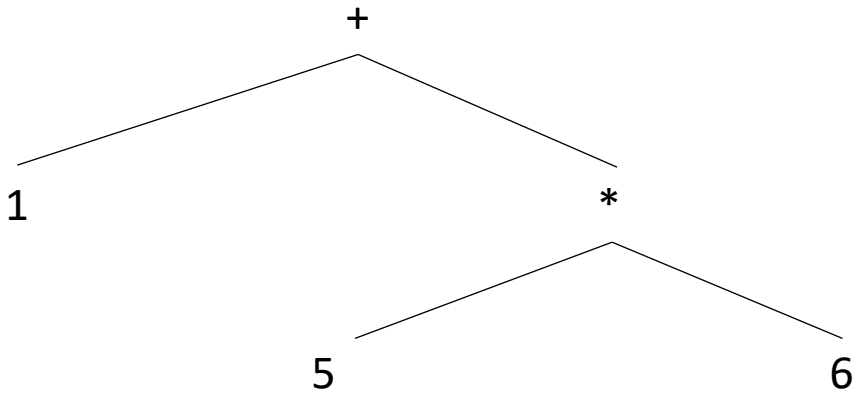
`input: 1+5*6`



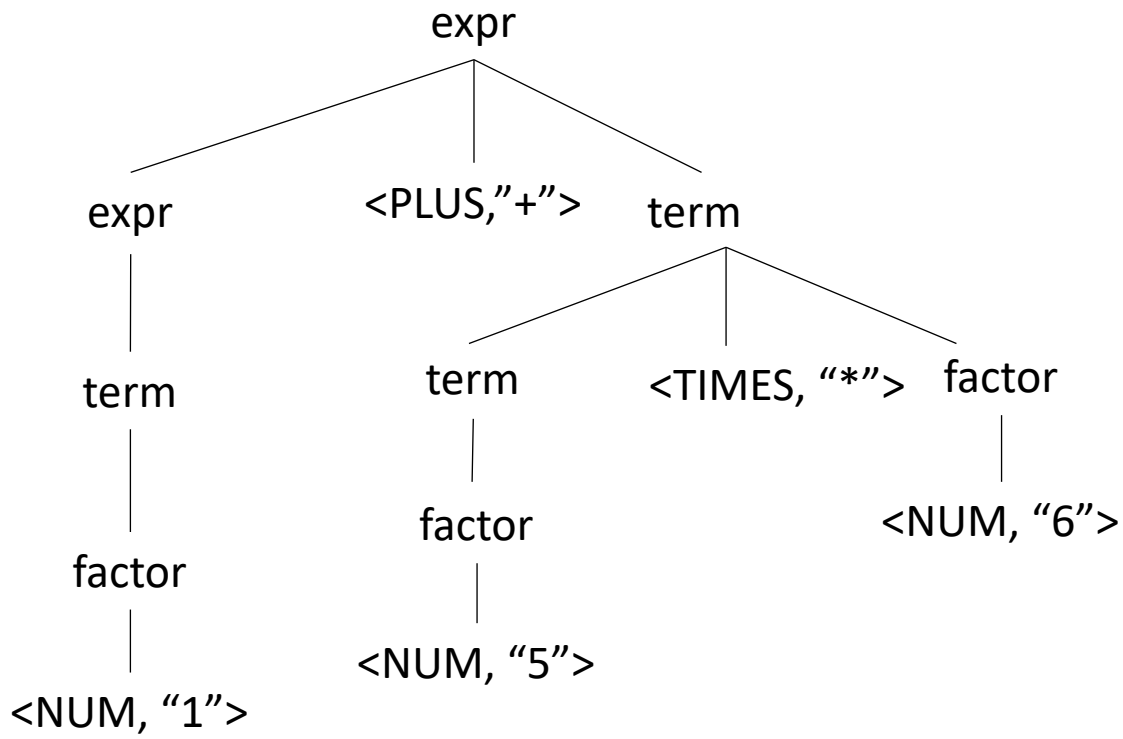
What is an AST?

`input: 1+5*6`

What are some differences?



AST

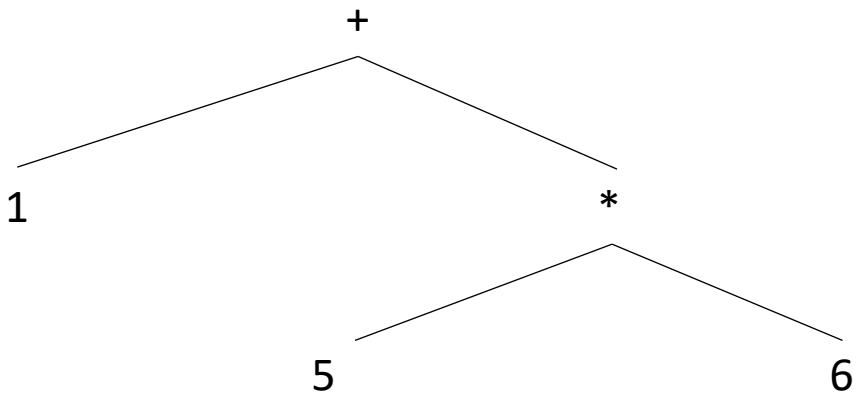


Parse Tree

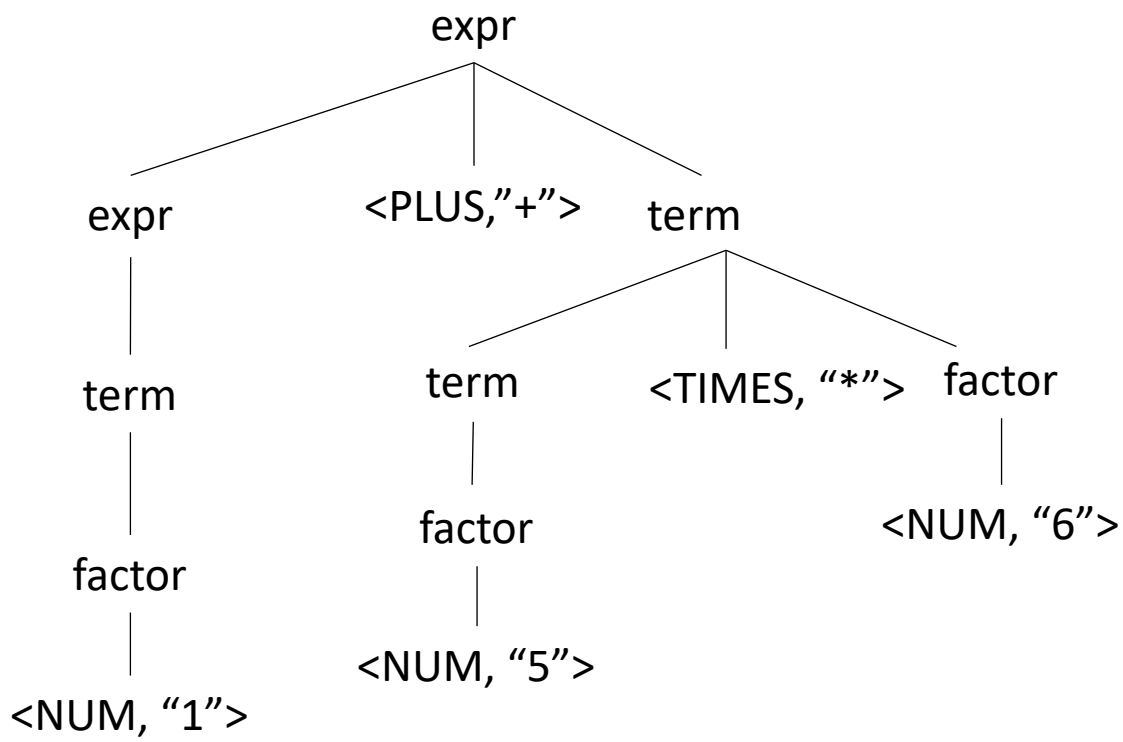
`input: 1+5*6`

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

Example

```
input: (1+5)*6
```

expr

what happens to ()s in an AST?

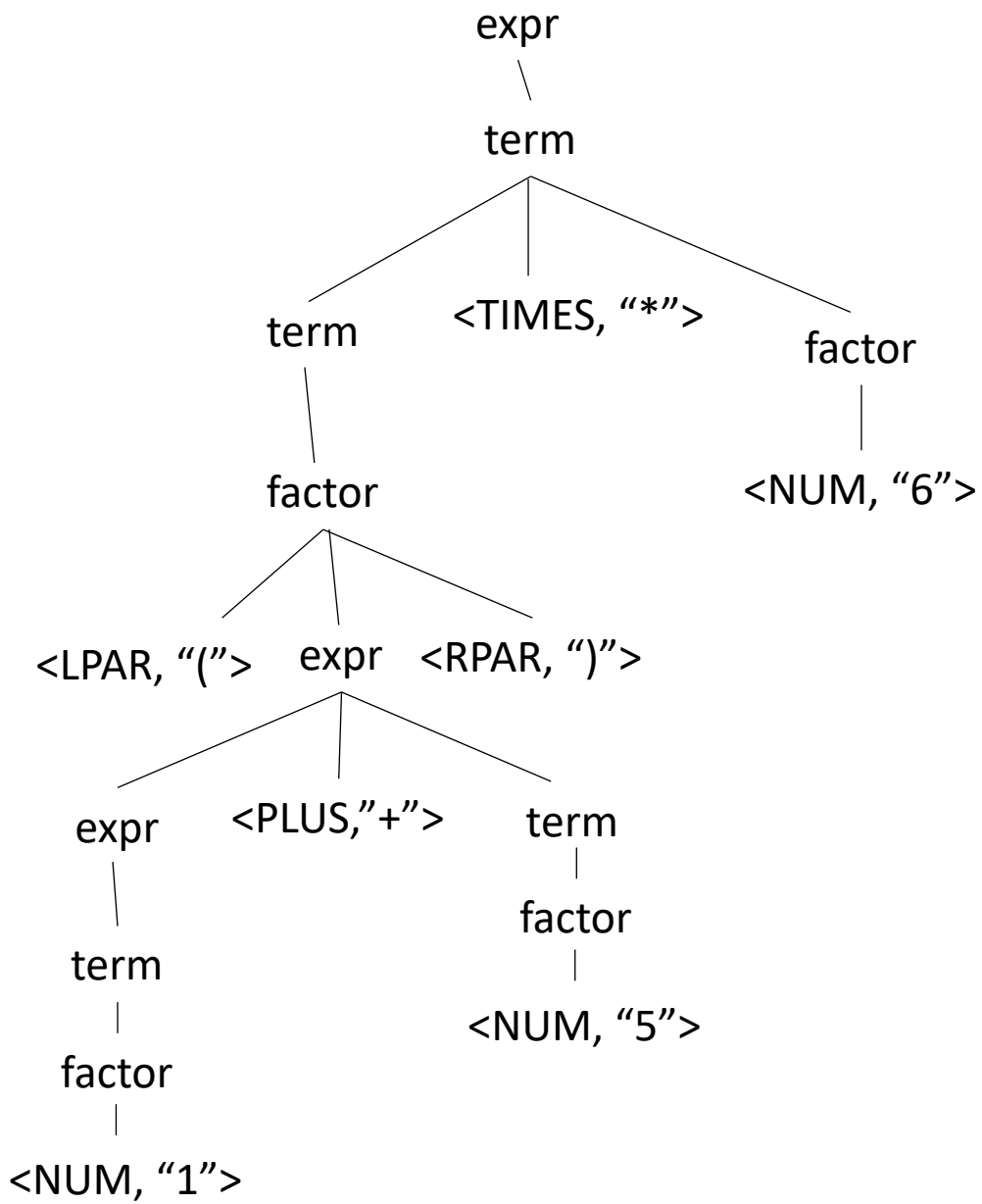
Operator	Name	Productions
+	`expr`	`: expr PLUS term` `| term`
*	`term`	`: term TIMES factor` `| factor`
()	`factor`	`: LPAR expr RPAR` `| NUM`

Example

what happens to ()s in an AST?

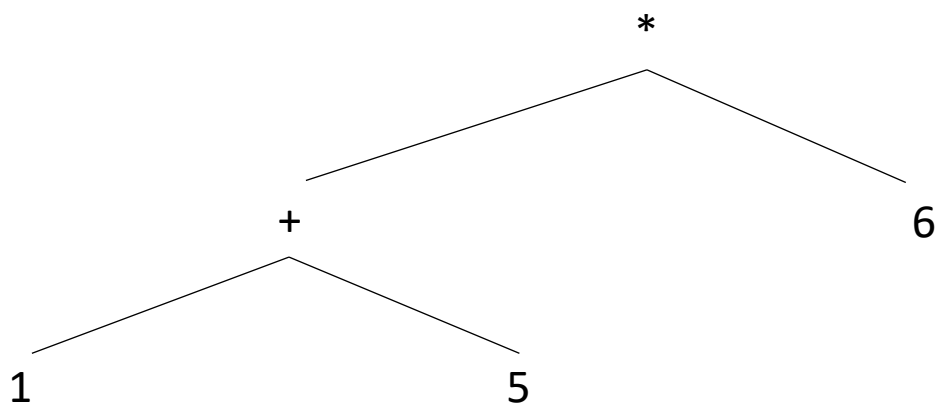
Operator	Name	Productions
+	expr	: expr PLUS term | term
*	term	: term TIMES factor | factor
()	factor	: LPAR expr RPAR | NUM

input: (1+5)*6



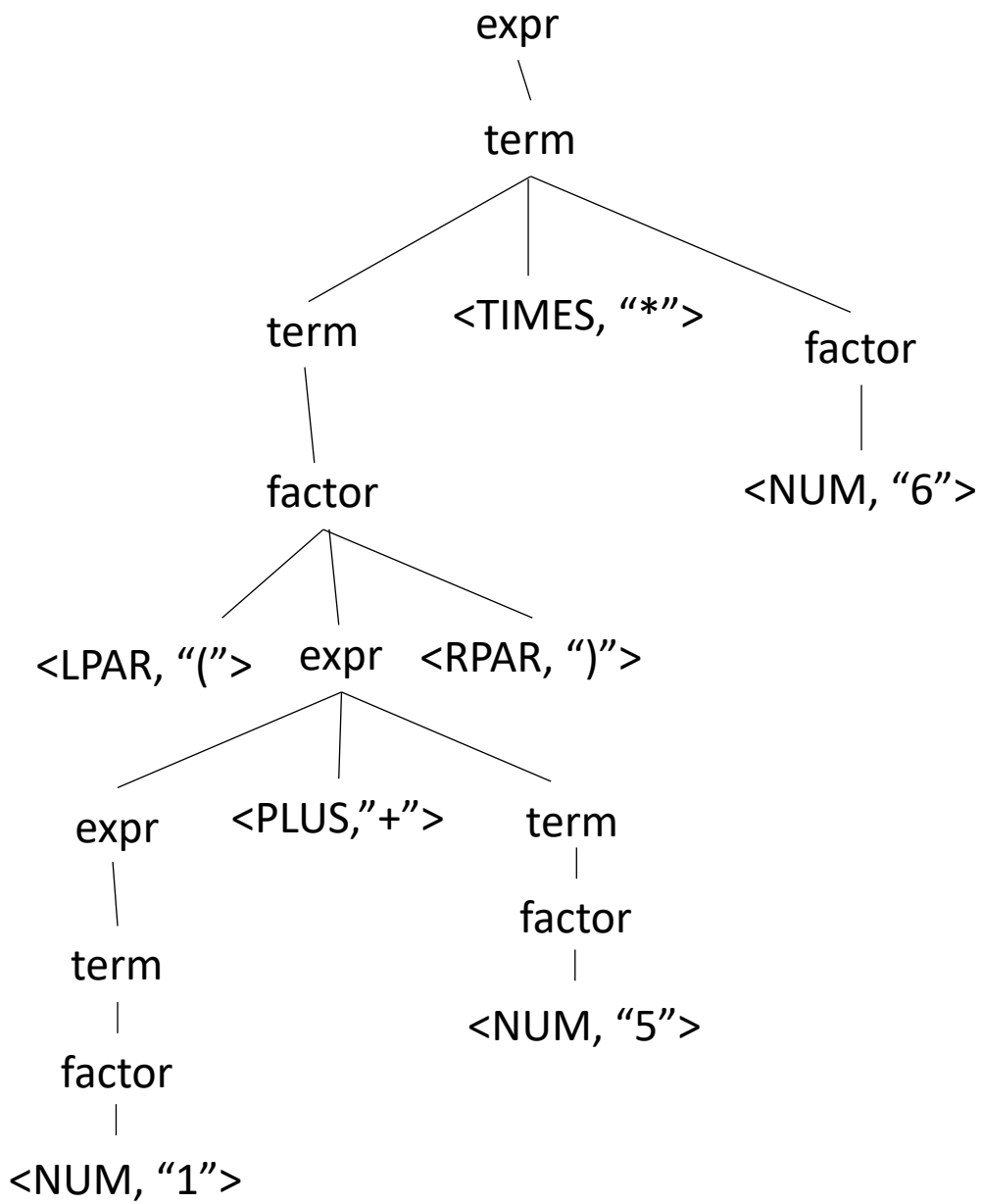
Example

what happens to ()s in an AST?



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

`input: (1+5)*6`



formalizing an AST

- A tree based data structure, used to represent ***expressions***

- Main building block: Node
  - Leaf node: ID or Number
  - Node with one child: Unary operator (`-`) or type conversion (`int_to_float`)
  - Node with two children: Binary operator (`+`, `*`)

```
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	{} {}
*	term	: term TIMES factor | factor	{} {}
()	factor	: LPAR expr RPAR | NUM | ID	{} {} {}

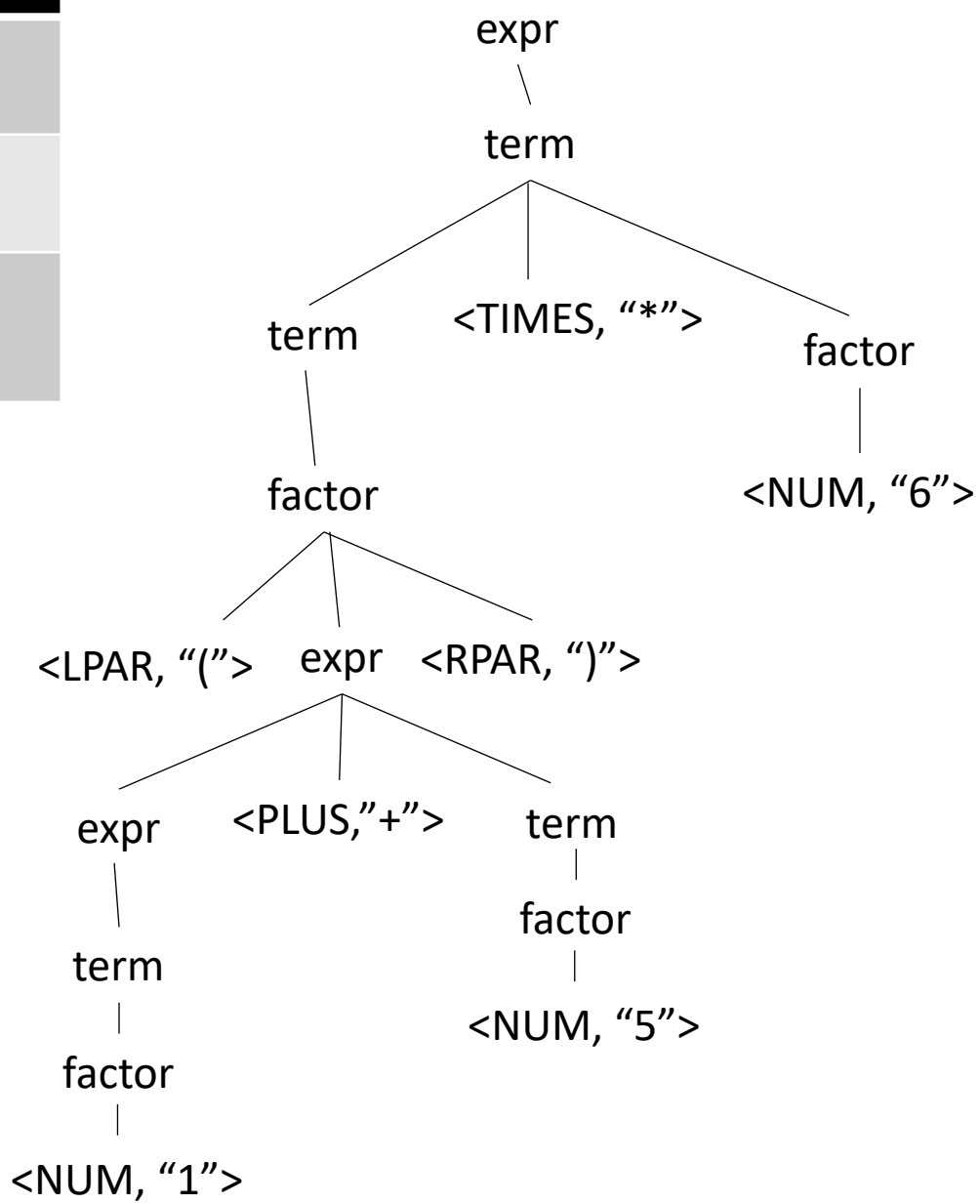
Creating an AST from production rules

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)}

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)}

Lets build the AST

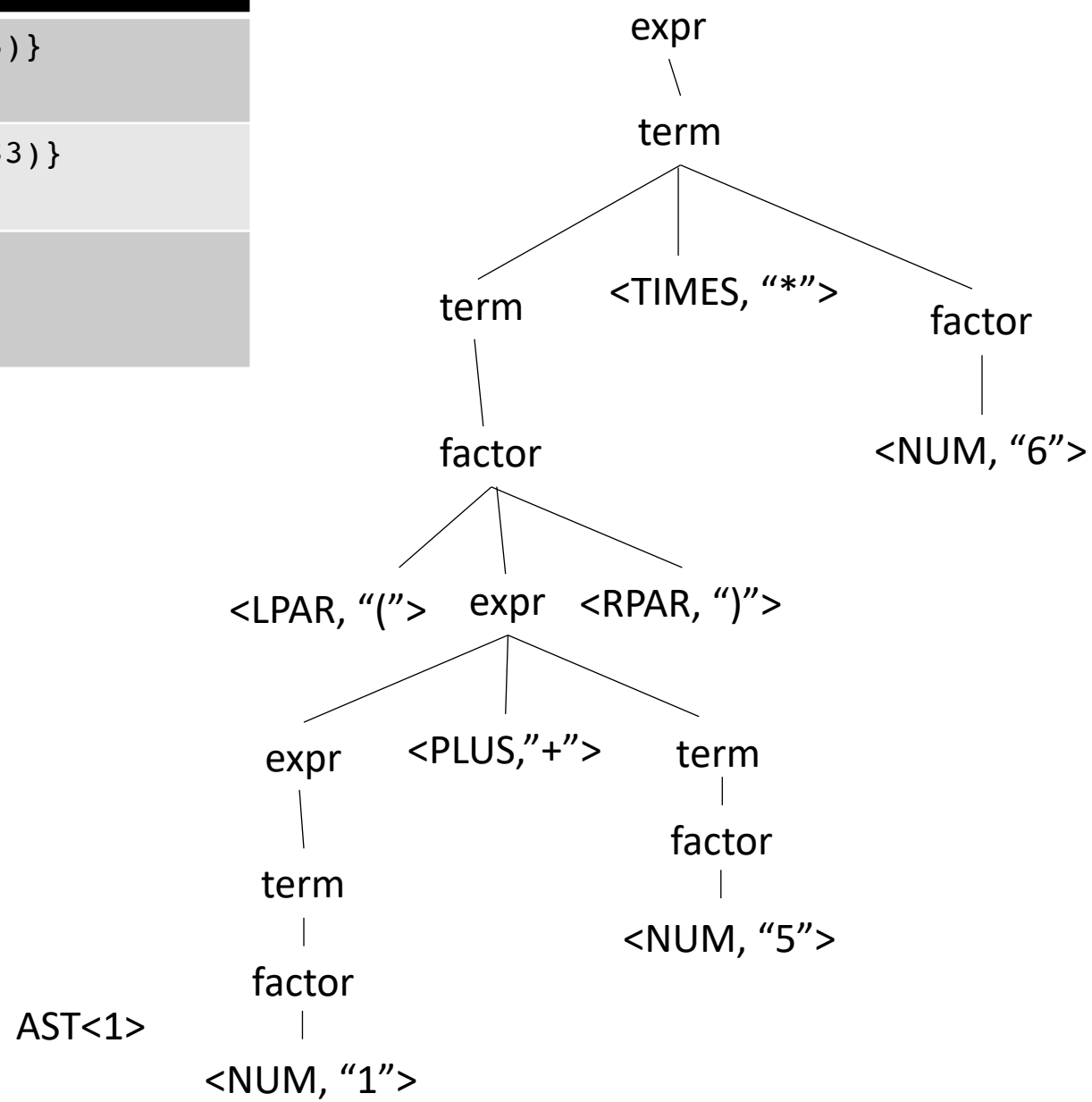
input: (1+5)*6



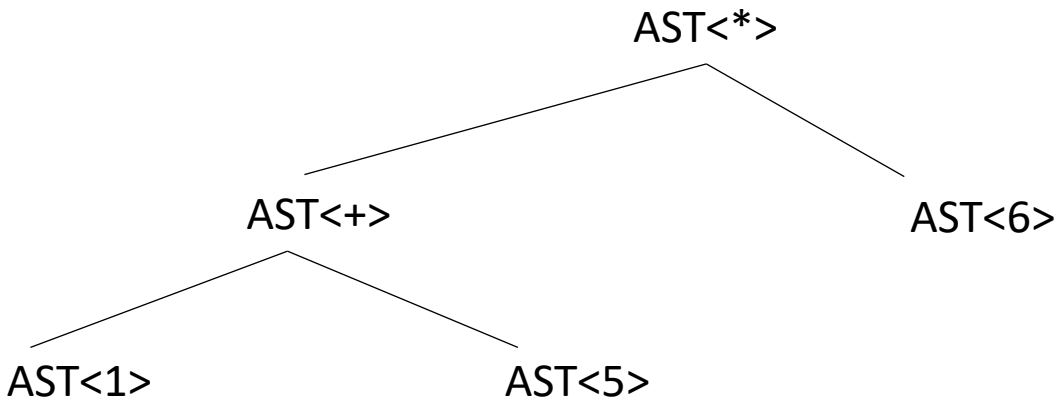
Name	Productions	Production action
expr	: expr PLUS term | term	{return ASTAddNode($1,$3)} {return $1}
term	: term TIMES factor | factor	{return ASTMultNode($1,$3)} {return $1}
factor	: LPAR expr RPAR | NUM | ID	{return $2} {return ASTNumNode($1)} {return ASTIDNode($1)}

input: (1+5)*6

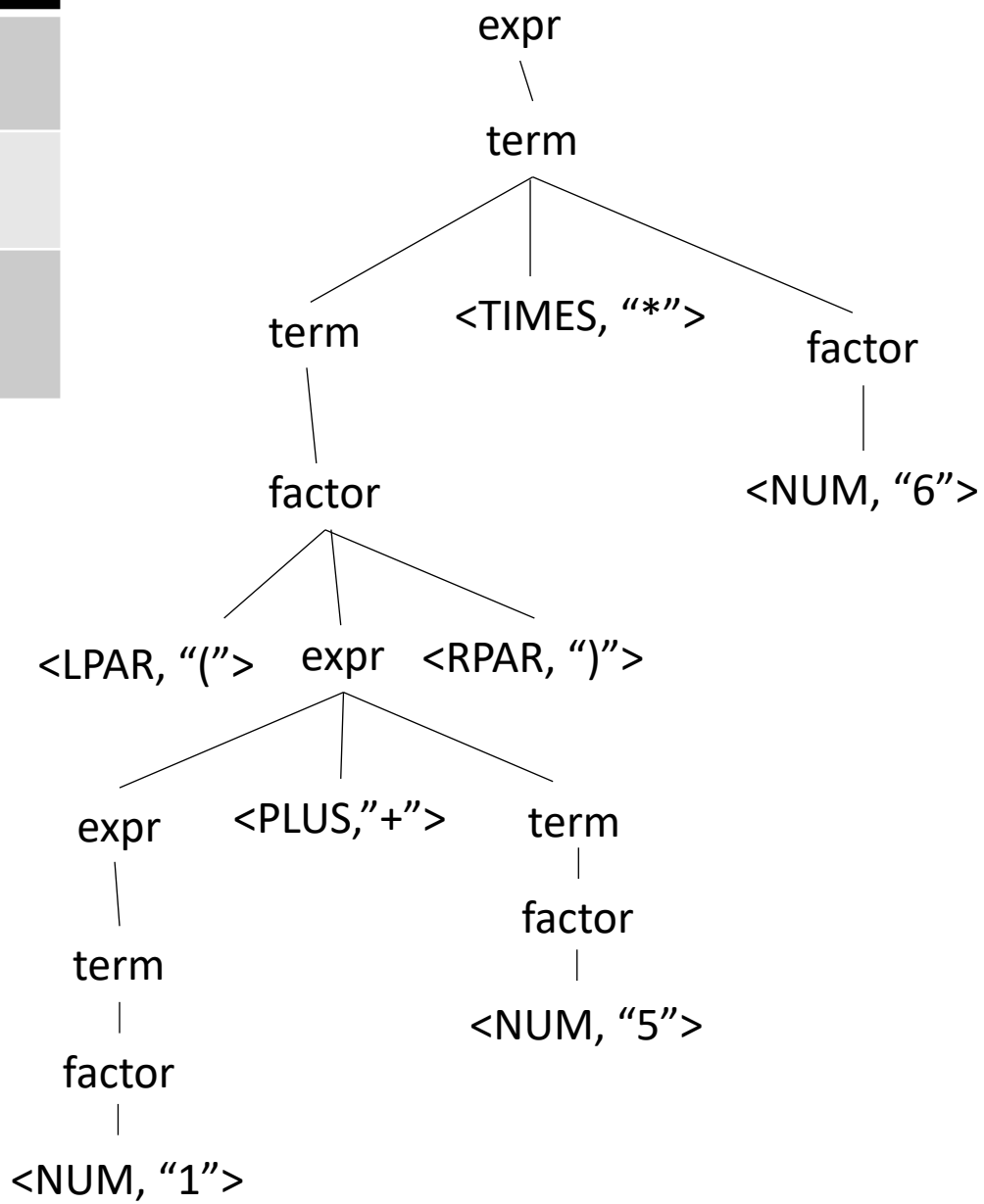
Lets build the AST



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

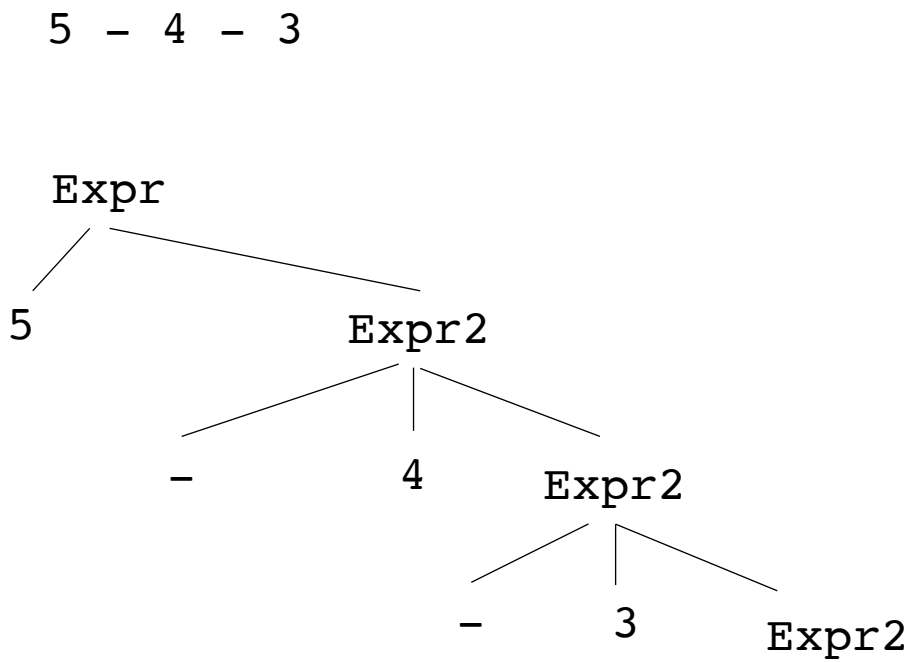


input: (1+5)*6



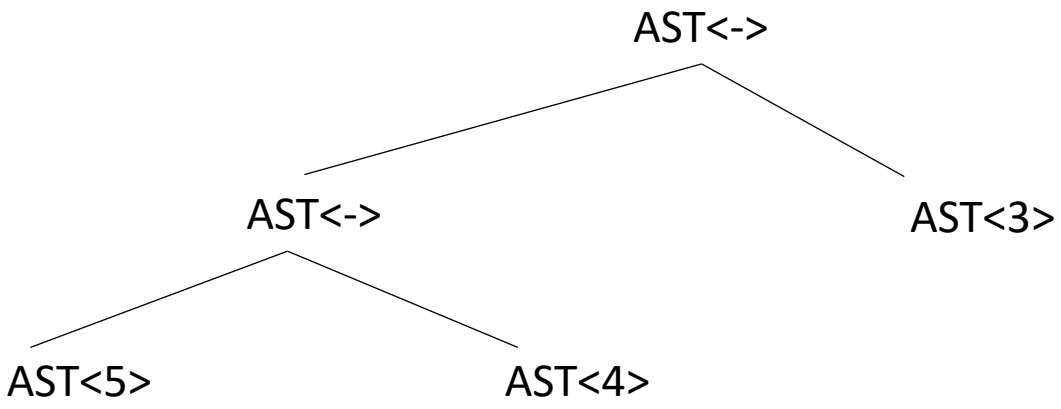
Creating an AST from predictive grammar

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

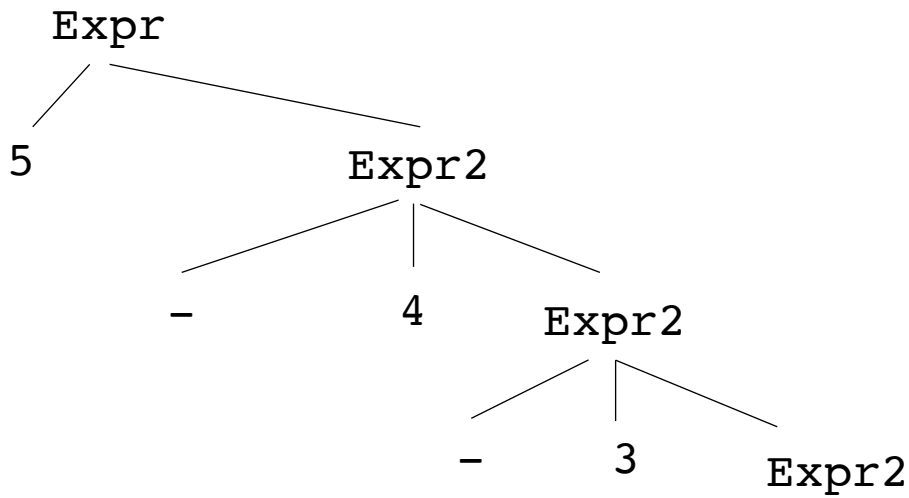


Creating an AST from predictive grammar

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



5 – 4 – 3

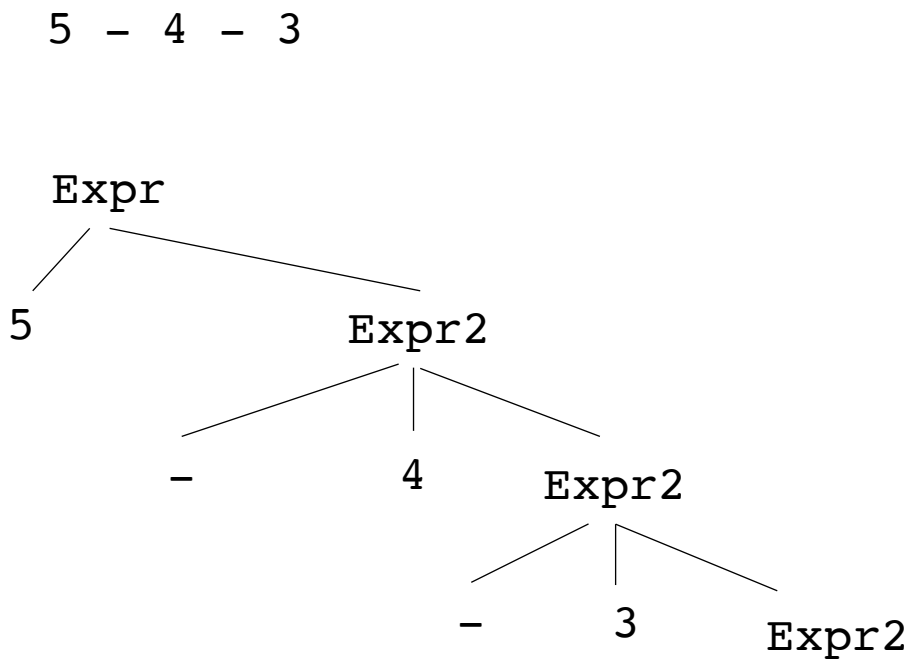


How do we get to the desired parse tree?

Creating an AST from predictive grammar

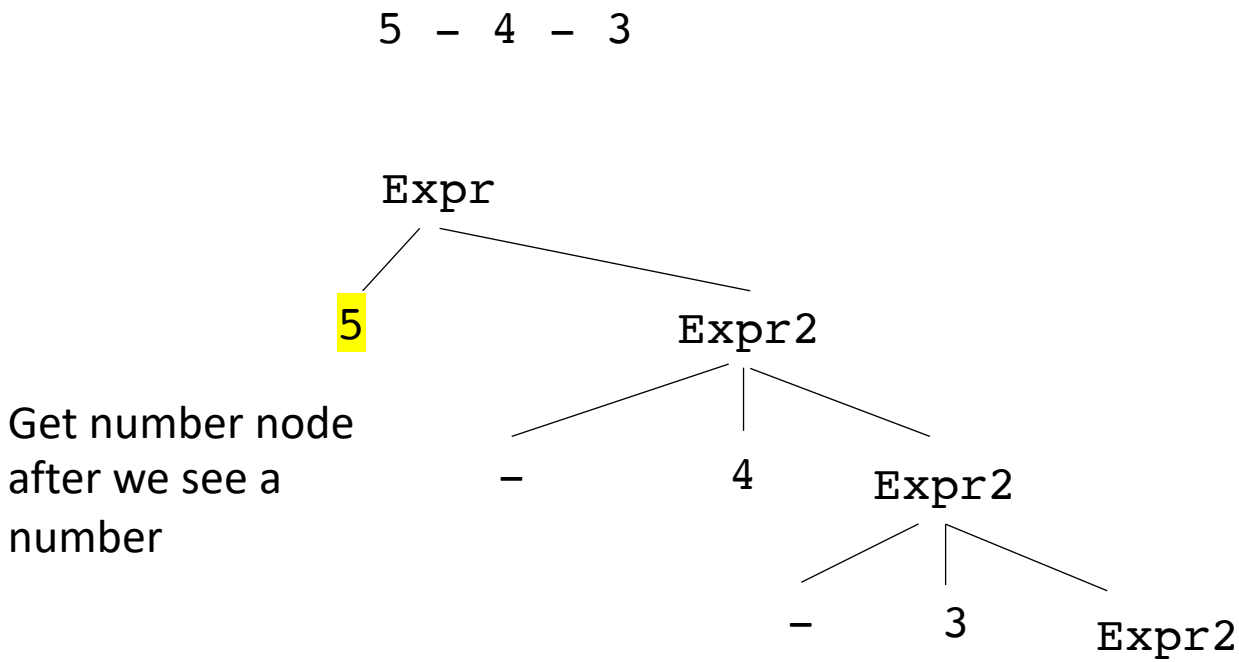
```
Expr  ::= NUM Expr2
Expr2 ::= MINUS NUM Expr2
        |   “”
```

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
       |   “”
```

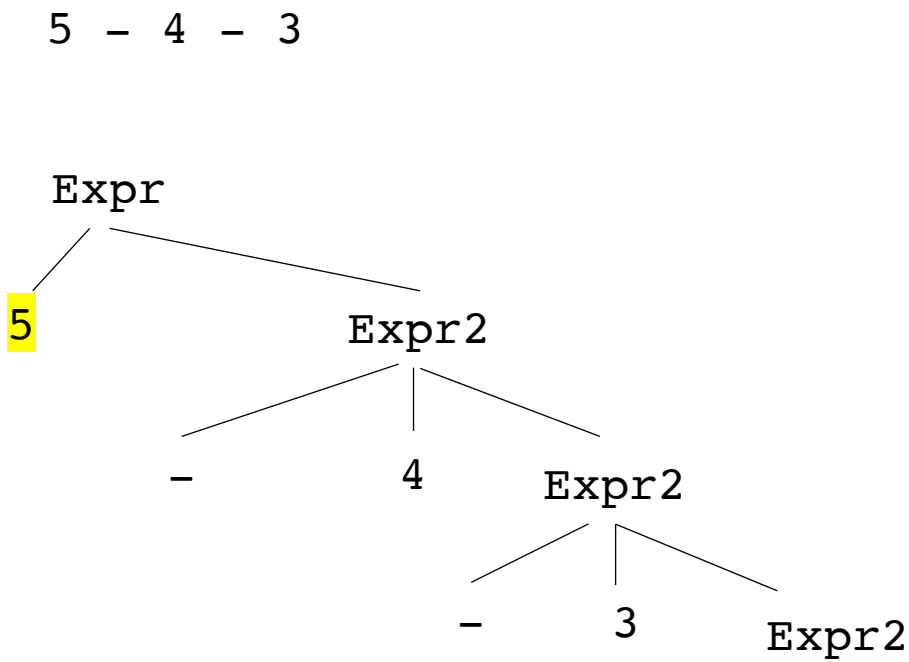


Get number node after we see a number

AST<5>

Creating an AST from predictive grammar

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

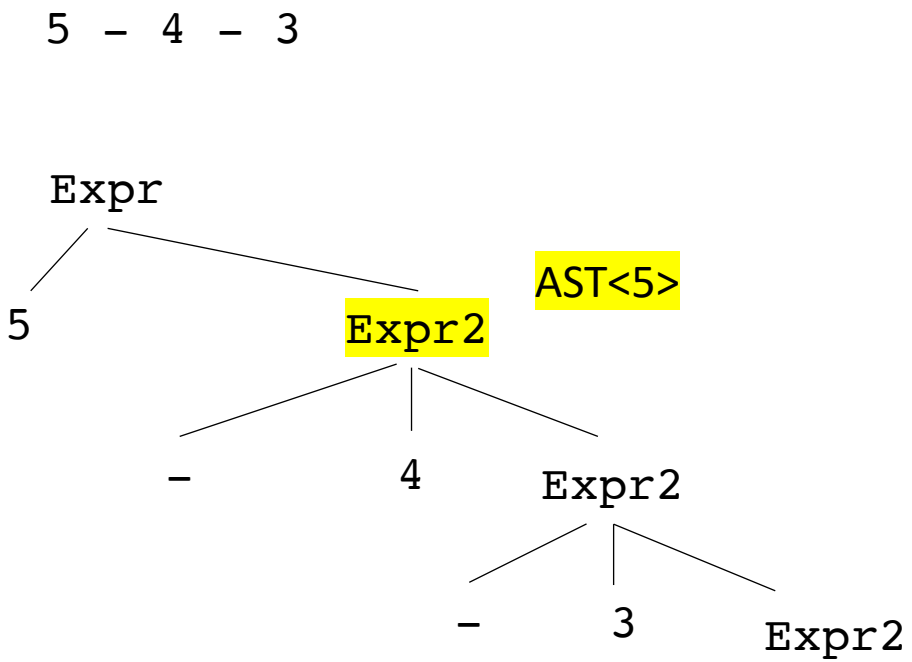


Pass the node down

AST<5>

Creating an AST from predictive grammar

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

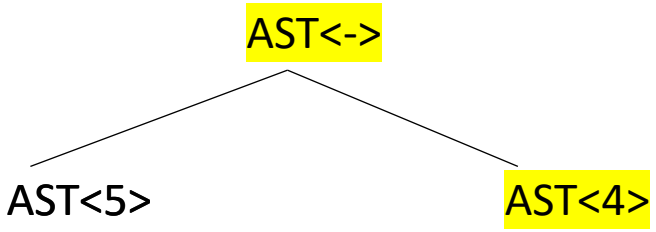


Pass the node down

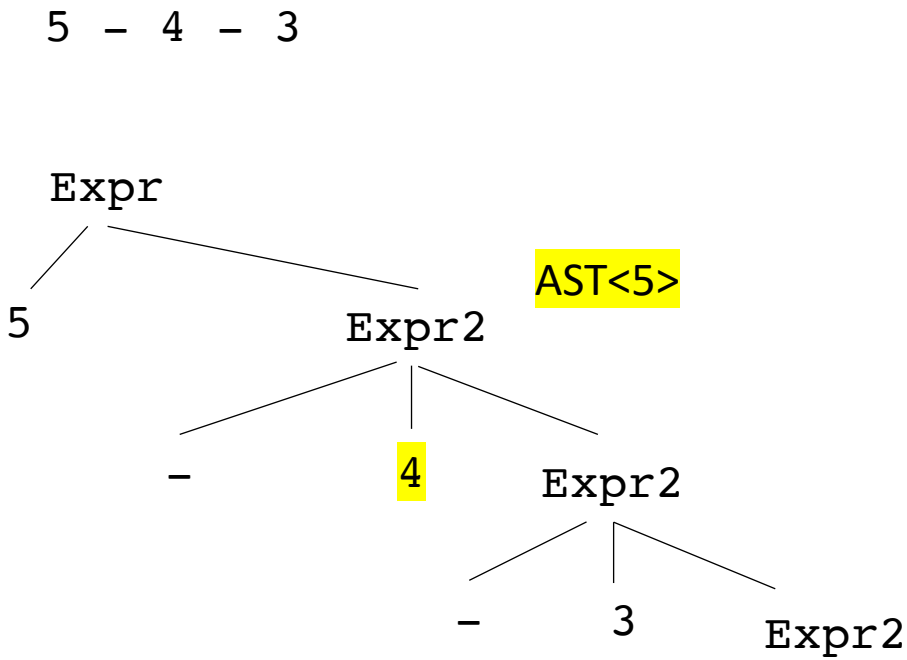
AST<5>

Creating an AST from predictive grammar

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

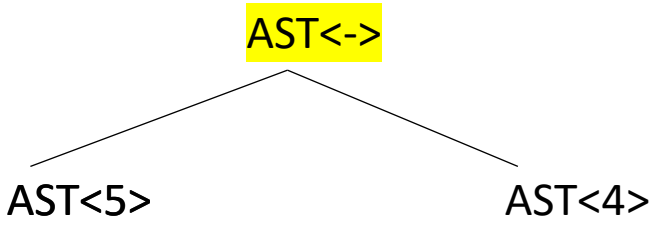


In Expr2, after 4 is parsed, create a number node and a minus node

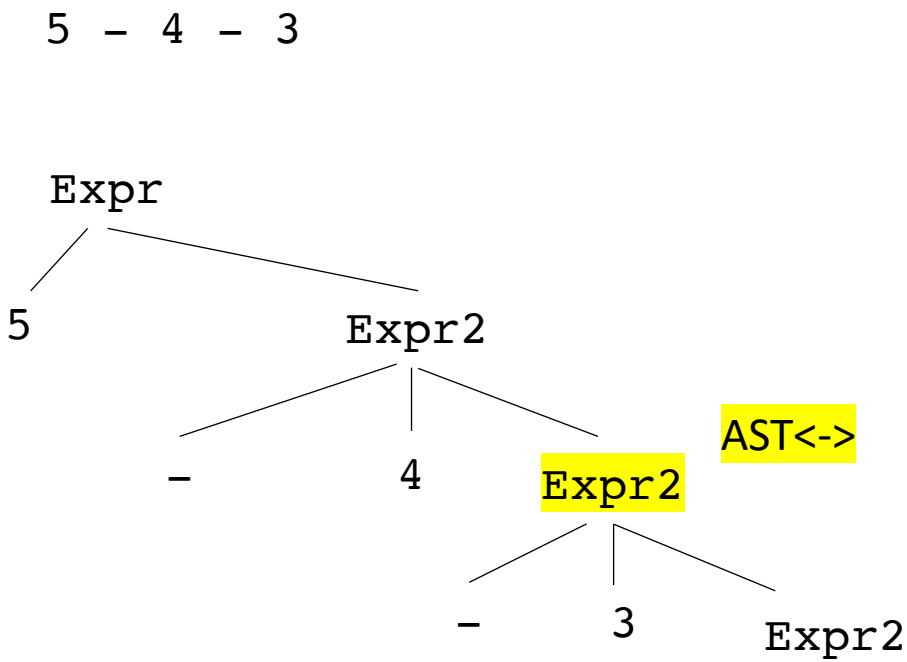


Creating an AST from predictive grammar

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

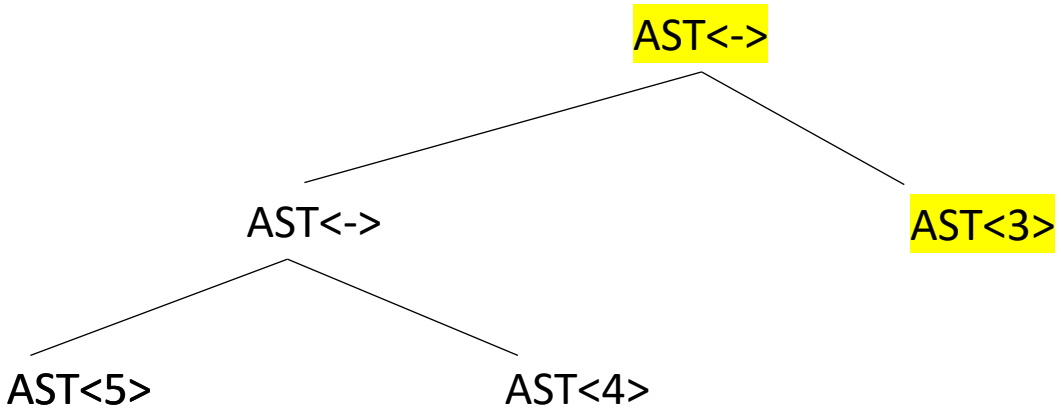


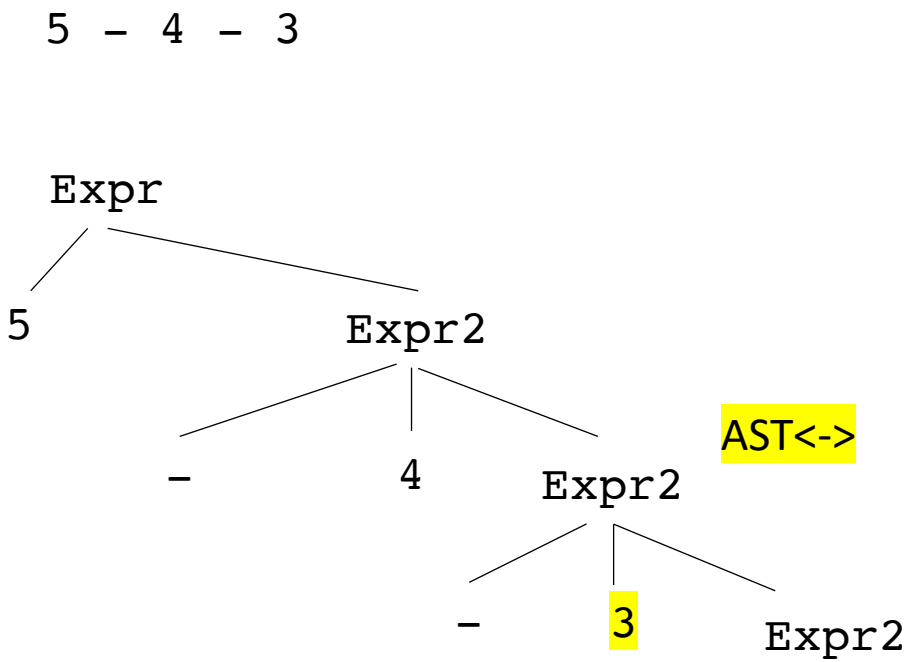
pass the new node down



Creating an AST from predictive grammar

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

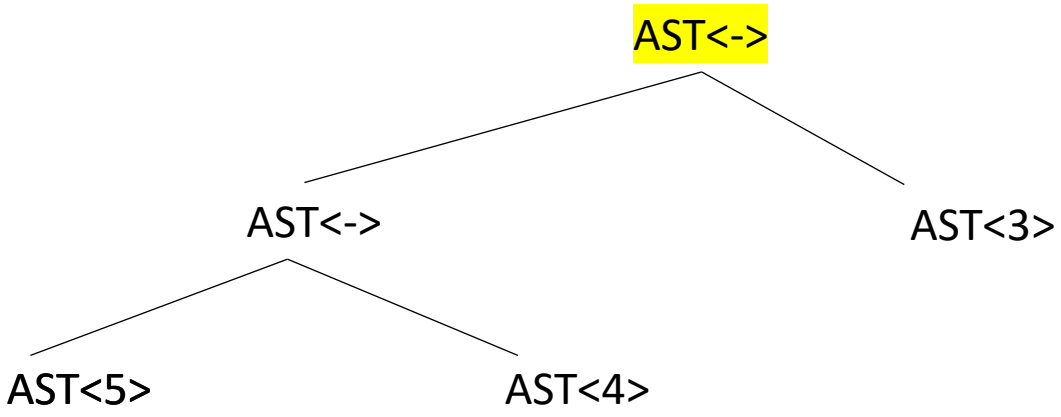




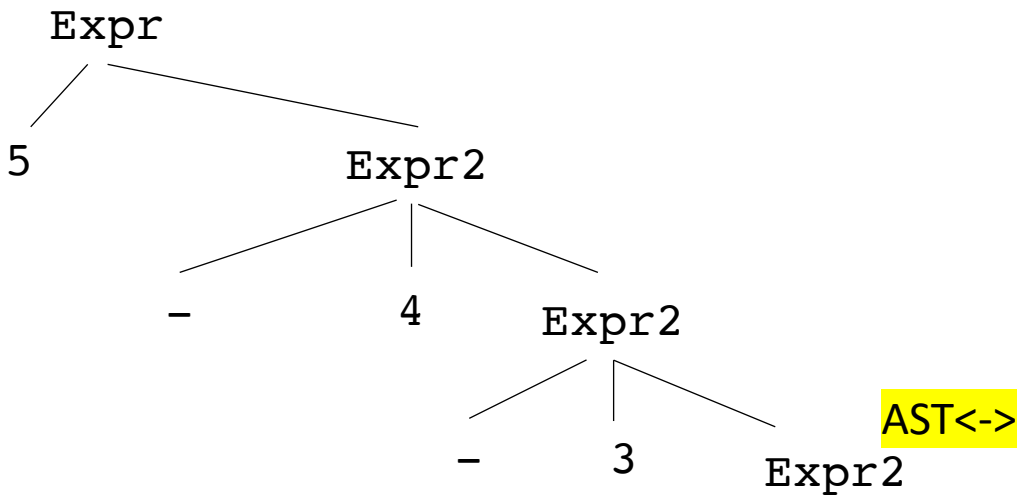
In Expr2, after 3 is parsed, create a number node and a minus node

Creating an AST from predictive grammar

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



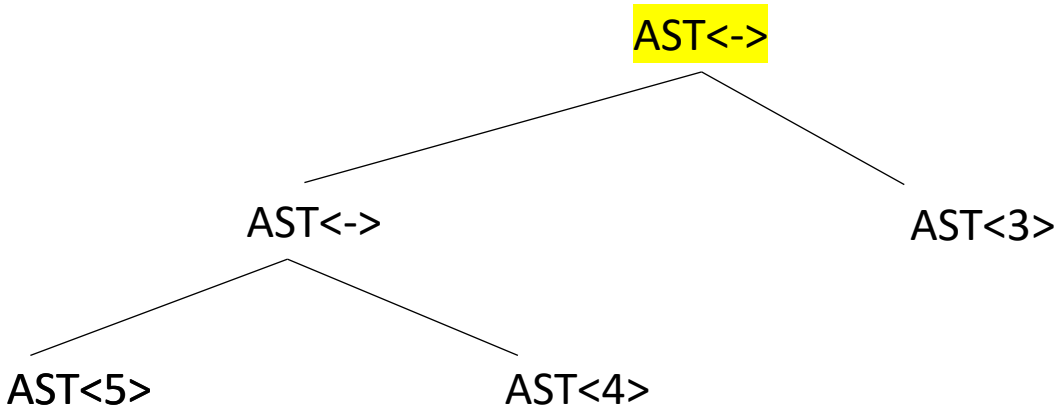
5 – 4 – 3

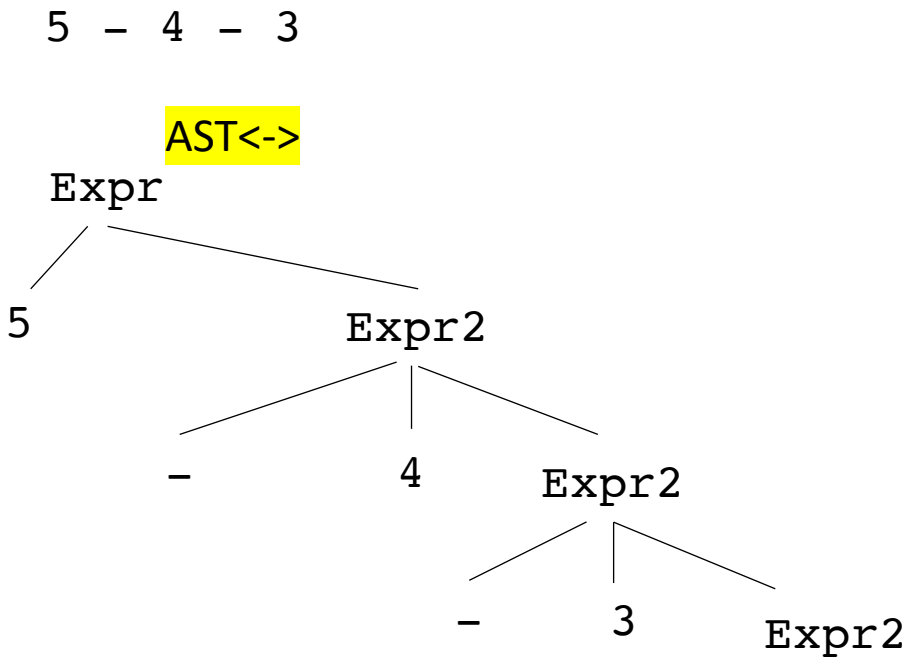


pass down the new node

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[1]
    node = ASTNumNode(value)
    self.eat("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[1]
    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.next_word[1]
        rhs_node = ASTNumNode(value)
        self.eat("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[1]
    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 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[1]
    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.next_word[1]
        rhs_node = ASTNumNode(value)
        self.eat("NUM")
        node = ASTMinusNode(lhs_node, rhs_node)
        return self.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("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

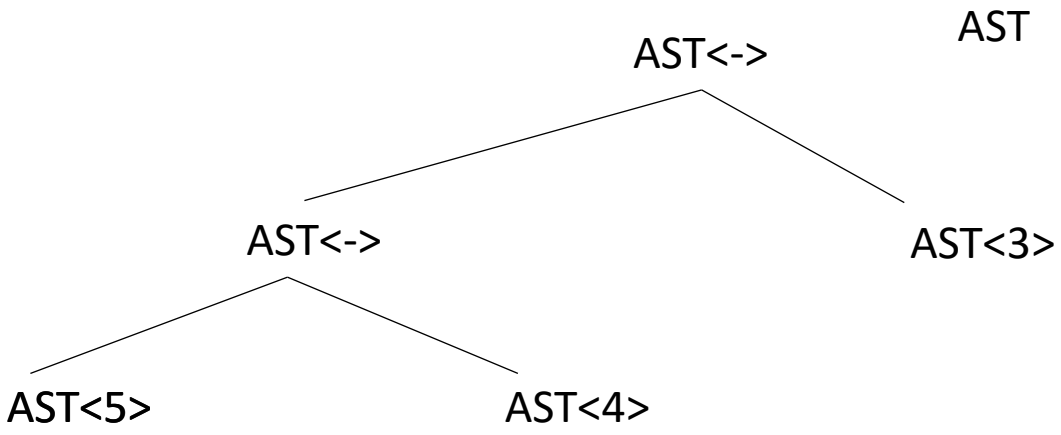
```
import ast

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

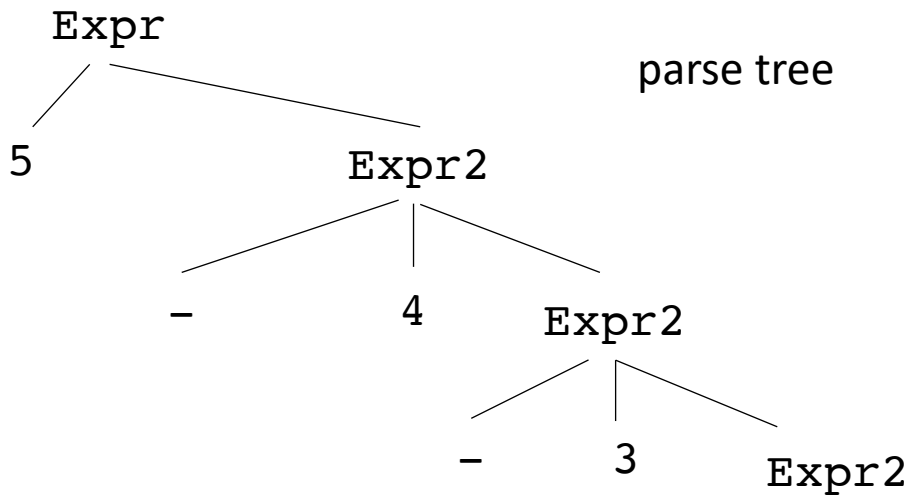
```
Expr(value=BinOp(left=BinOp(left=Num(n=5), op=Sub(), right=Num(n=4)), op=Sub(), right=Num(n=2)))
```

Evaluate an AST by doing a post order traversal

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



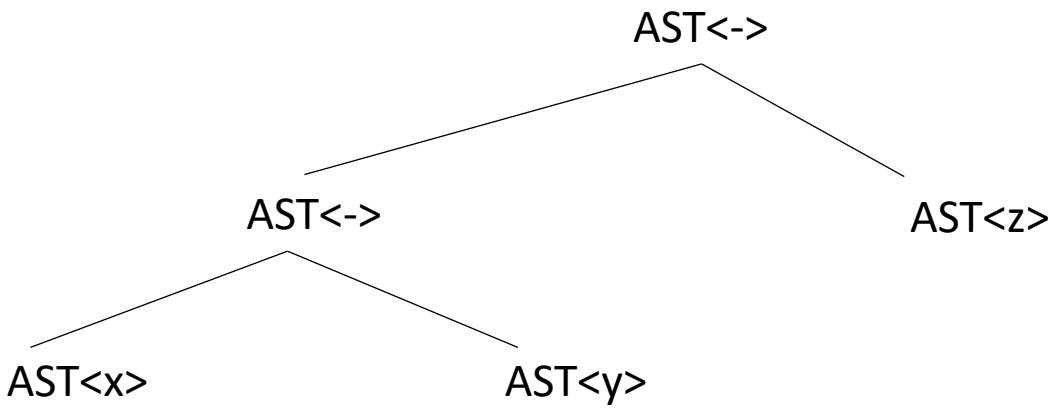
5 – 4 – 3



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
        |   “”
```

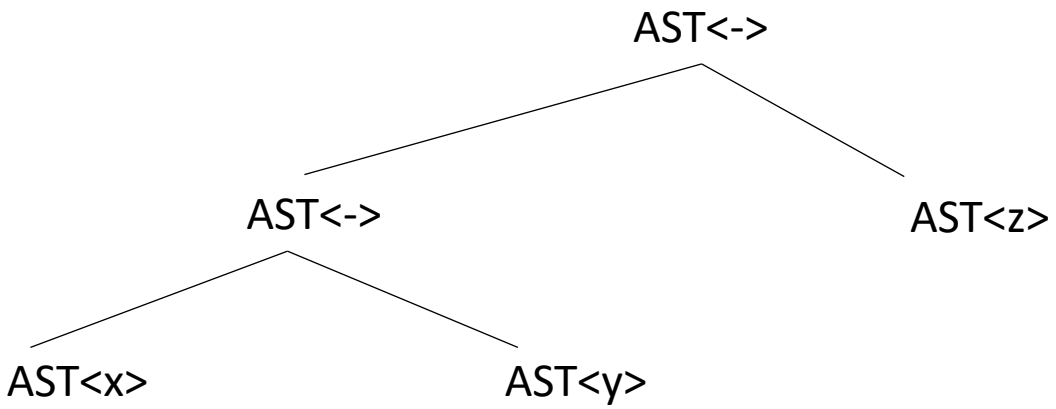


What if you cannot evaluate it?
What else might you do?

```
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
```

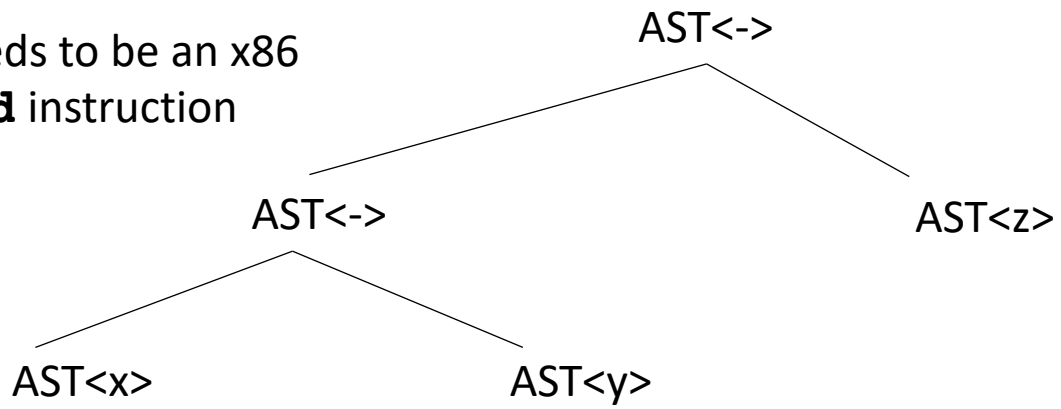
How does this change things?

Evaluate an AST by doing a post order traversal

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

needs to be an x86 **addss** instruction

needs to be an x86 **add** 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?

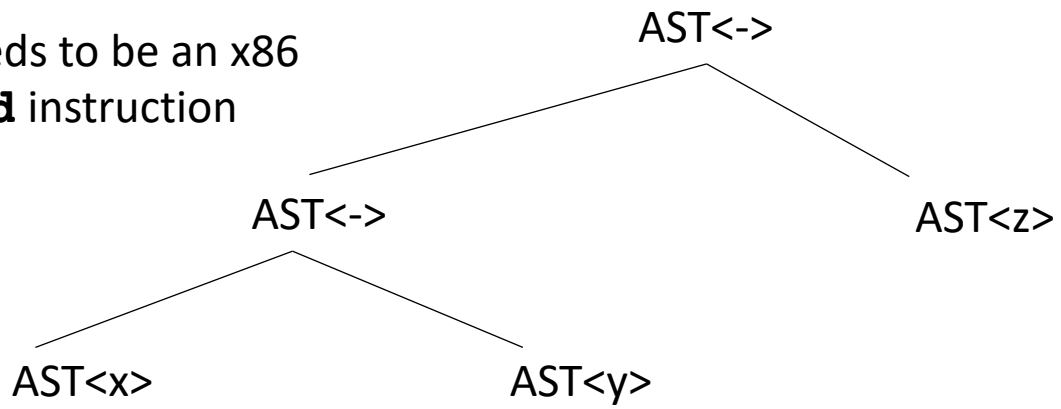
Evaluate an AST by doing a post order traversal

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

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

needs to be an x86 **addss** instruction

needs to be an x86 **add** instruction



Lets do some experiments.

What should 5 + 5.0 be?

Is this all?

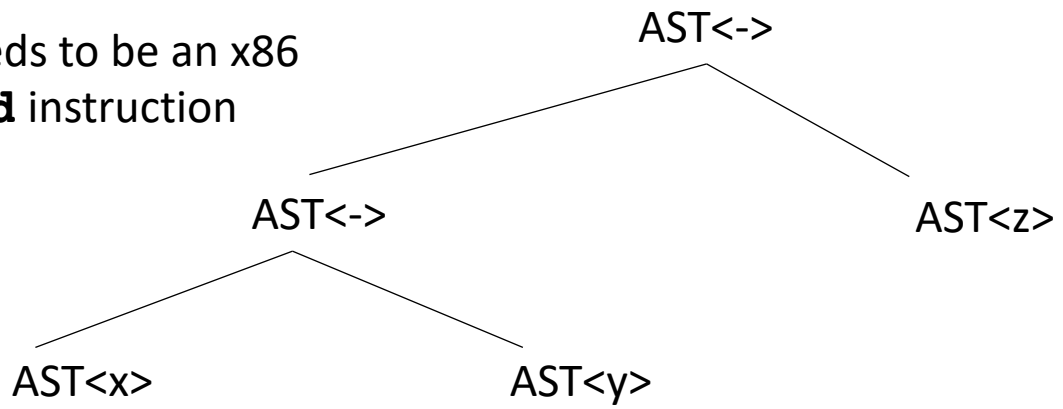
Evaluate an AST by doing a post order traversal

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

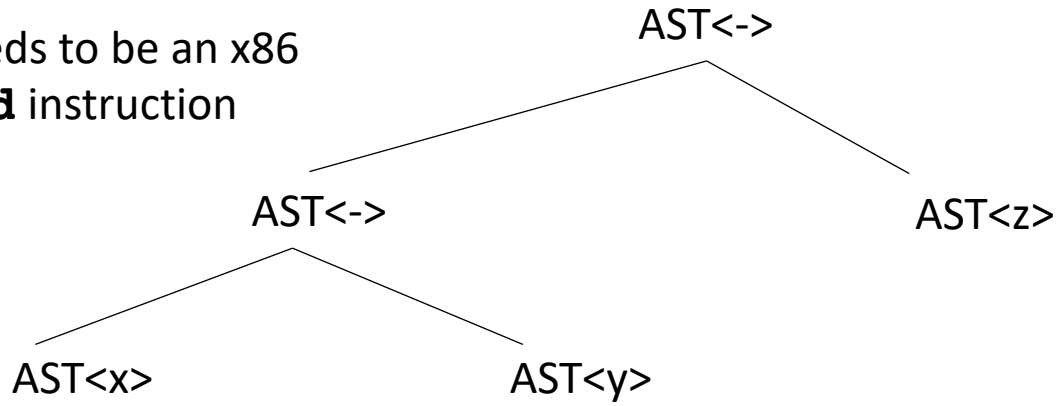
Evaluate an AST by doing a post order traversal

```
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 add them!

Is this all?

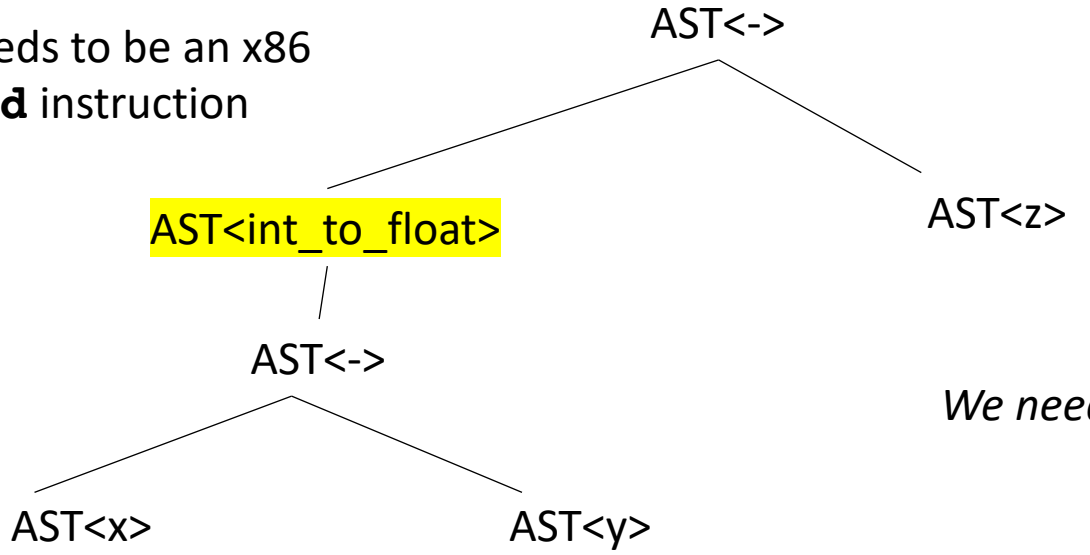
Evaluate an AST by doing a post order traversal

```
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

- 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:

- Base types in the language:
  - ints
  - chars
  - strings
  - floats
  - bool

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

Type systems

Considerations:

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

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:

- Base types:
  - 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:

- Base types:
  - 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:

- 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

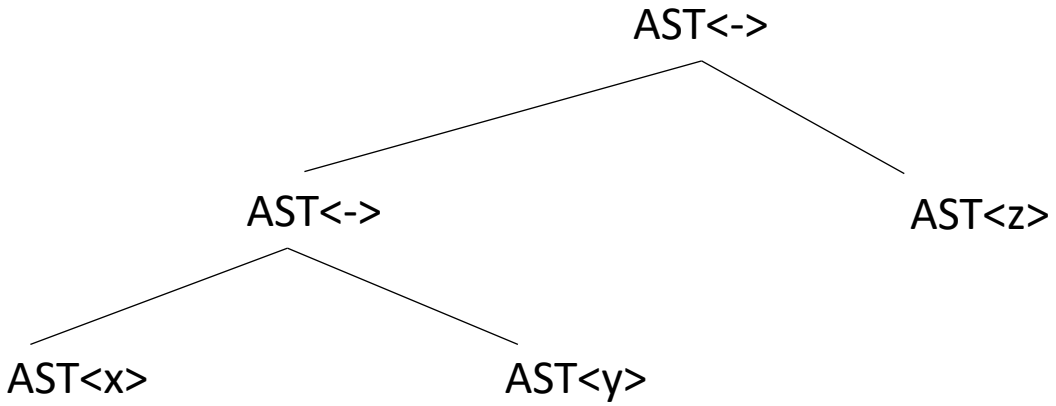
- How to combine types in expressions:
  - int and float?
  - int and char?
  - int and bool?

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

Type checking on an AST

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

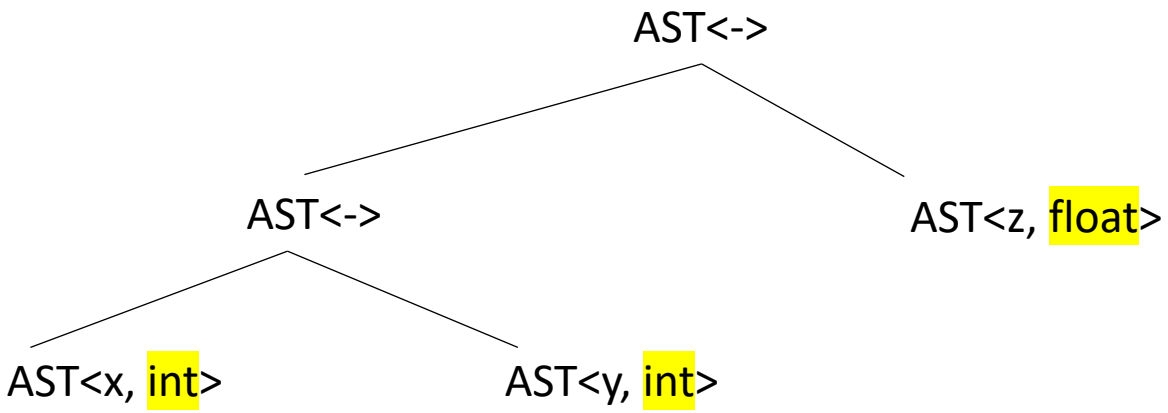
each node additionally gets a type



Type checking on an AST

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

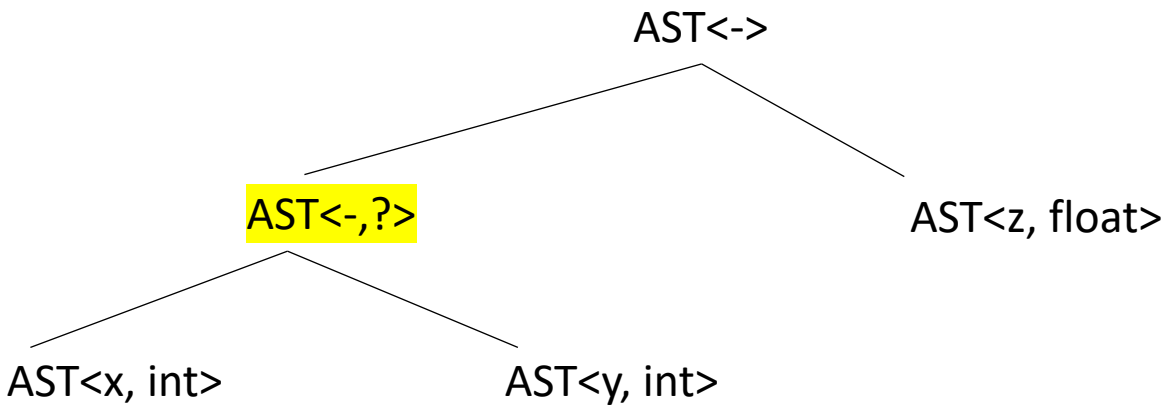
each node additionally gets a type
we can get this from the symbol table for the leaves



Type checking on an AST

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

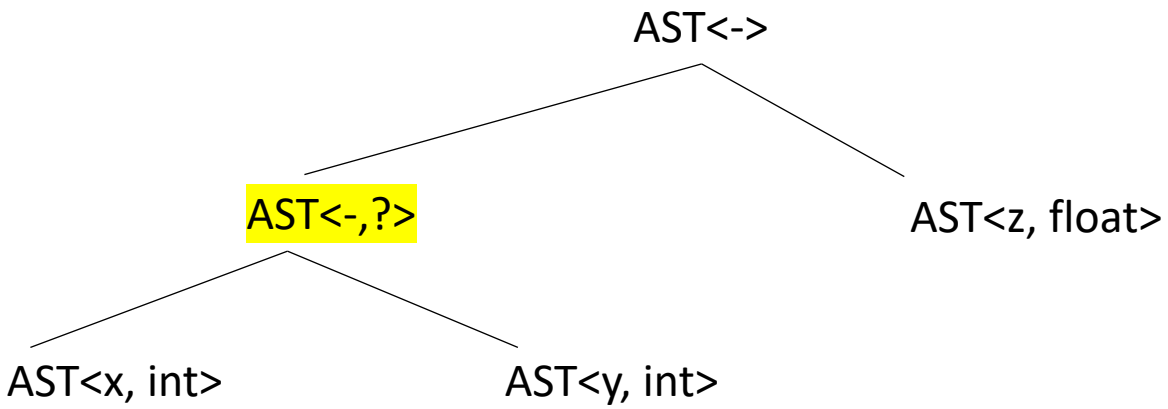
How do we get the type for this one?



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?



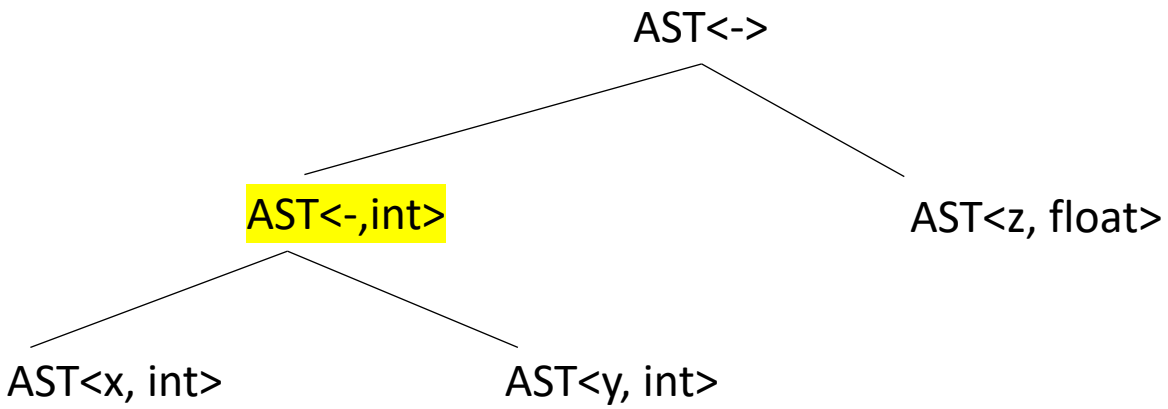
combination rules for subtraction:

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?



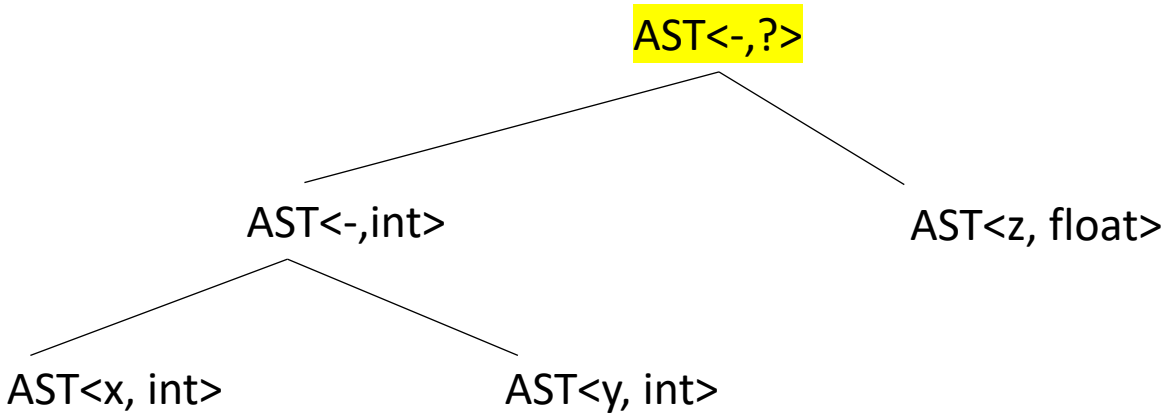
inference rules for subtraction:

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?



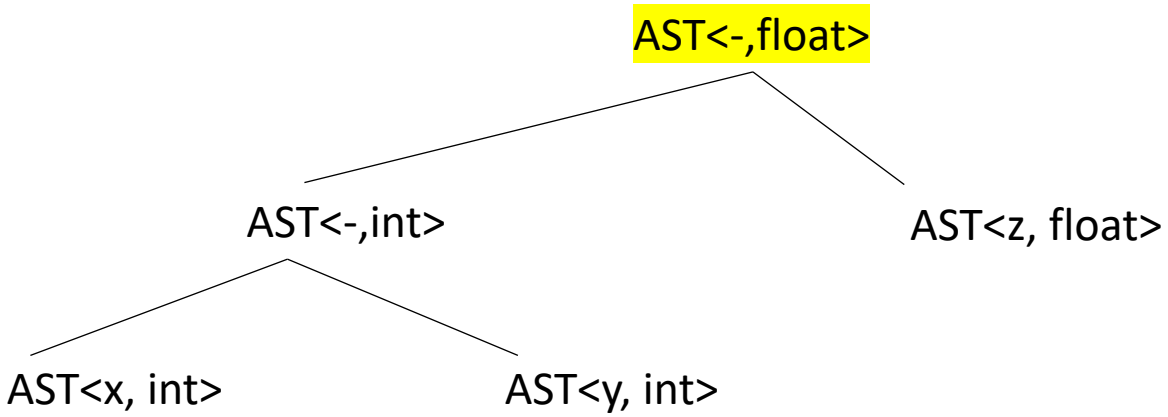
inference rules for subtraction:

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?



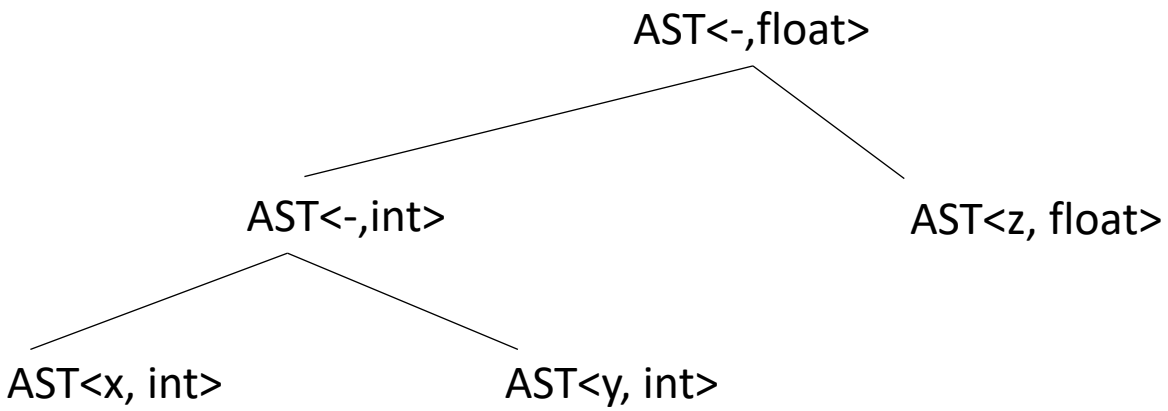
inference rules for subtraction:

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:

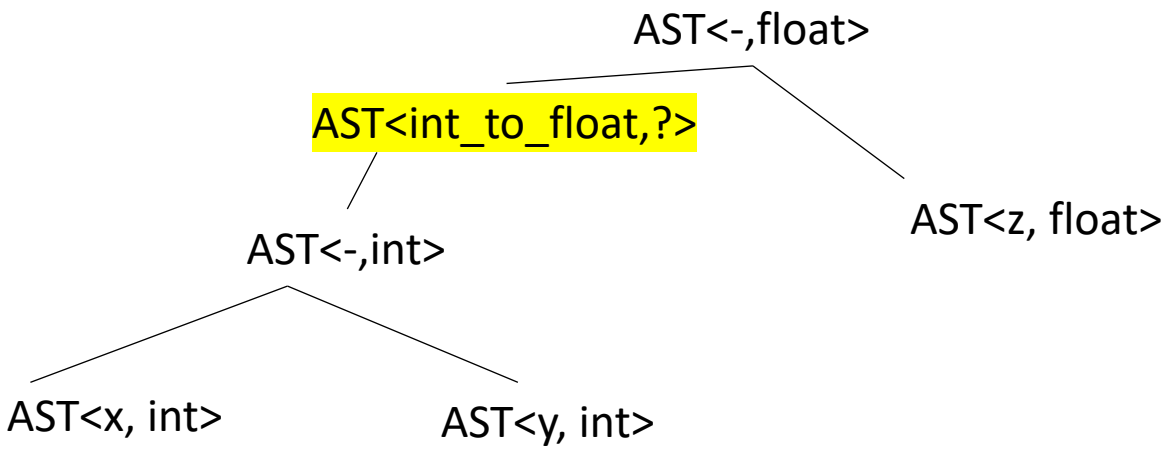
first	second	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

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

See everyone on Monday

- We will discuss implementing type inference on Monday