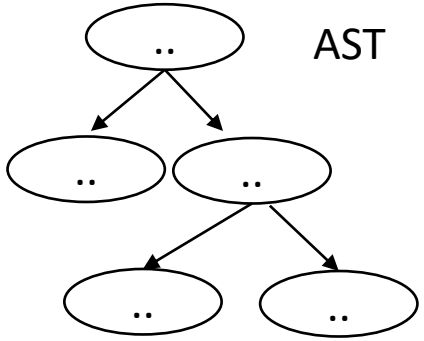
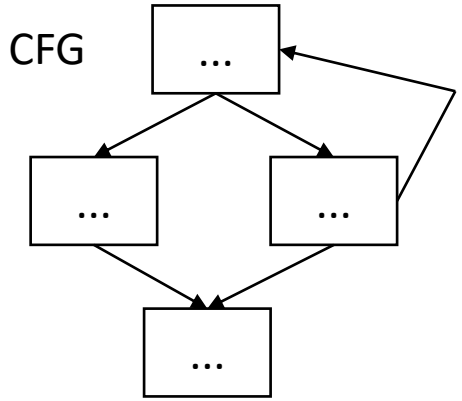
CSE110A: Compilers

May 1, 2024



Topics:

- *Module 3: Intermediate representations*
  - *Intro to intermediate representations*
  - *ASTs*



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 2 was due on 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

- First 2 classes of module 3

Quiz

Quiz

Error messages about undeclared variables are printed by

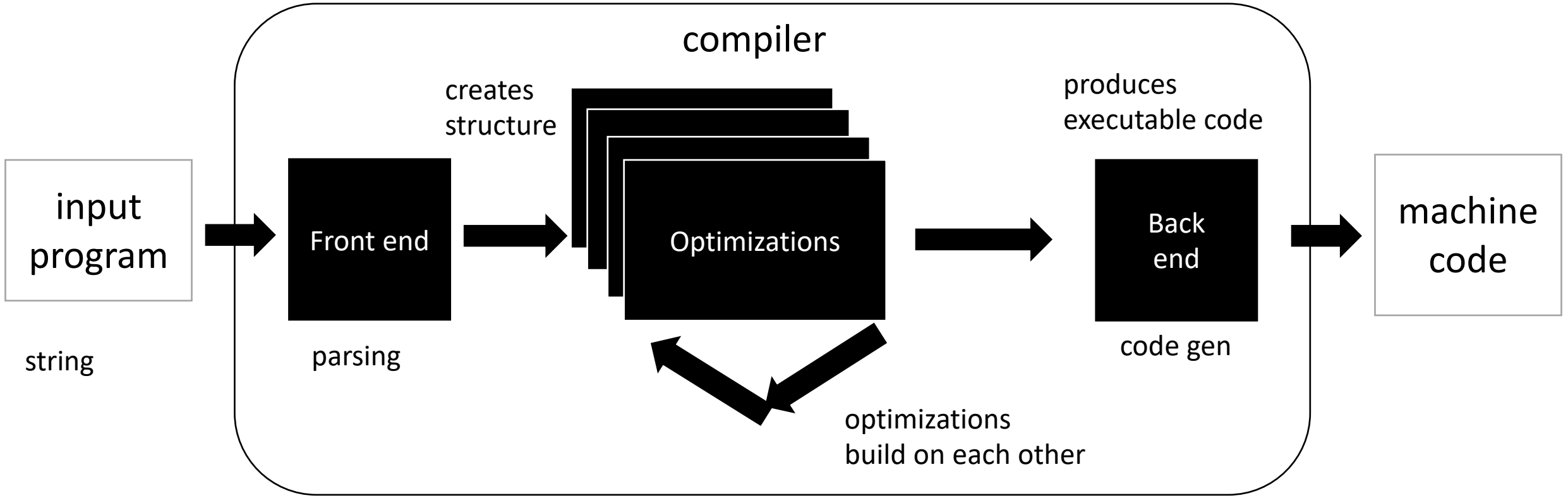
- ○ The scanner
- ○ The parser
- ○ Symbol Table
- ○ Backend

Quiz

Thinking about scoping rules for Python and C: write a few sentences about the differences in how each language utilizes a symbol table to track variables and catch errors related to undeclared variables. Feel free to run a few experiments if you aren't sure how each language tracks variables. Did you find anything that surprised you?

New Module!

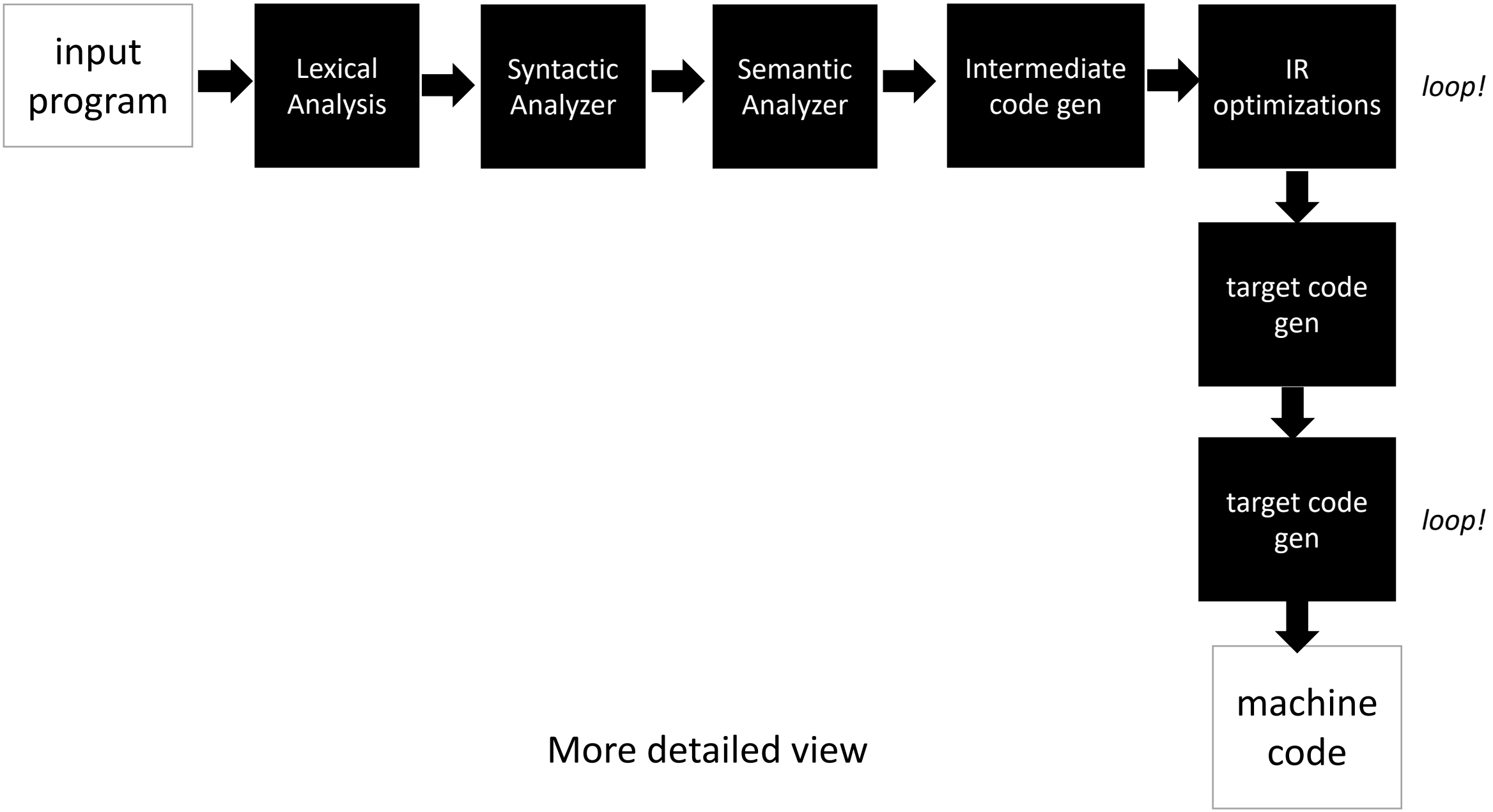
Compiler Architecture



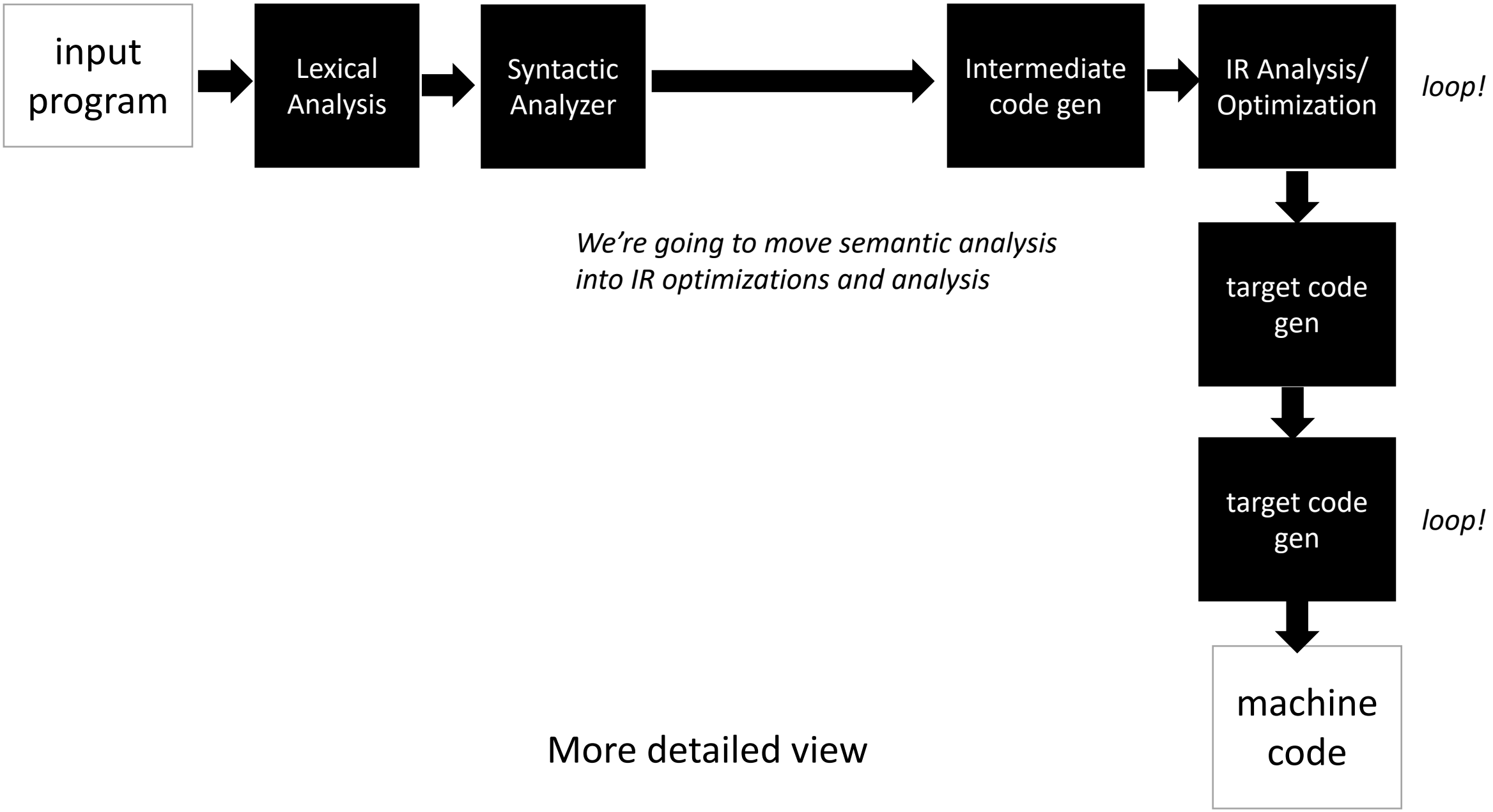
Medium detailed view

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

More detailed view

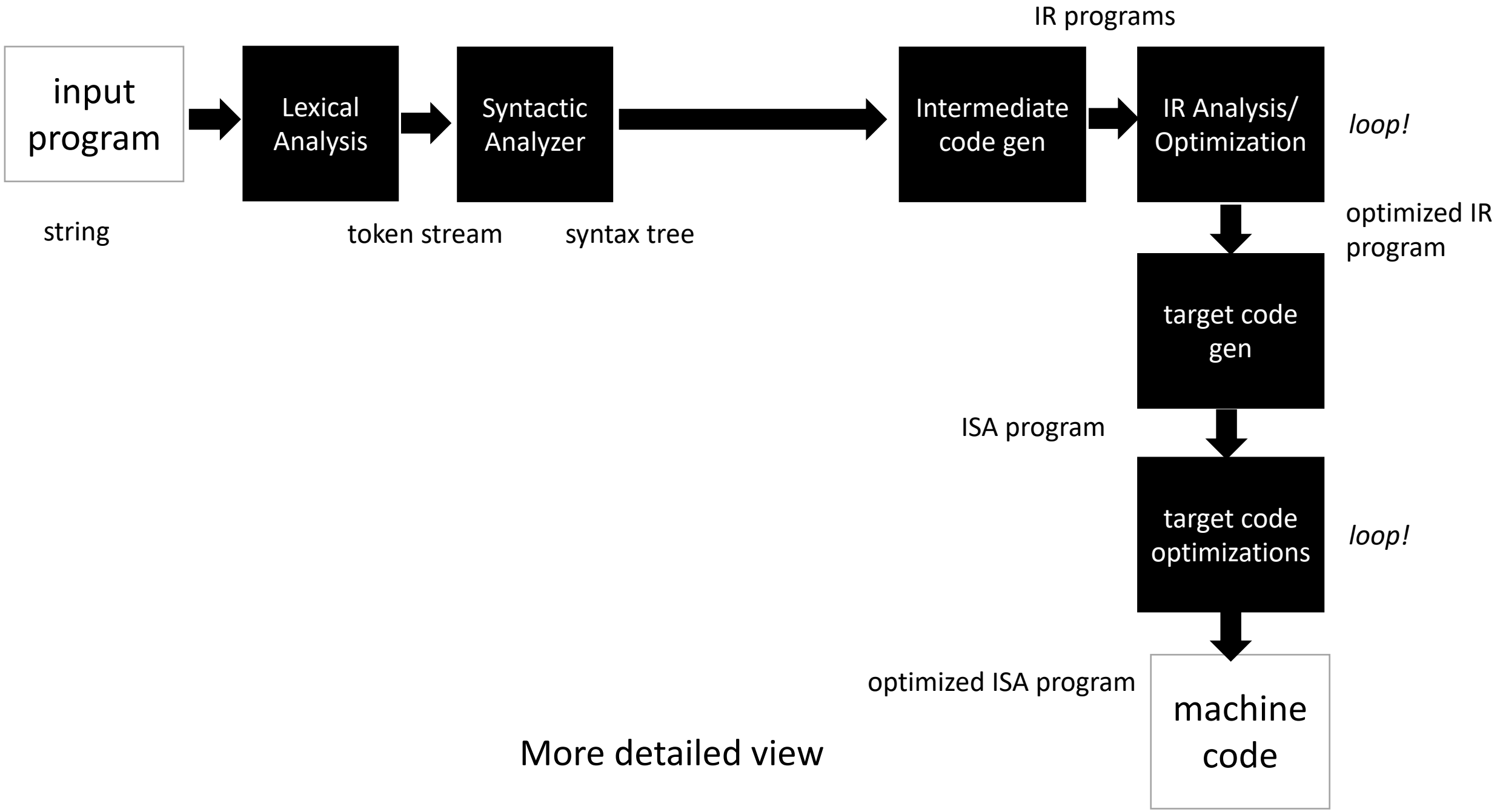


More detailed view

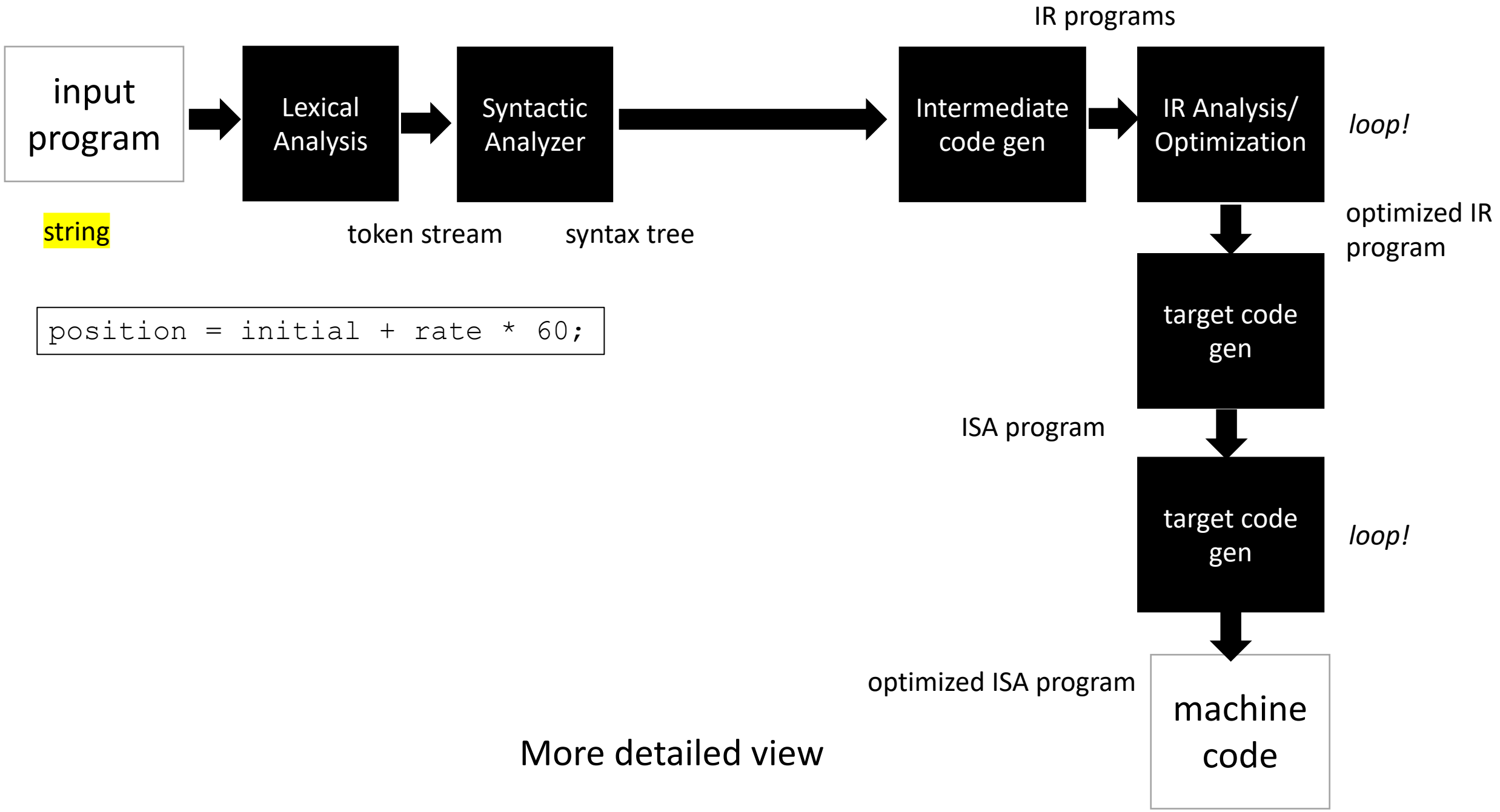


We're going to move semantic analysis into IR optimizations and analysis

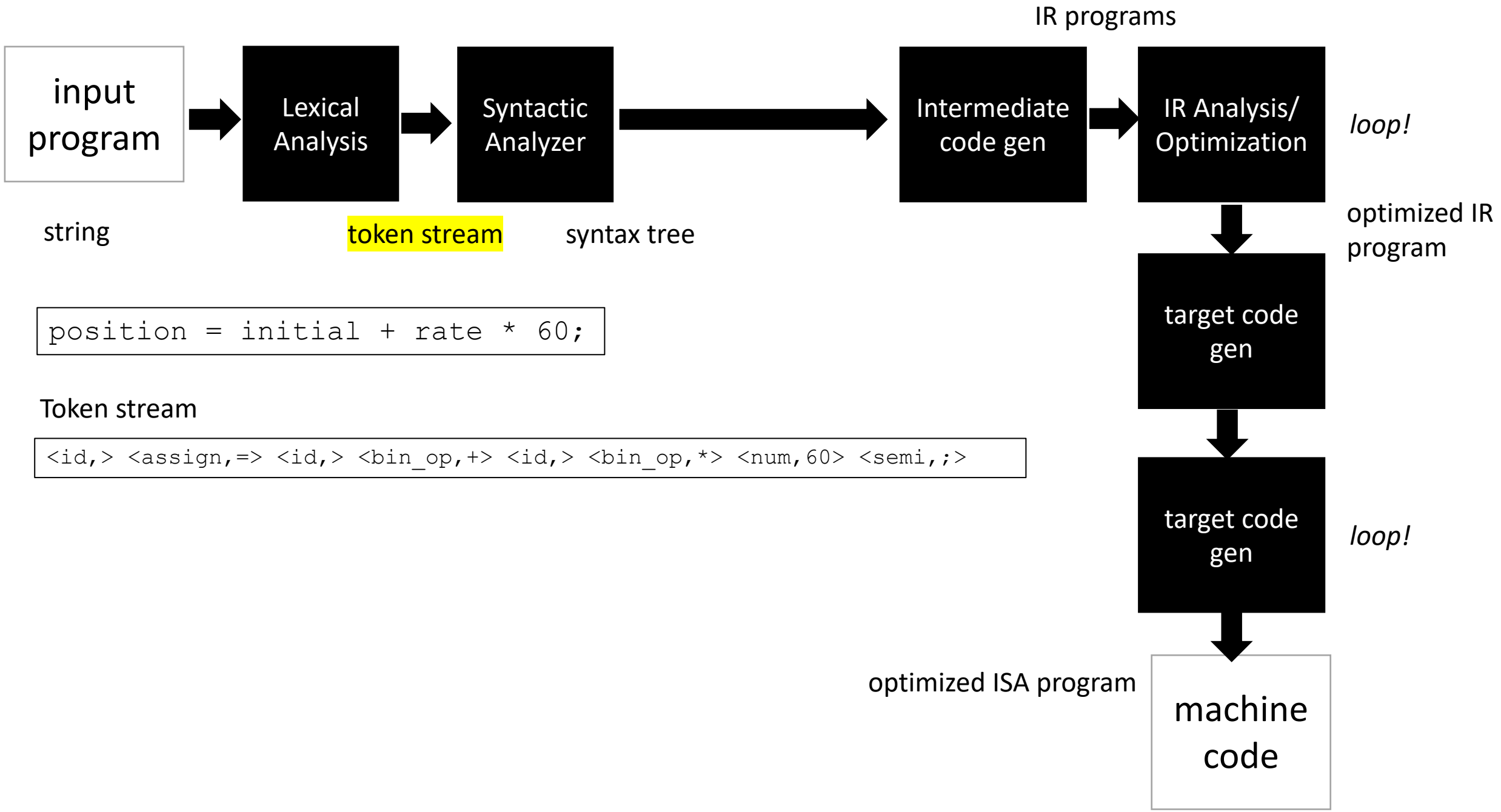
More detailed view



More detailed view



More detailed view

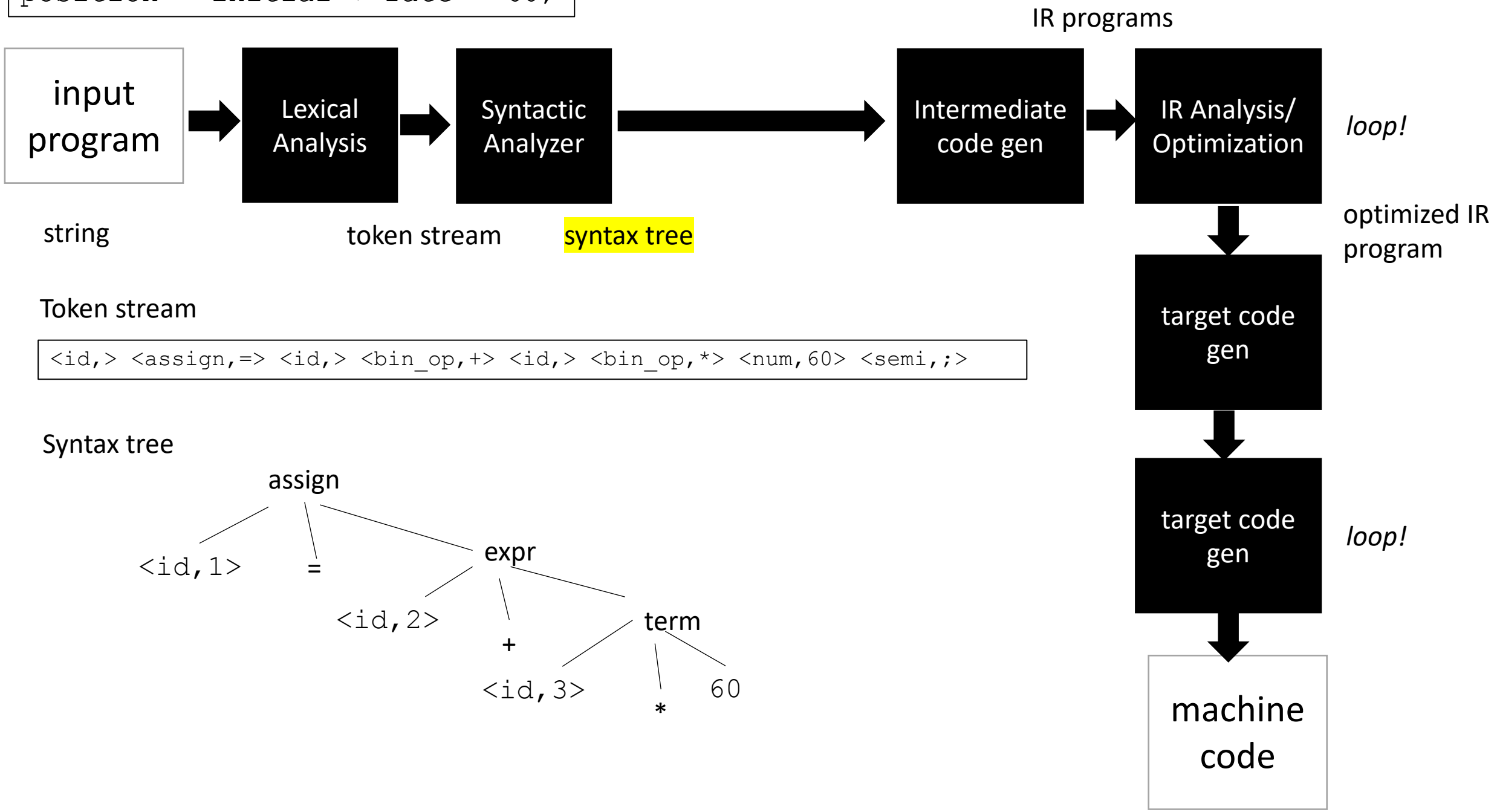



```
position = initial + rate * 60;
```

Token stream

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

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

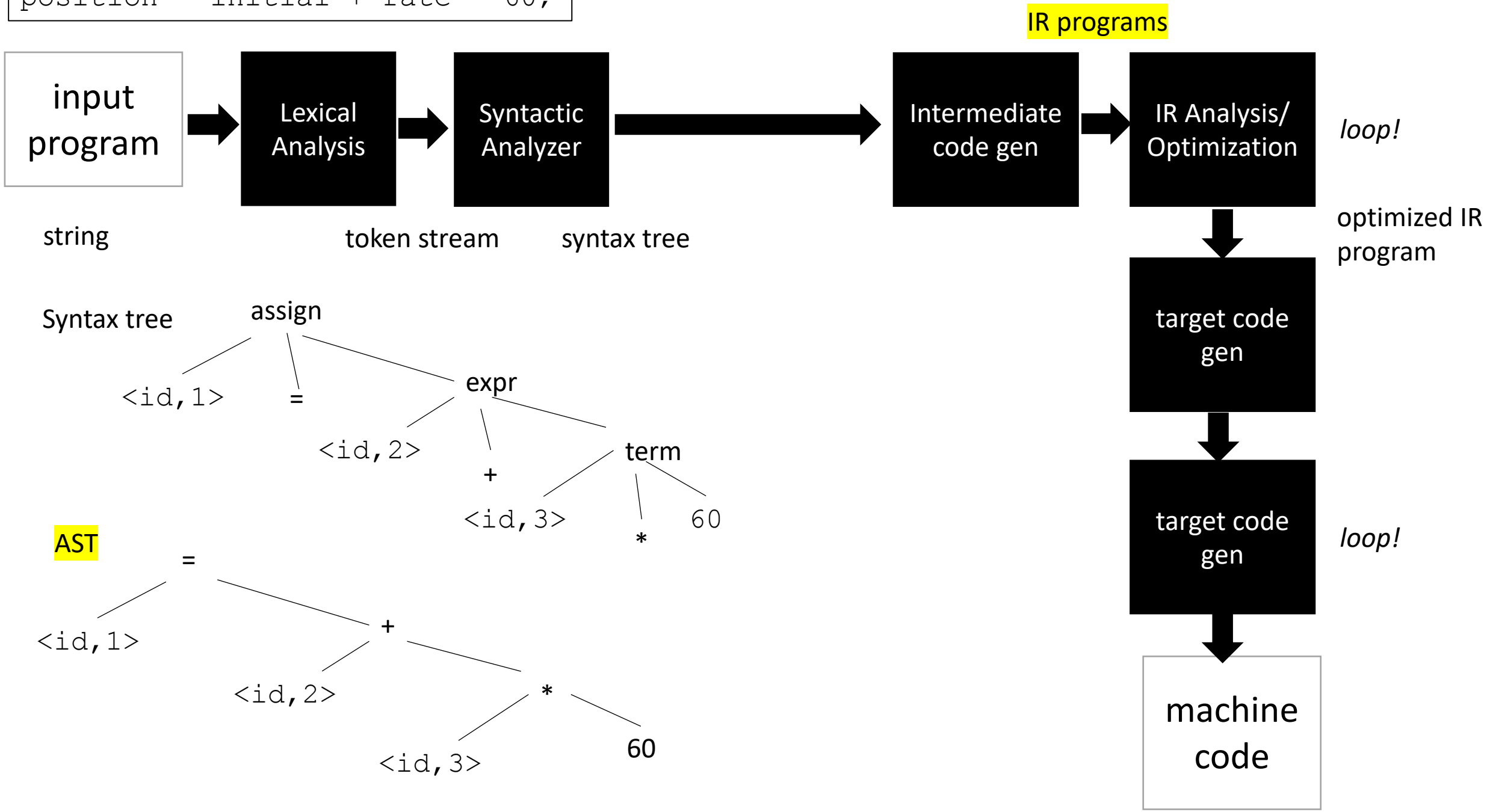


Token stream

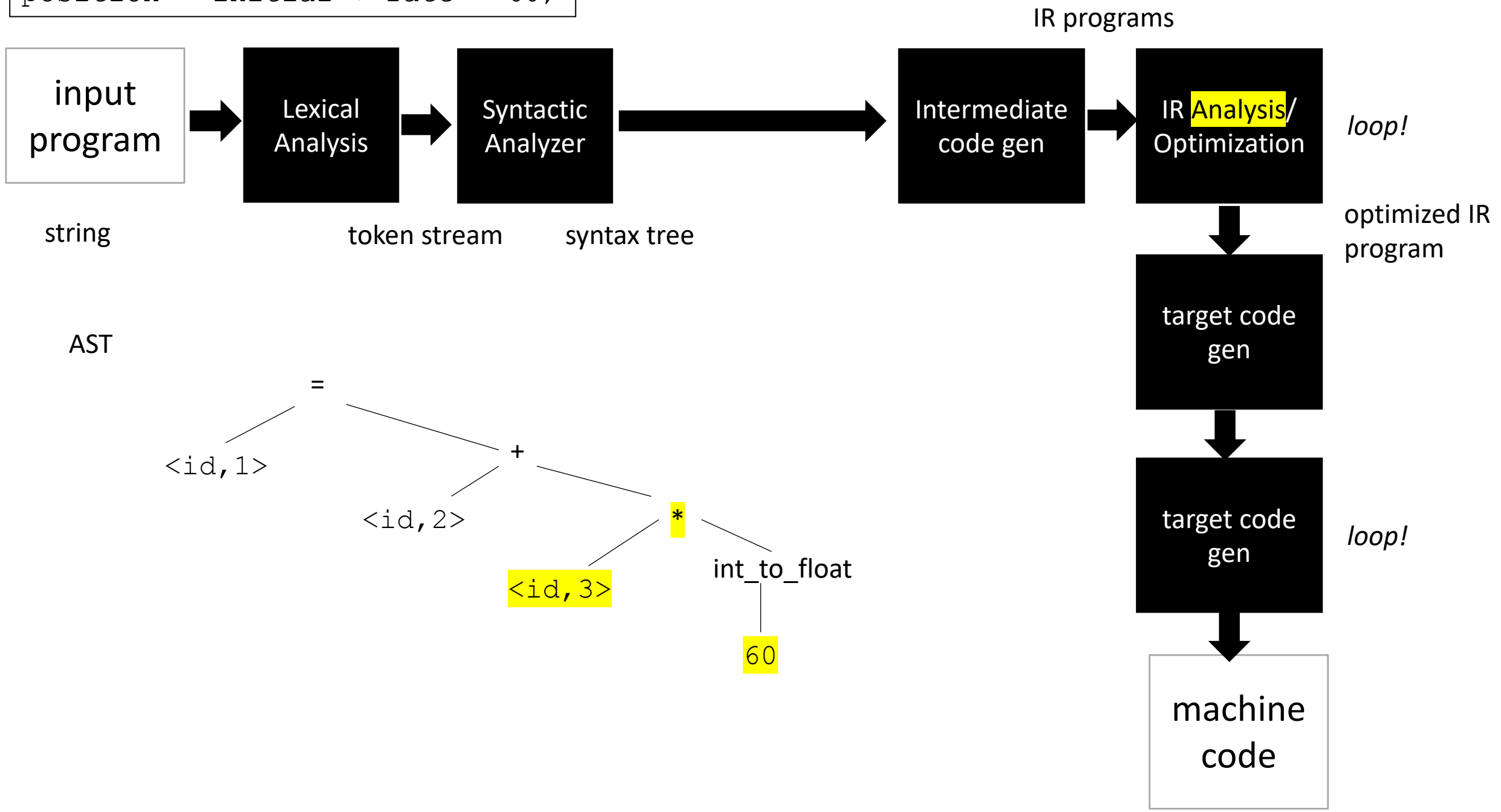
```
<id,> <assign,=> <id,> <bin_op,+> <id,> <bin_op,*> <num,60> <semi,;>
```

Syntax tree

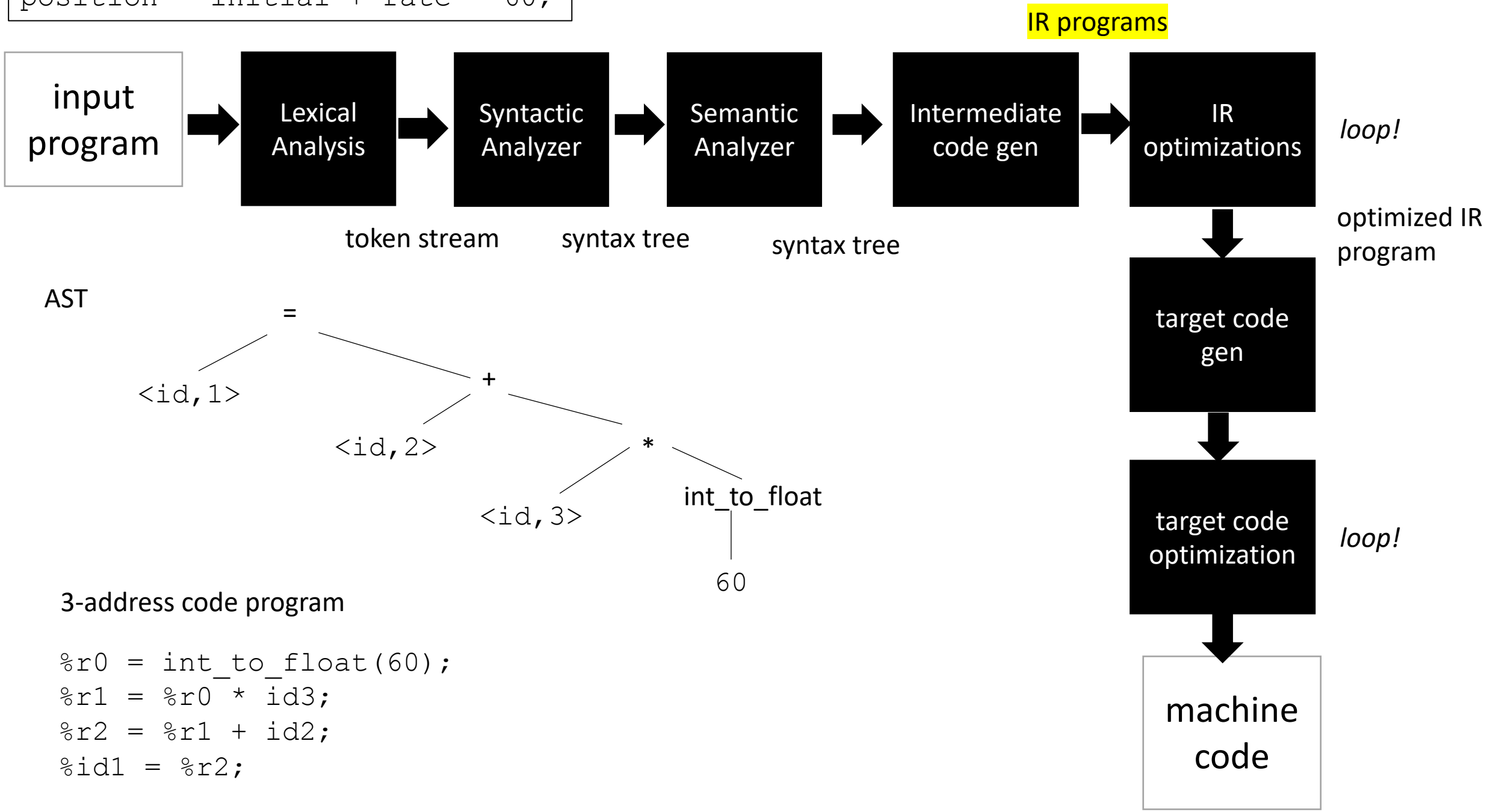
```
position = initial + rate * 60;
```



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



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



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

Example Clang and LLVM

- Clang:
  - a parser for C/++
  - compiles down to an IR: LLVM IR

- LLVM (low-level virtual machine)
  - An IR and specification
  - unlimited registers
  - simple expressions

Example Clang and LLVM

Quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```

use flag: -emit-llvm

Intermediate representations

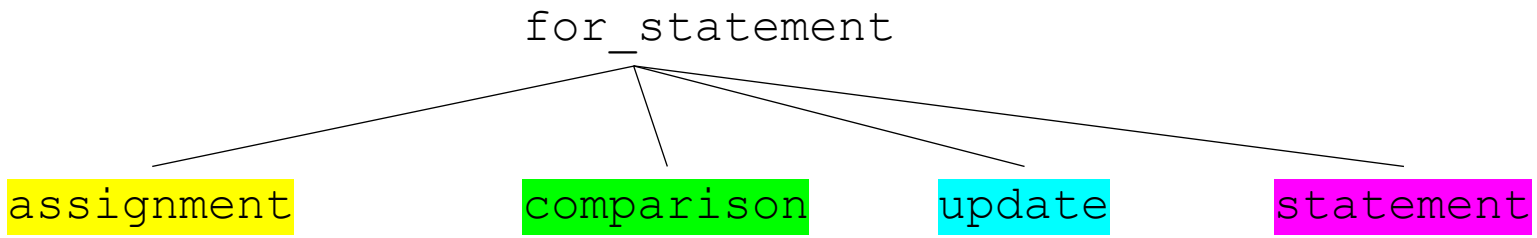
- Several forms:
  - tree - abstract syntax tree
  - graphs - control flow graph
  - linear program - 3 address code

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

Example: loop unrolling

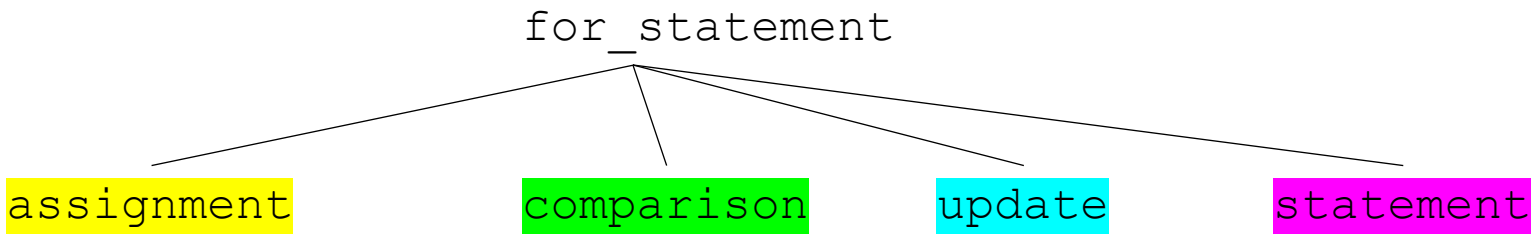
```
for (i = 0; i < 100; i = i + 1) {
  print(i);
  x = x + 1;
}
```

Example: loop unrolling




```
for (i = 0; i < 100; i = i +1) {
   x = x + 1;
}
```

Example: loop unrolling




```
for (i = 0; i < 100; i = i + 1) {
   x = x + 1;
}
```

Check:

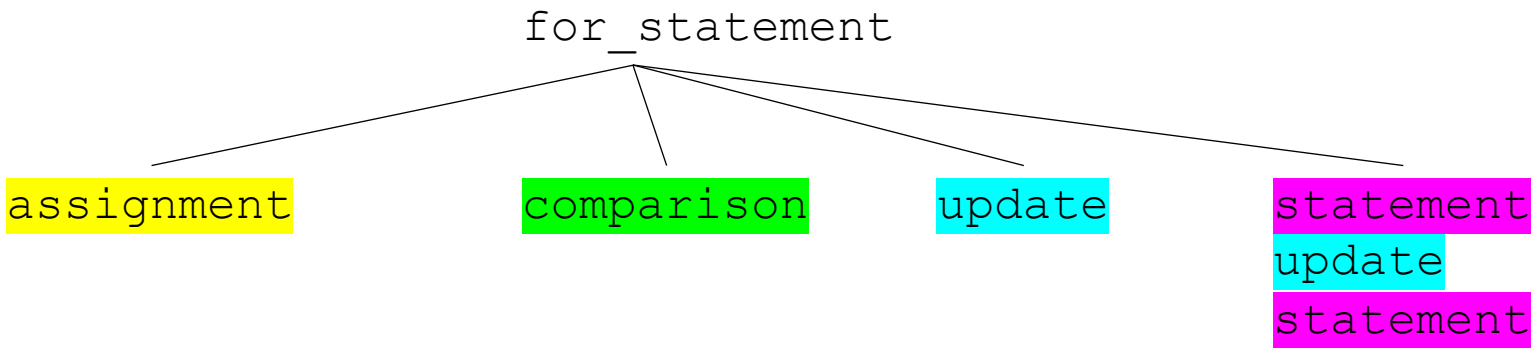
1. Find iteration variable by examining assignment, comparison and update.

2. found i

3. check that statement doesn't change i.

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

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

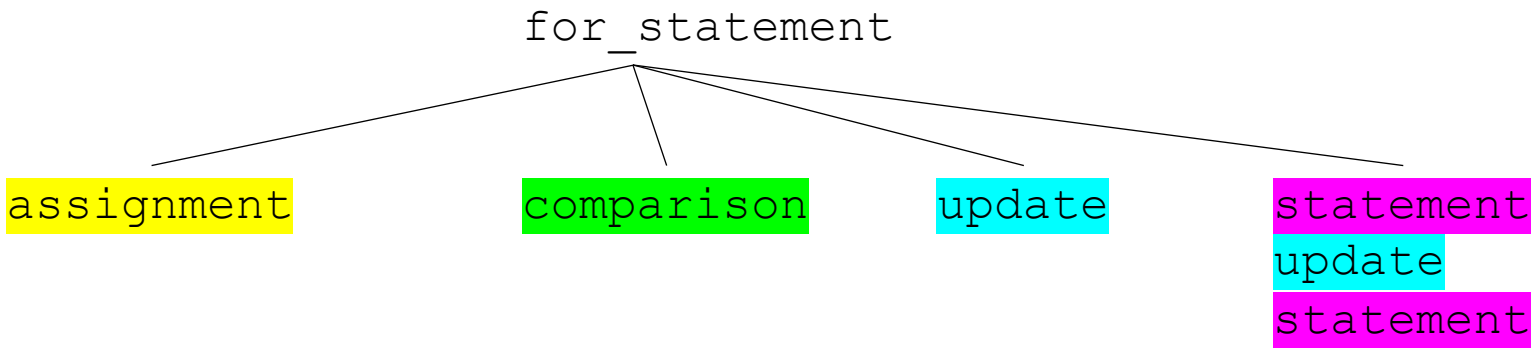
3. check that statement doesn't change i.

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

Perform optimization

copy statement and put an update before it

Example: loop unrolling




```
for (i = 0; i < 100; i = i + 1) {
   x = x + 1;
   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

Example: loop unrolling

```
br label %3, !dbg !22

3: ; preds = %13, %0
%4 = load i32, ptr %1, align 4, !dbg !23
%5 = icmp slt i32 %4, 100, !dbg !25
br i1 %5, label %6, label %16, !dbg !26

6: ; preds = %3
%7 = load i32, ptr %2, align 4, !dbg !27
%8 = add nsw i32 %7, 1, !dbg !29
store i32 %8, ptr %2, align 4, !dbg !30
%9 = load i32, ptr %1, align 4, !dbg !31
%10 = add nsw i32 %9, 1, !dbg !32
store i32 %10, ptr %1, align 4, !dbg !33
%11 = load i32, ptr %2, align 4, !dbg !34
%12 = add nsw i32 %11, 1, !dbg !35
store i32 %12, ptr %2, align 4, !dbg !36
br label %13, !dbg !37

13: ; preds = %6
%14 = load i32, ptr %1, align 4, !dbg !38
%15 = add nsw i32 %14, 1, !dbg !39
store i32 %15, ptr %1, align 4, !dbg !40
br label %3, !dbg !41, !llvm.loop !42
```

LLVM IR for the for loop. Much harder to analyze!

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

Example: common subexpression elimination

```
z = x + y;
a = b + c;
d = x + y;
```

Can this be optimized?

Example: common subexpression elimination

```
z = x + y;
a = b + c;
d = x + y;
```

Can this be optimized?

```
z = x + y;
a = b + c;
d = z;
```

remove redundant addition

Easy to do this optimization when code is a low level form like this

Our first IR: abstract syntax tree

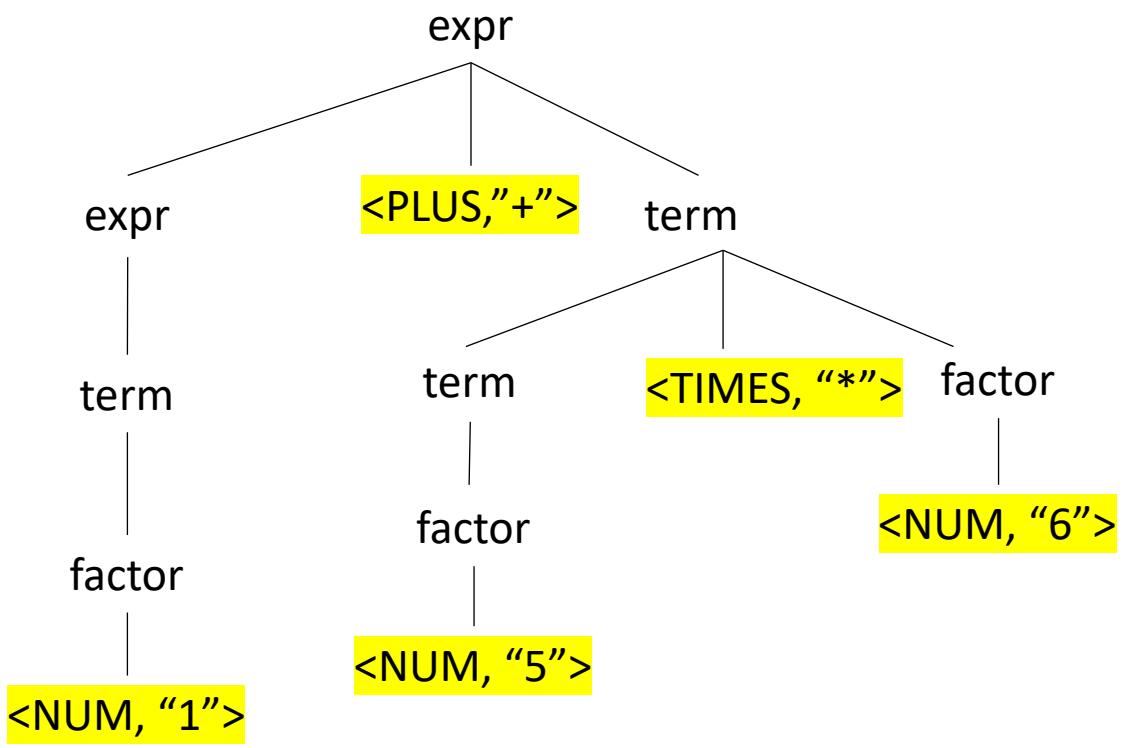
- One step away from parse trees
- Great representation for expressions
- Natural representation to apply type checking/inference
- Can view in clang with: -Xclang -ast-dump

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`	`: LPAREN expr RPAREN` `| NUM`

`input: 1+5*6`

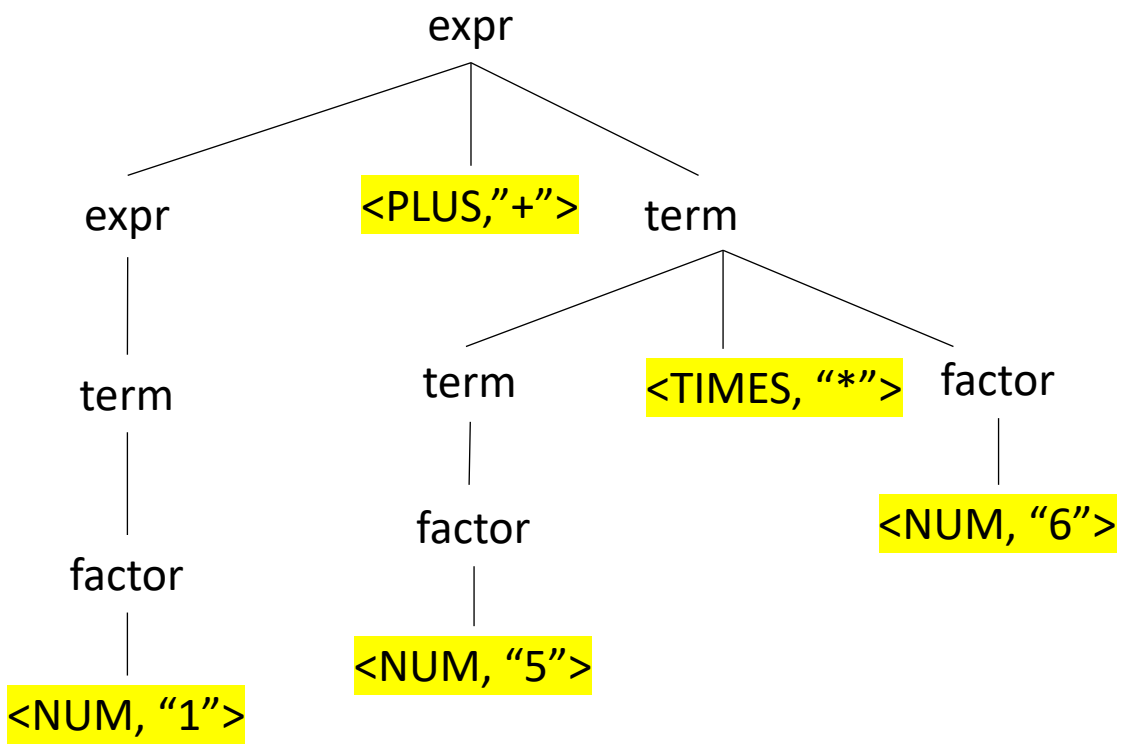


What is an AST?

We'll start by looking at a parse tree:

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

`input: 1+5*6`



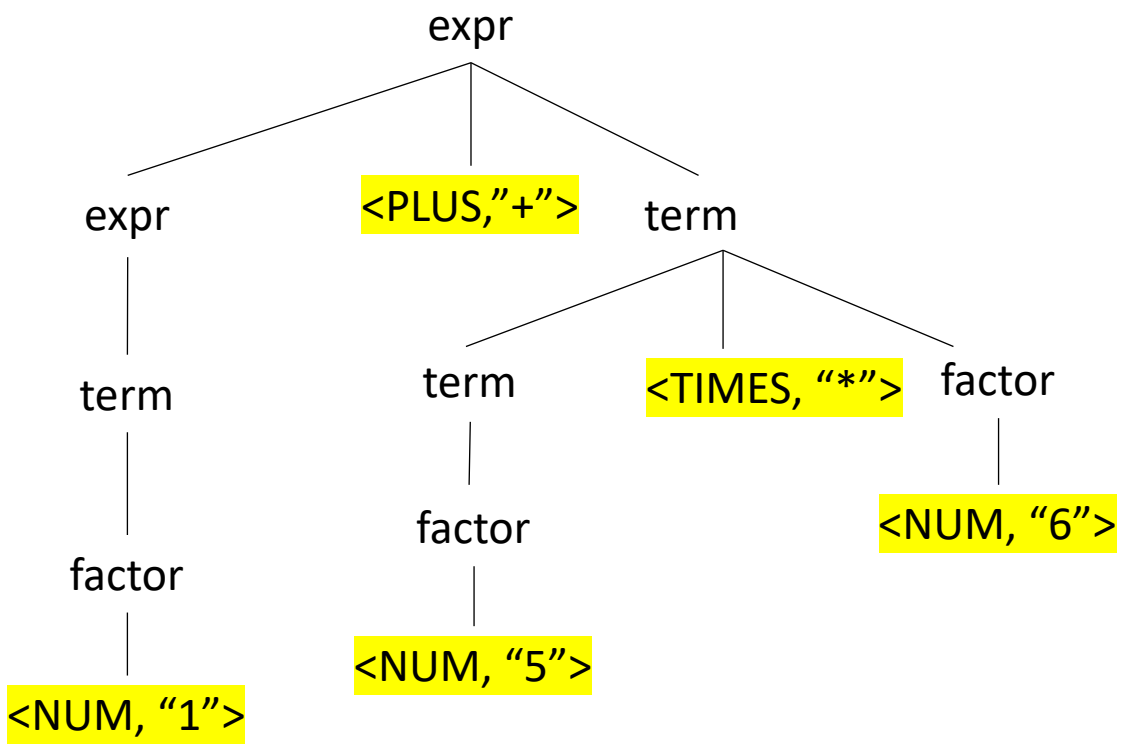
What are leaves?

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`	`: LPAREN expr RPAREN` `| NUM`

`input: 1+5*6`



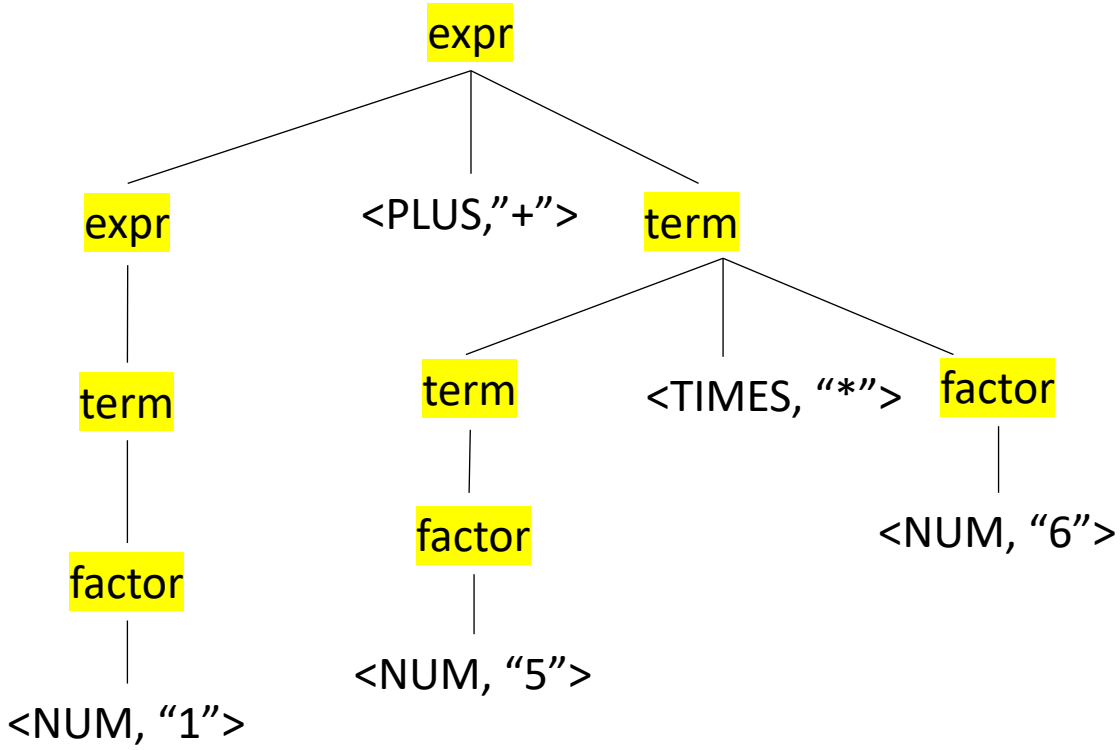
What are leaves? lexemes

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`	`: LPAREN expr RPAREN` `| NUM`

`input: 1+5*6`



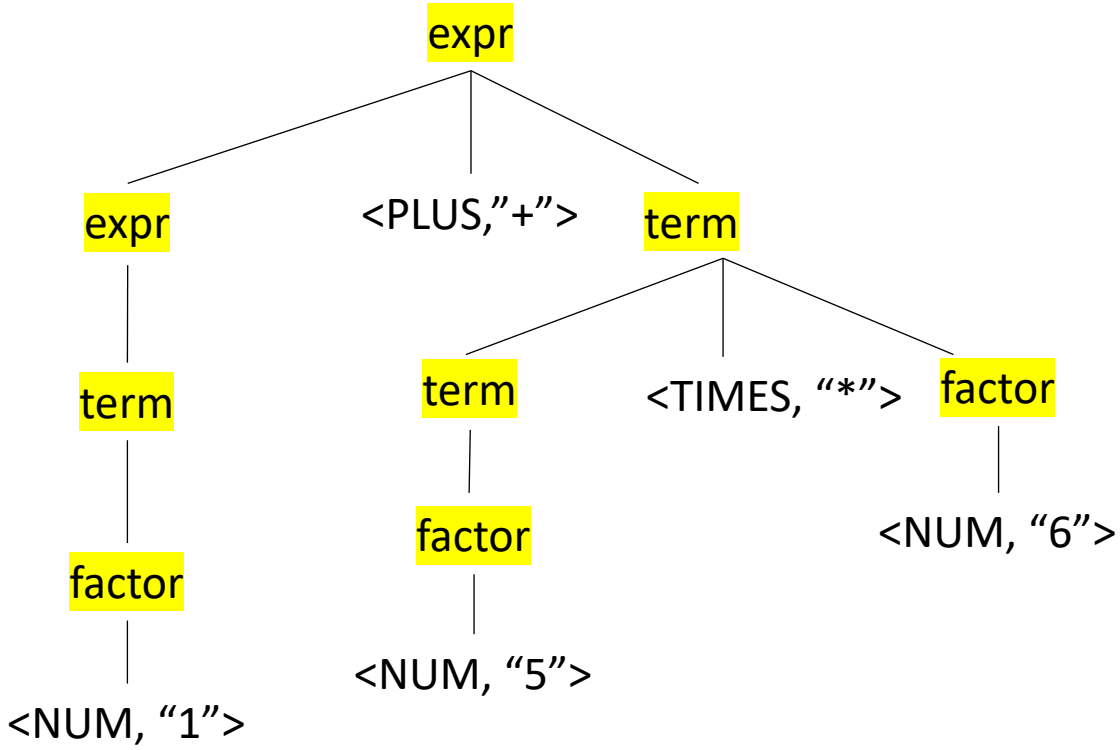
What are nodes?

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`	`: LPAREN expr RPAREN` `| NUM`

`input: 1+5*6`



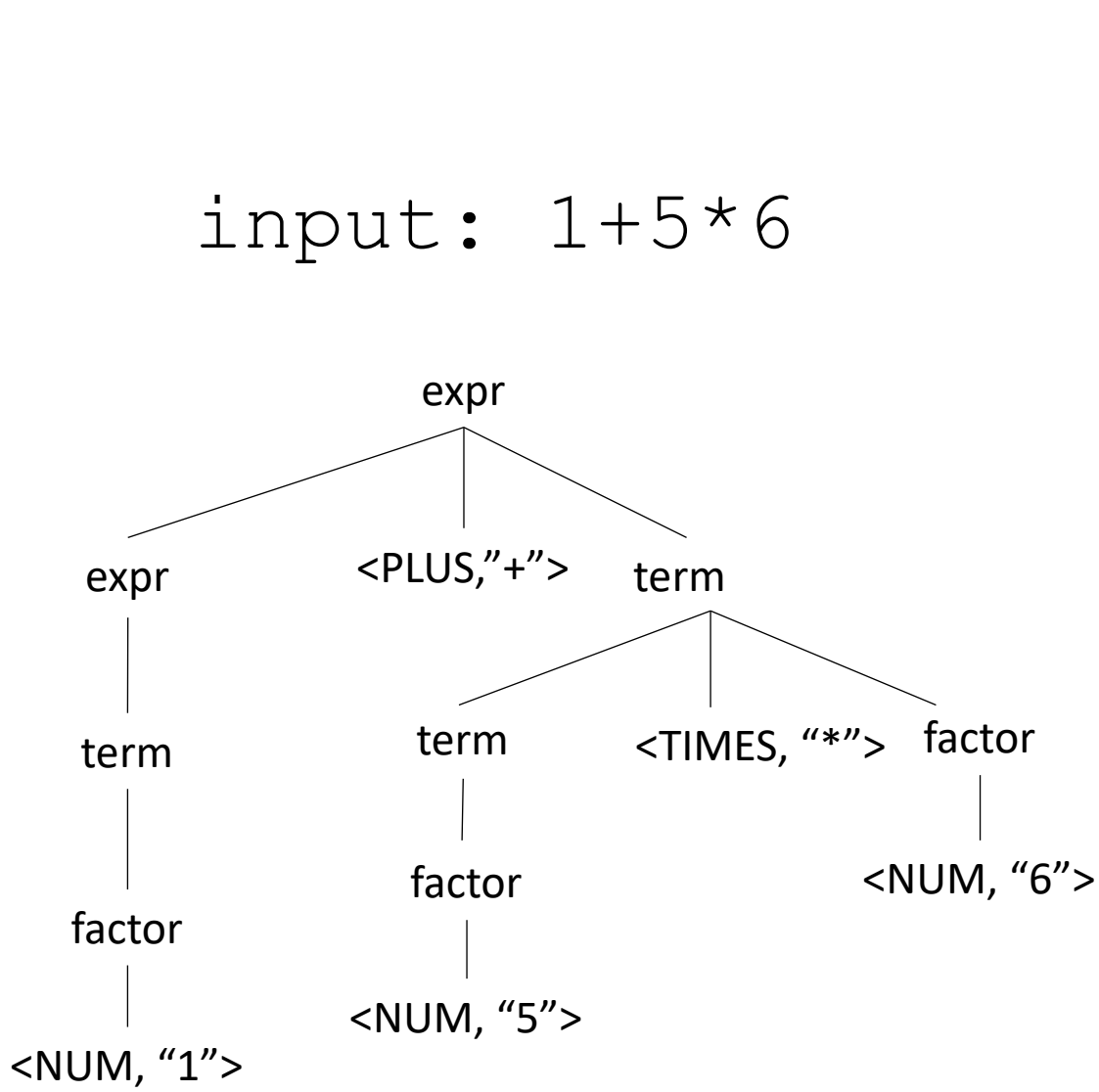
What are nodes? non-terminals

What is an AST?

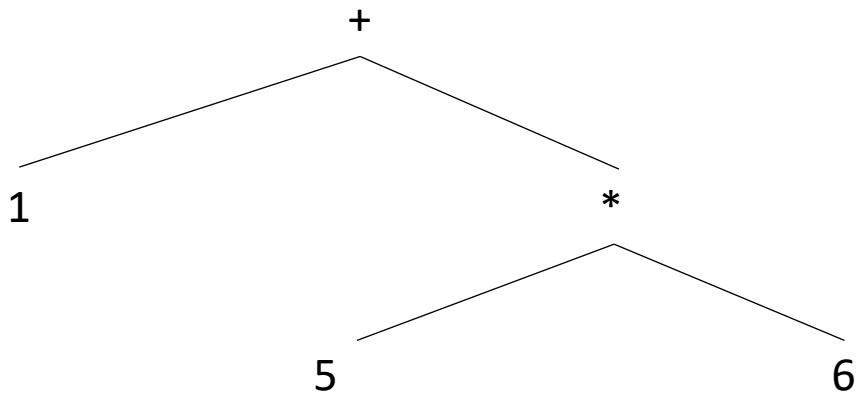
Parse trees are defined by the grammar

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

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



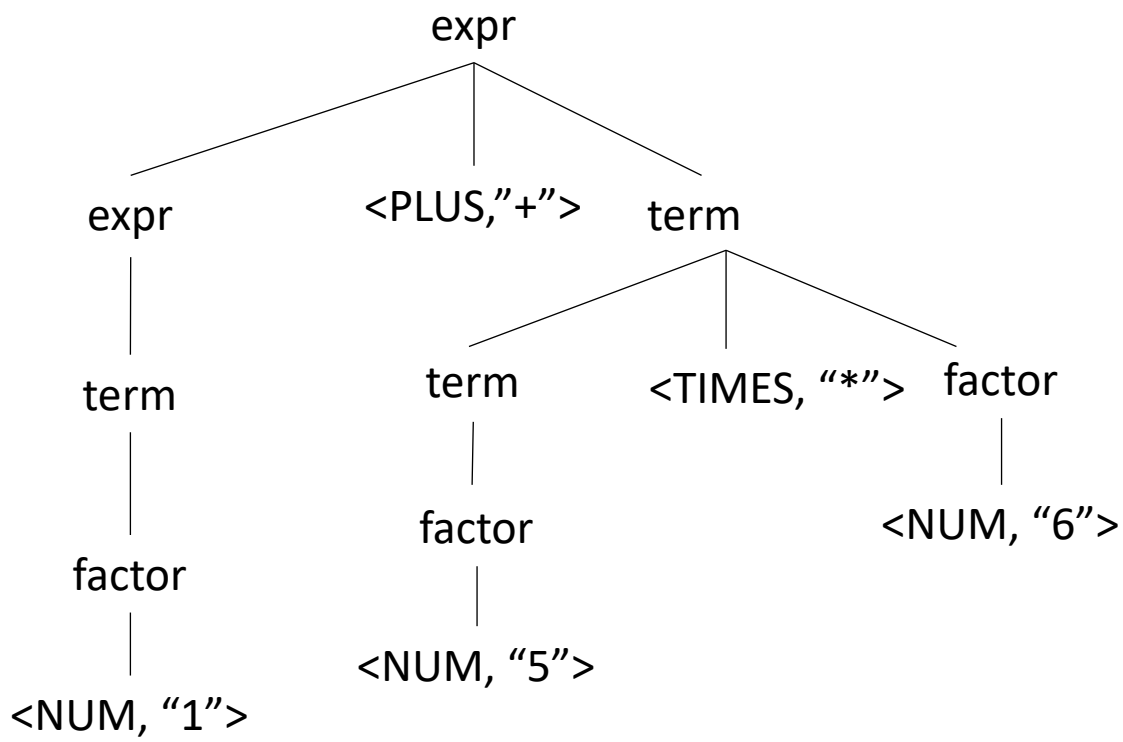
What is an AST?



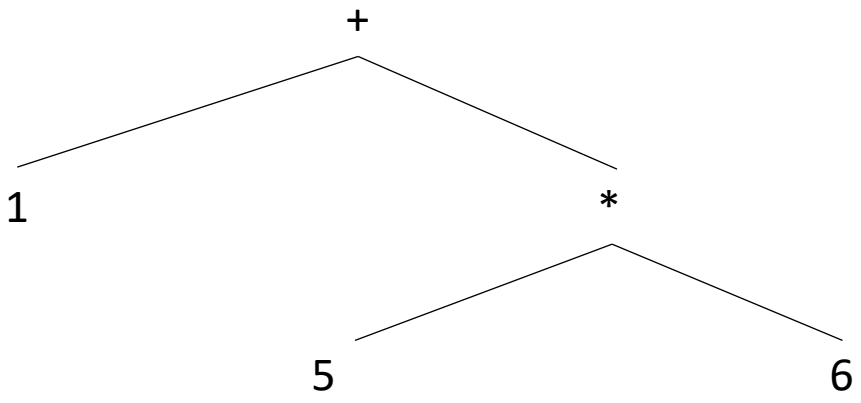
AST

What are some differences?

`input: 1+5*6`



What is an AST?

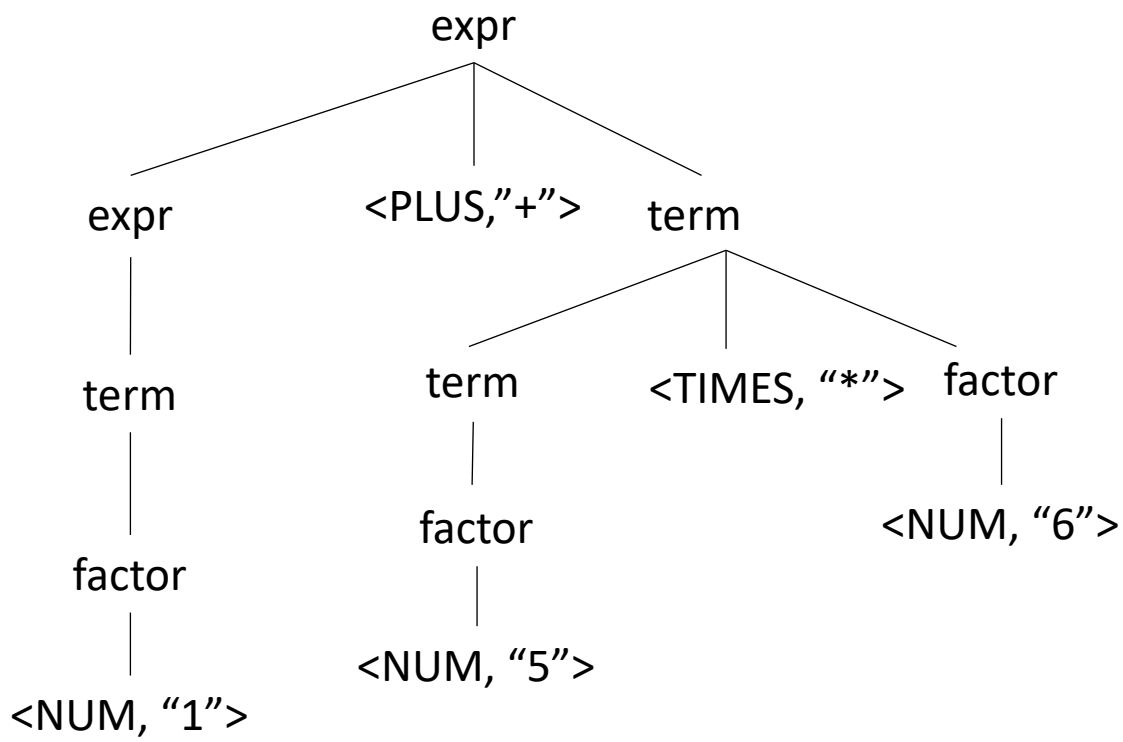


AST

What are some differences?

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

`input: 1+5*6`

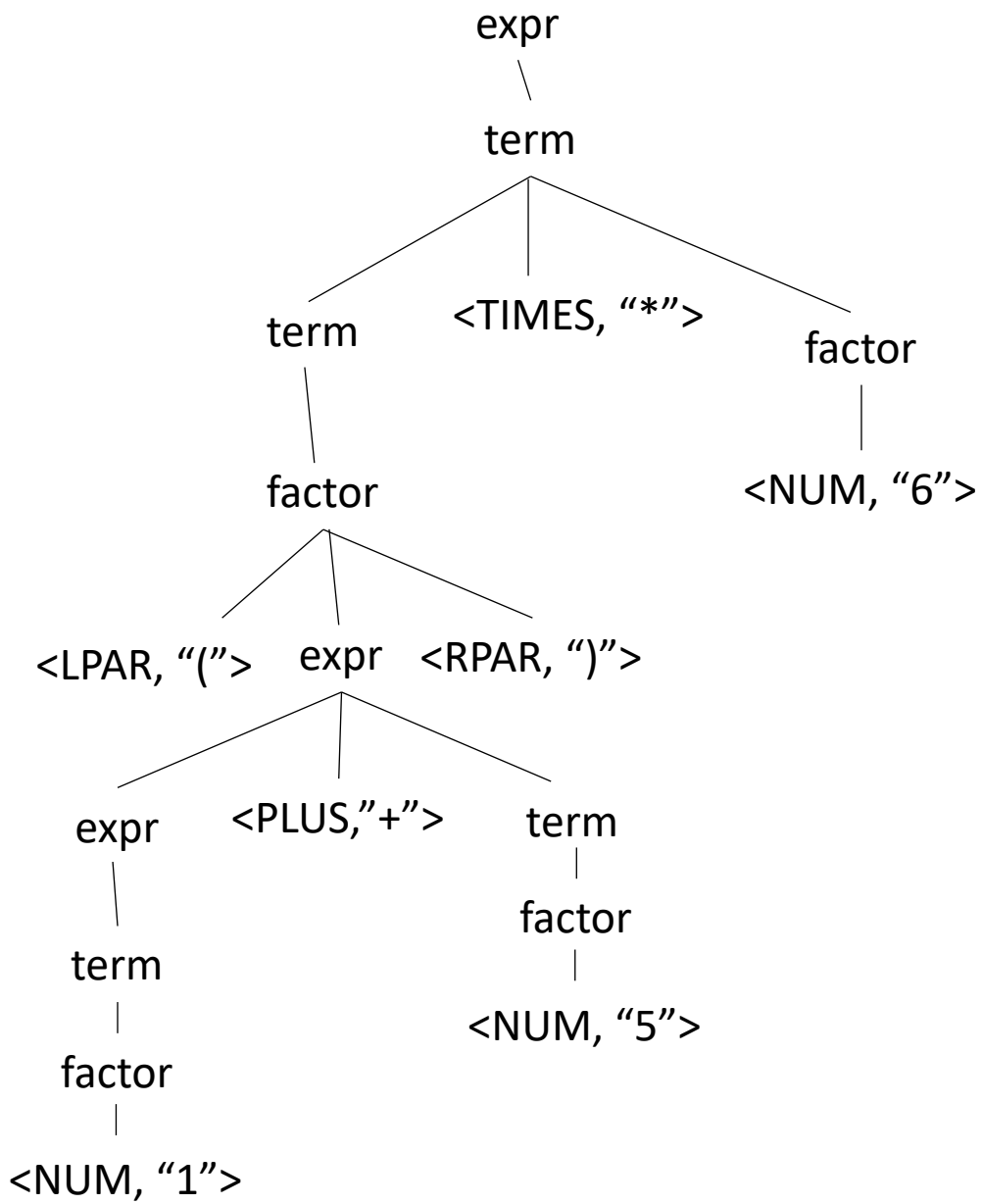


Example

what happens to ()s in an AST?

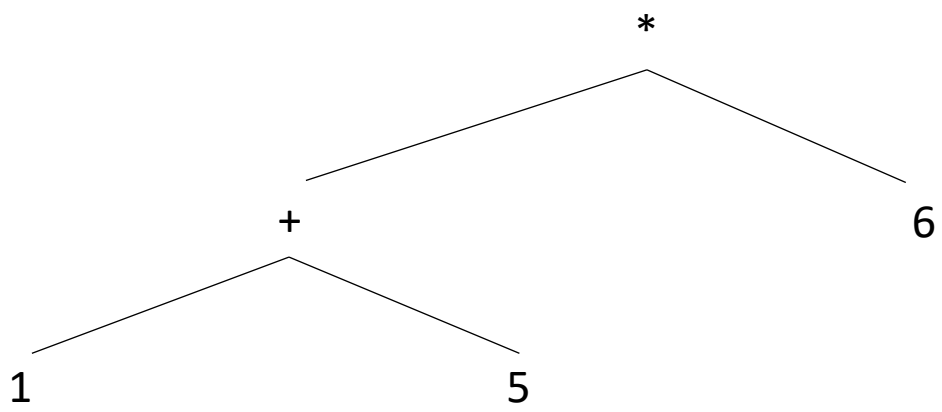
Operator	Name	Productions
+	`expr`	`: expr PLUS term` `| term`
*	`term`	`: term TIMES factor` `| factor`
()	`factor`	`: LPAR expr RPAR` `| NUM`

`input: (1+5)*6`



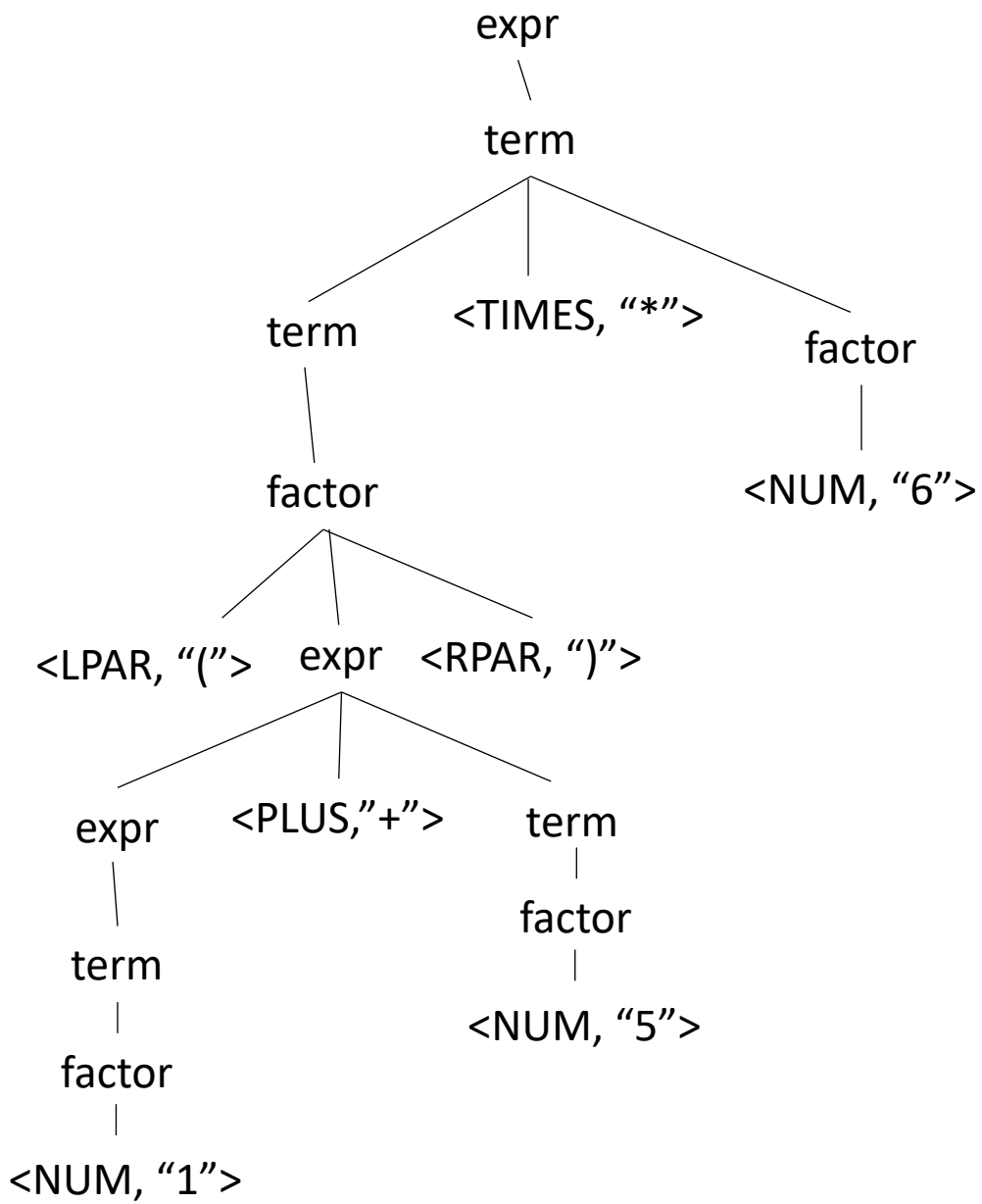
Example

what happens to ()s in an AST?



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

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 a parser

Parser actions

- Like token actions: perform an action each time a production option is matched.
- Typically performed after the entire production action is matched
- Can be useful for catching errors early as well.

Example

Say we are matched the statement:

```
int x;
```

- `SymbolTable ST;`

Parser actions would be written like this

$1 $2 $3

declare_statement ::= TYPE ID SEMI

```
{
   ST.insert($2, None);
}
```

result of each symbol. For a terminal it will be the value

always some way to refer to symbol value, e.g. an array

What values get returned from non-terminals?

```
1: Expr   ::= Expr ‘+’ Unit     {print $1}
2:          |   Expr ‘-’ Unit
3:          |   Unit
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    NUM
```

What does this print?

What values get returned from non-terminals?

```
1: Expr   ::= Expr ‘+’ Unit     {print $1; return “expr”}
2:          |   Expr ‘-’ Unit     {return “expr”}
3:          |   Unit             {...}
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    NUM
```

Each production rule needs to return something

What values get returned from non-terminals?

building a calculator

```
1: Expr  ::= Expr ‘+’ Unit     {}
2:       |   Expr ‘-’ Unit     {}
3:       |   Unit              {}
4: Unit  ::= ‘(‘ Expr ‘)’      {}
5:       |    NUM              {}
```

What values get returned from non-terminals?

building a calculator

```
1: Expr  ::= Expr ‘+’ Unit     {return $1 + $3}
2:       |   Expr ‘-’ Unit     {return $1 - $3}
3:       |   Unit              {return $1}
4: Unit  ::= ‘(‘ Expr ‘)’      {return $2}
5:       |    NUM              {return $1}
```

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	{} {} {}

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

```
class ASTLeafNode(ASTNode):
    def __init__(self, value):
        self.value = value


class ASTNumNode(ASTLeafNode):
    def __init__(self, value):
        super().__init__(value)


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

```
class ASTBinOpNode(ASTNode):
    def __init__(self, l_child, r_child):
        self.l_child = l_child
        self.r_child = r_child


class ASTPlusNode(ASTBinOpNode):
    def __init__(self, l_child, r_child):
        super().__init__(l_child,r_child)


class ASTMultNode(ASTBinOpNode):
    def __init__(self, l_child, r_child):
        super().__init__(l_child,r_child)
```

Creating an AST from production rules

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

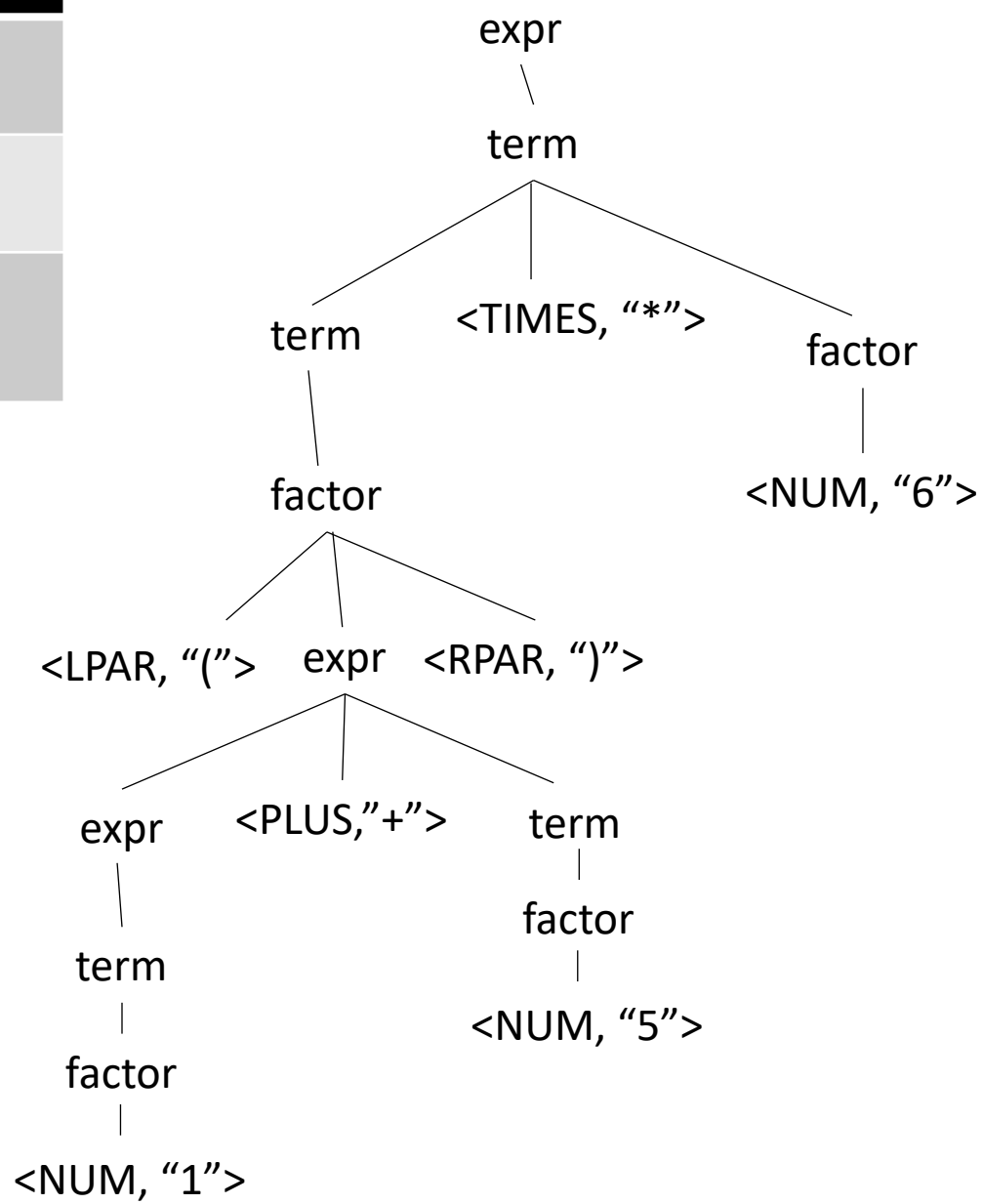
()	`factor`	`: LPAR expr RPAR`
		`| NUM`
		`| ID`
			`{return $2}`
			`{return ASTNumNode($1)}`
			`{return ASTIDNode($1)}`

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

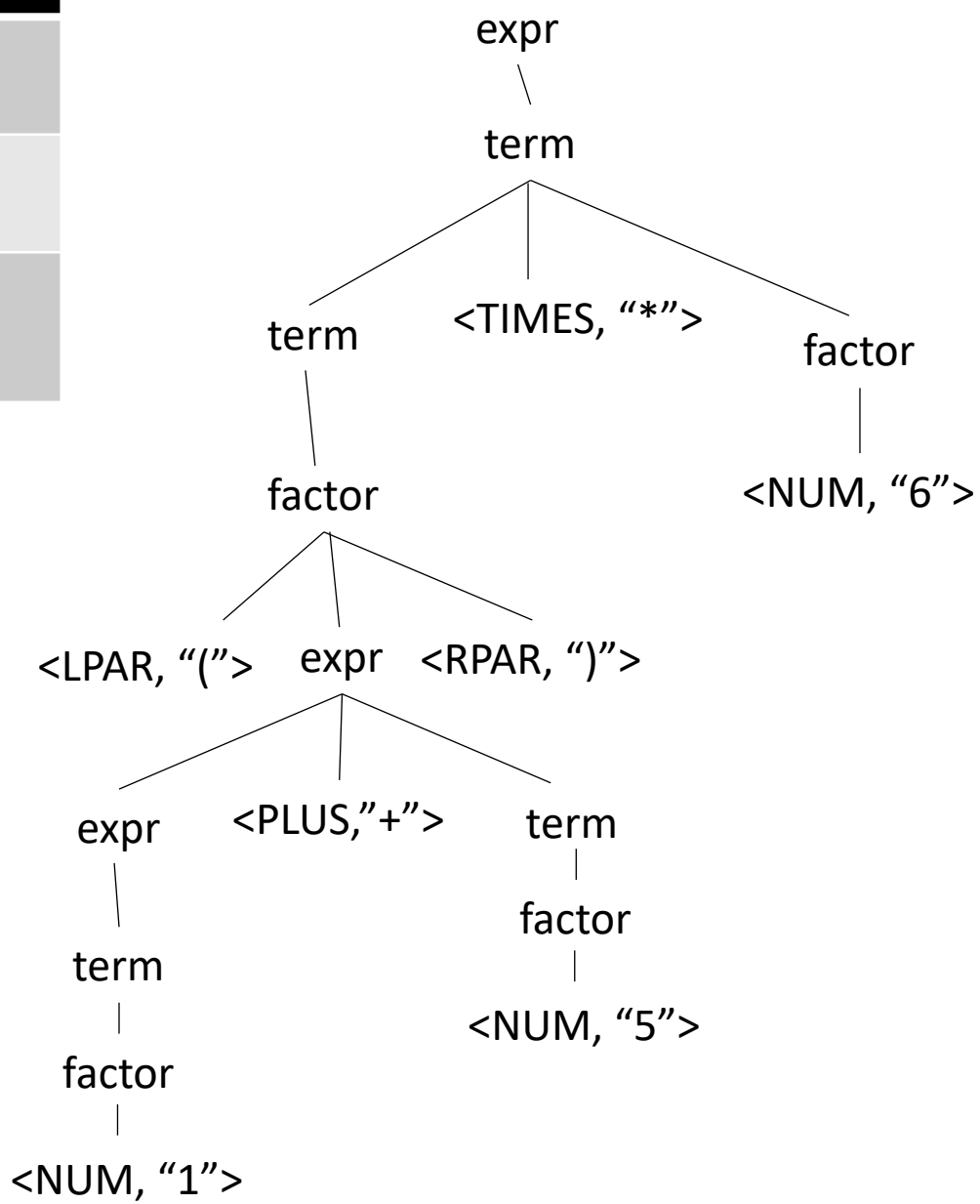
Lets build the AST

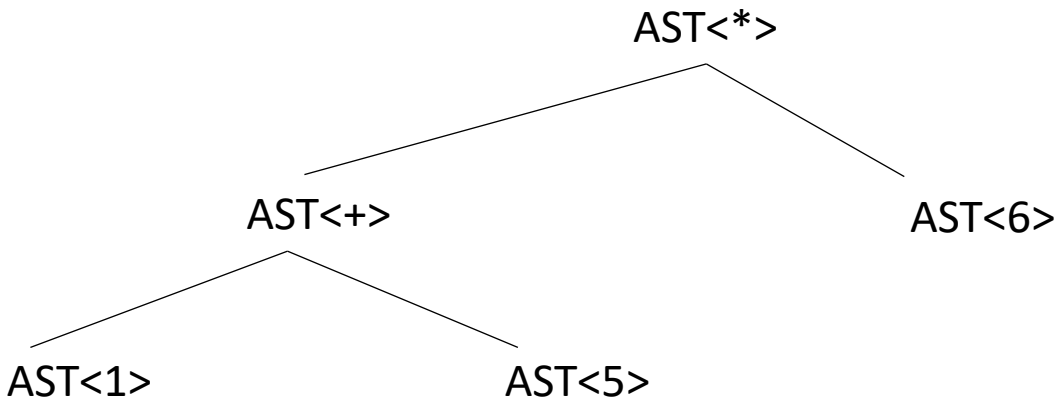
AST<?>



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)}

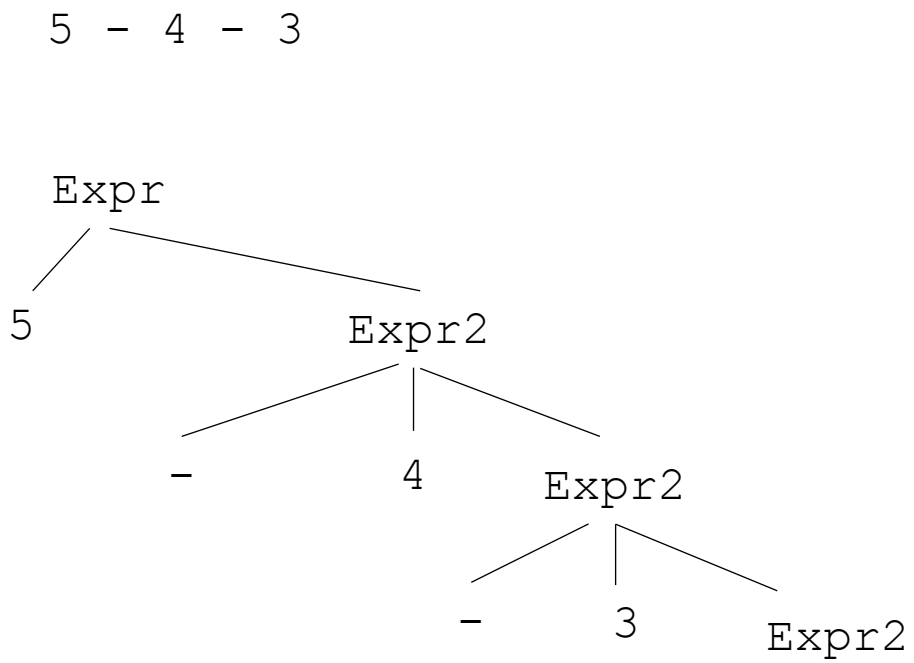
input: (1+5)*6





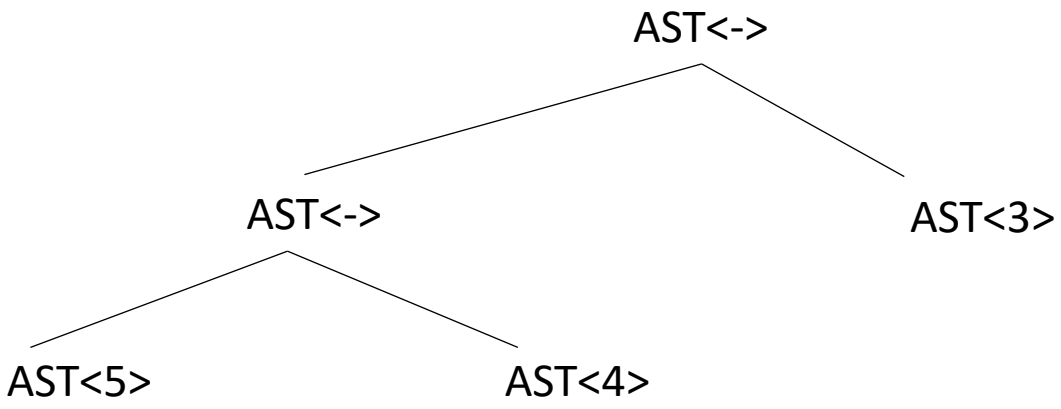
Creating an AST from top down grammar

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

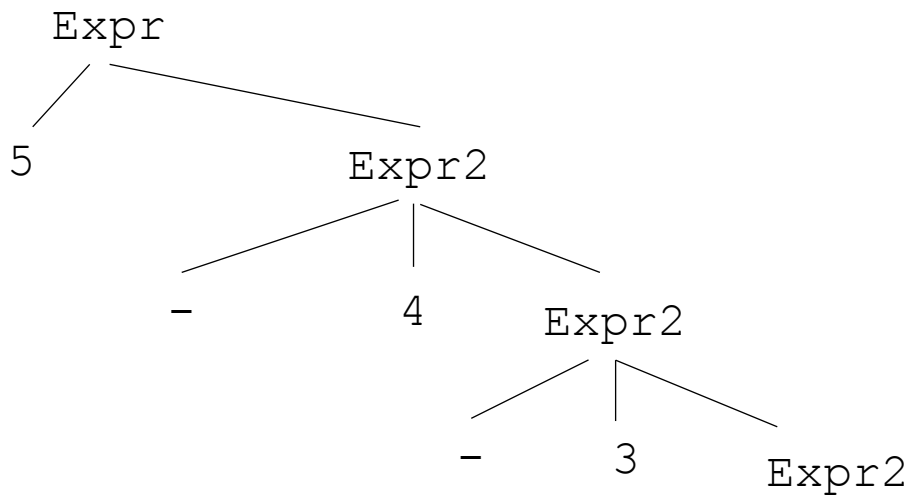


Creating an AST from top down grammar

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



5 - 4 - 3

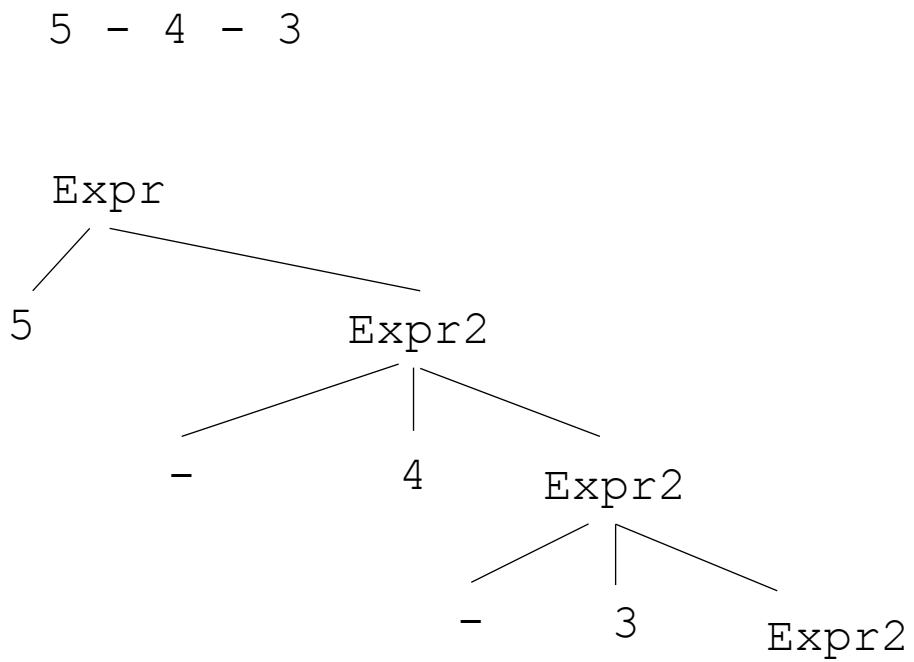


How do we get to the desired parse tree?

Creating an AST from top down grammar

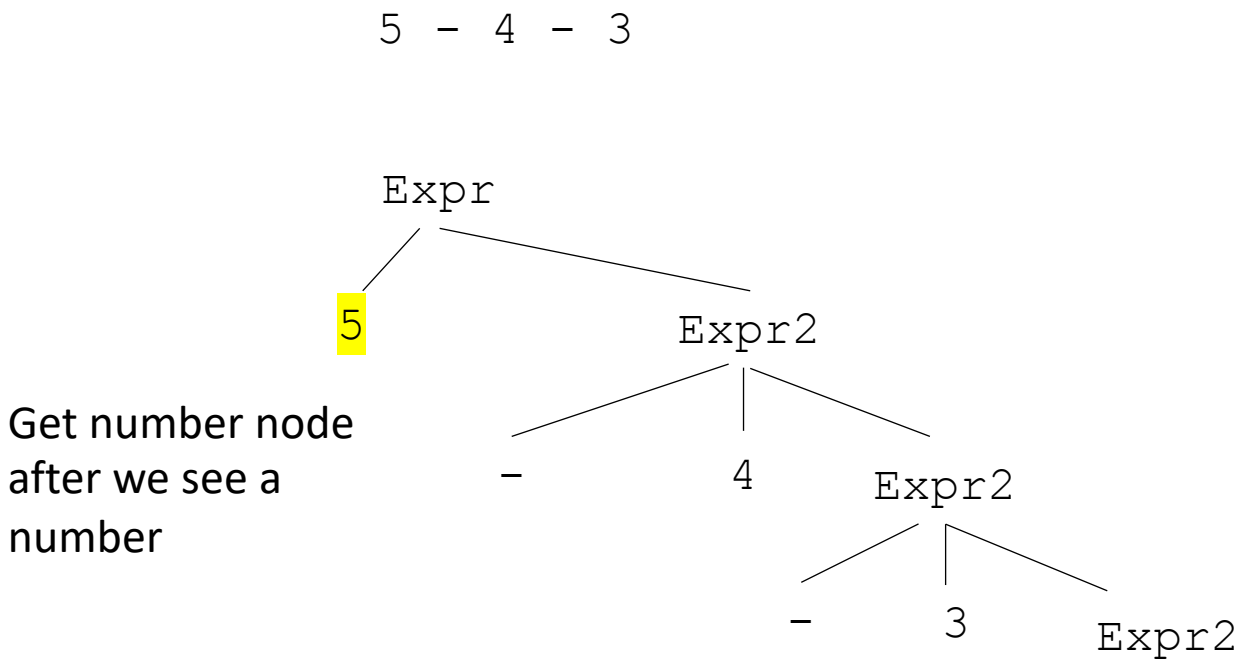
```
Expr  ::= NUM Expr2
Expr2 ::= MINUS NUM Expr2
        |   “”
```

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
       |   “”
```



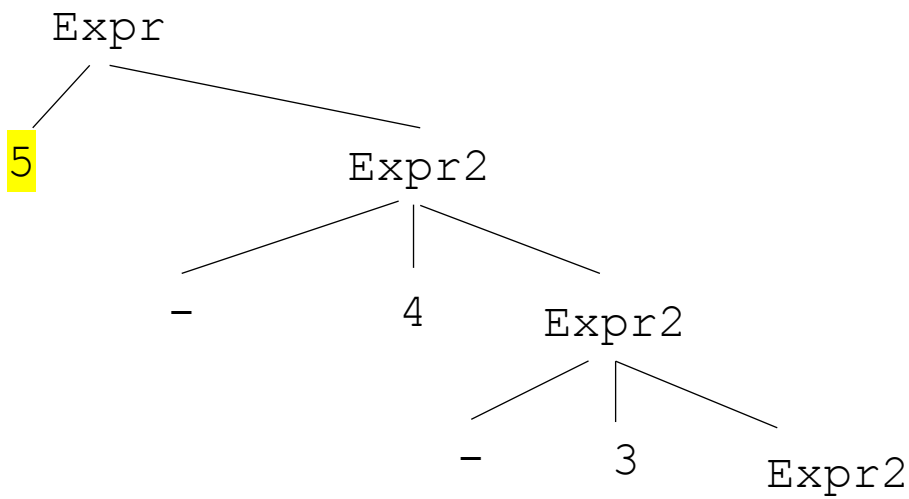
Get number node after we see a number

AST<5>

Creating an AST from top down grammar

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

5 - 4 - 3

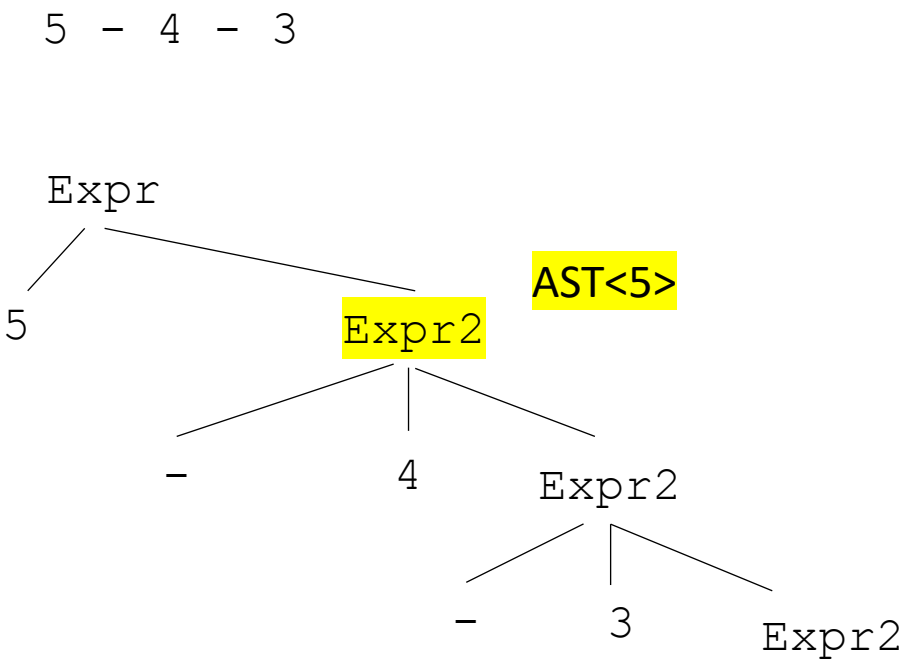


Pass the node down

AST<5>

Creating an AST from top down grammar

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

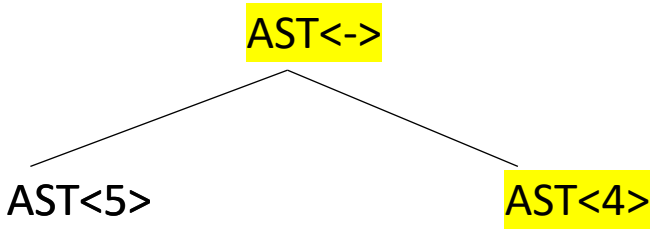


Pass the node down

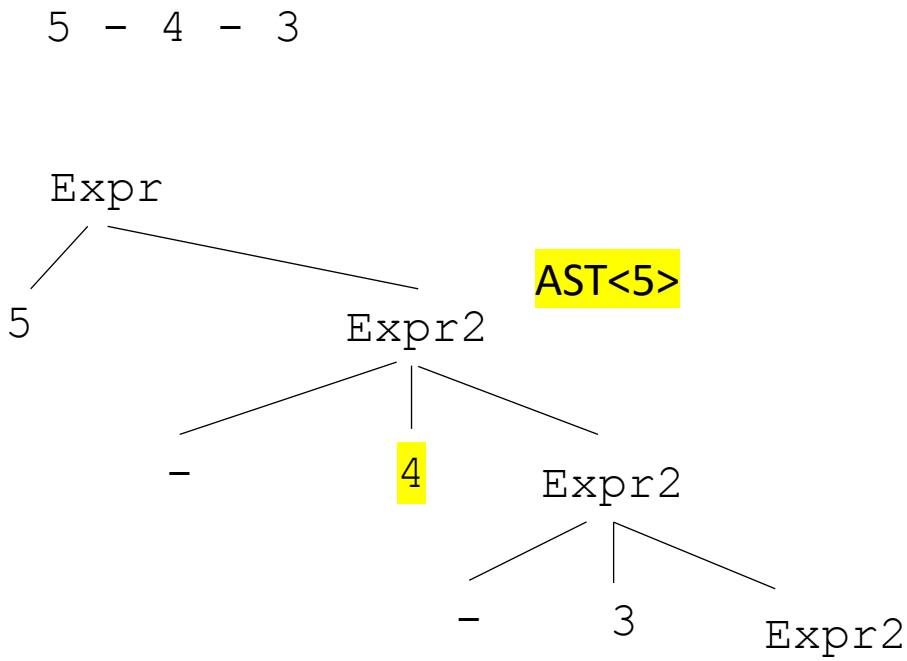
AST<5>

Creating an AST from top down grammar

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

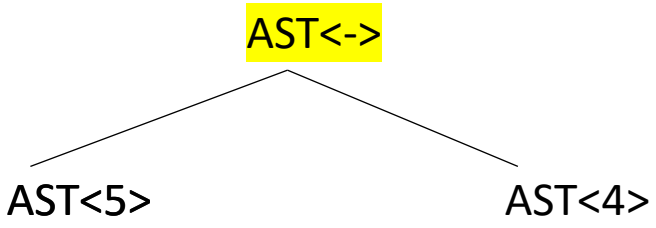


In Expr2, after 4 is parsed, create a number node and a minus node

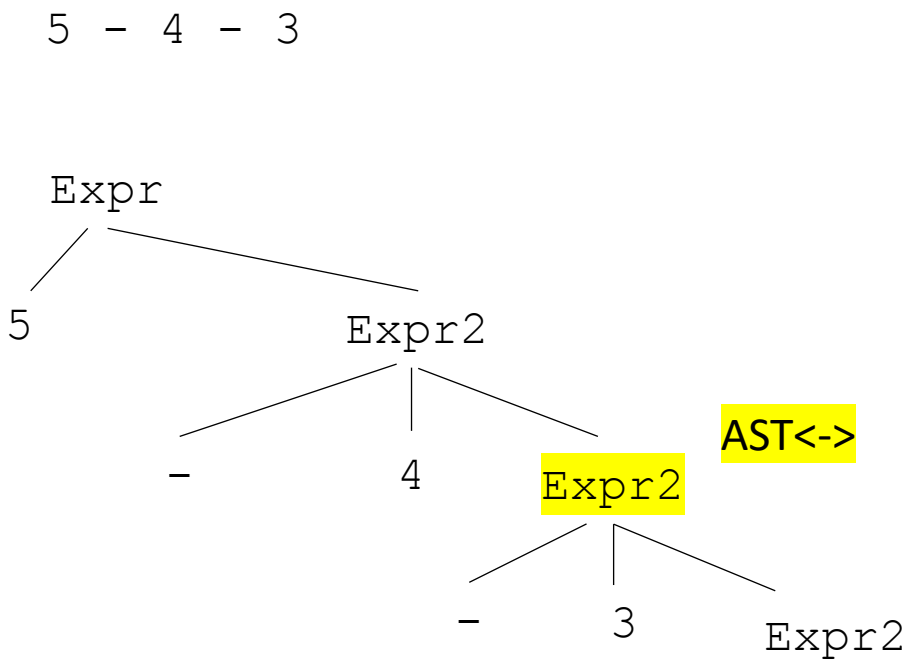


Creating an AST from top down grammar

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

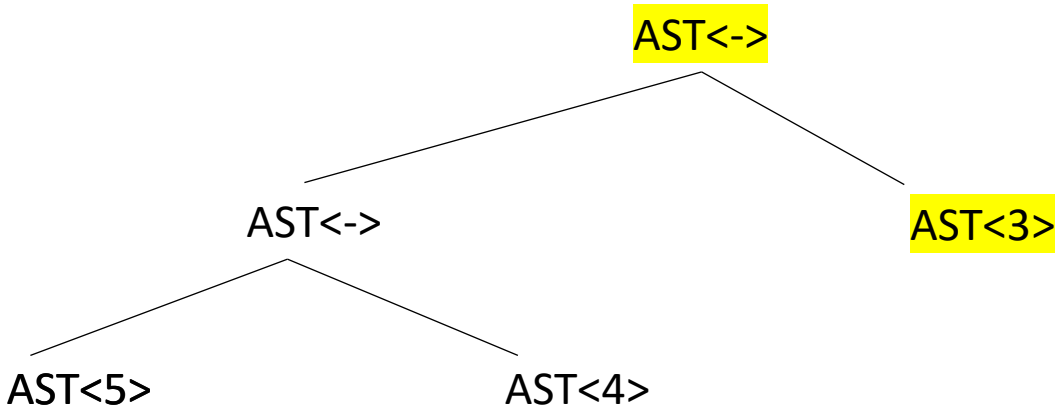


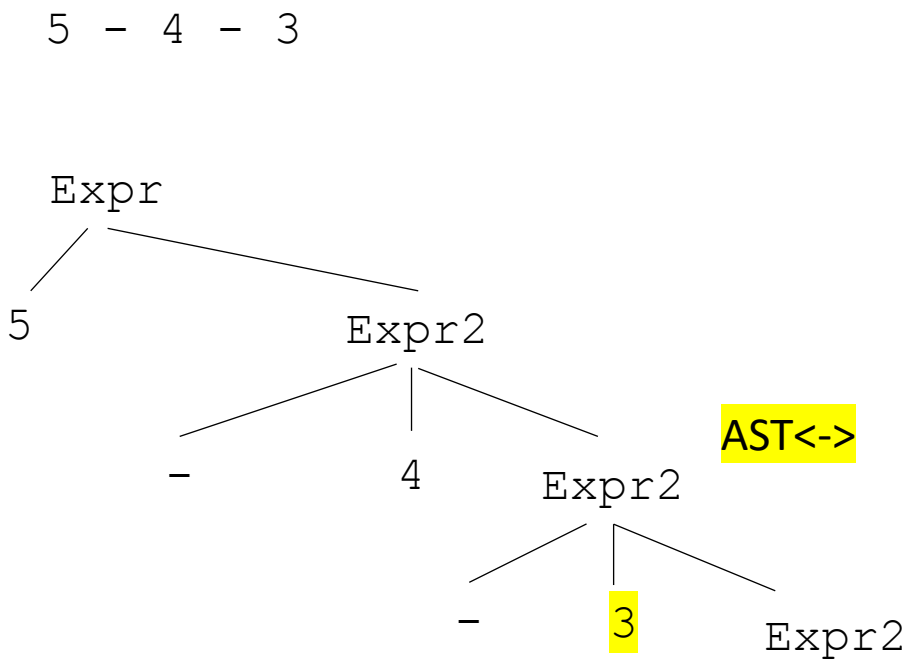
pass the new node down



Creating an AST from top down grammar

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

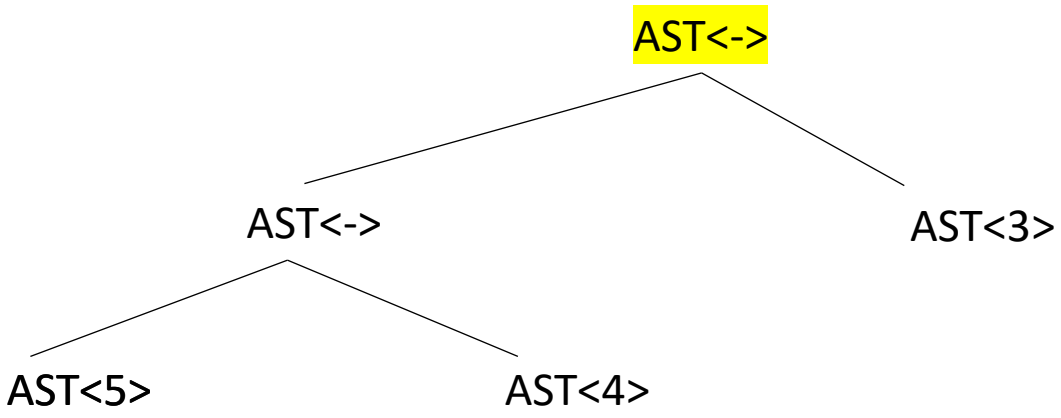


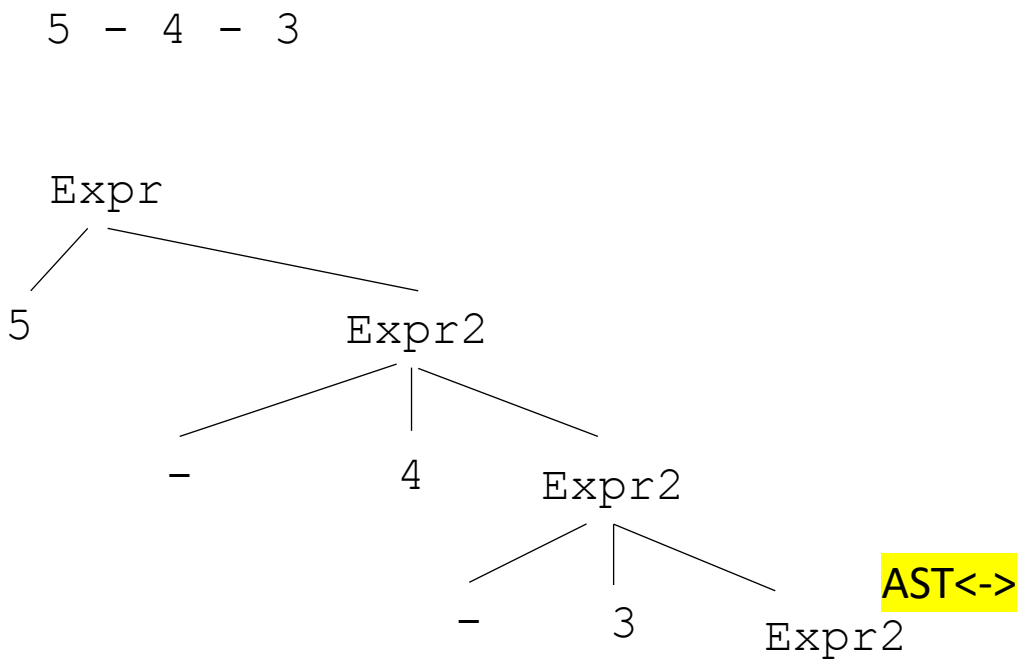


In Expr2, after 3 is parsed, create a number node and a minus node

Creating an AST from top down grammar

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

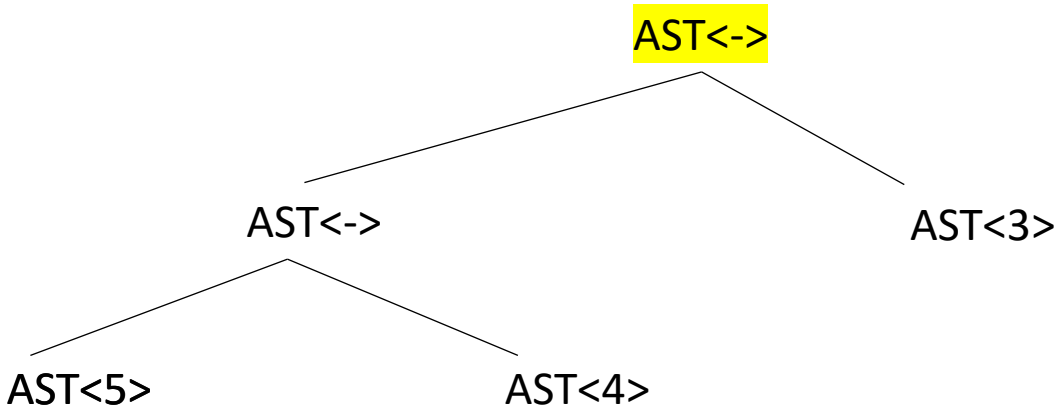


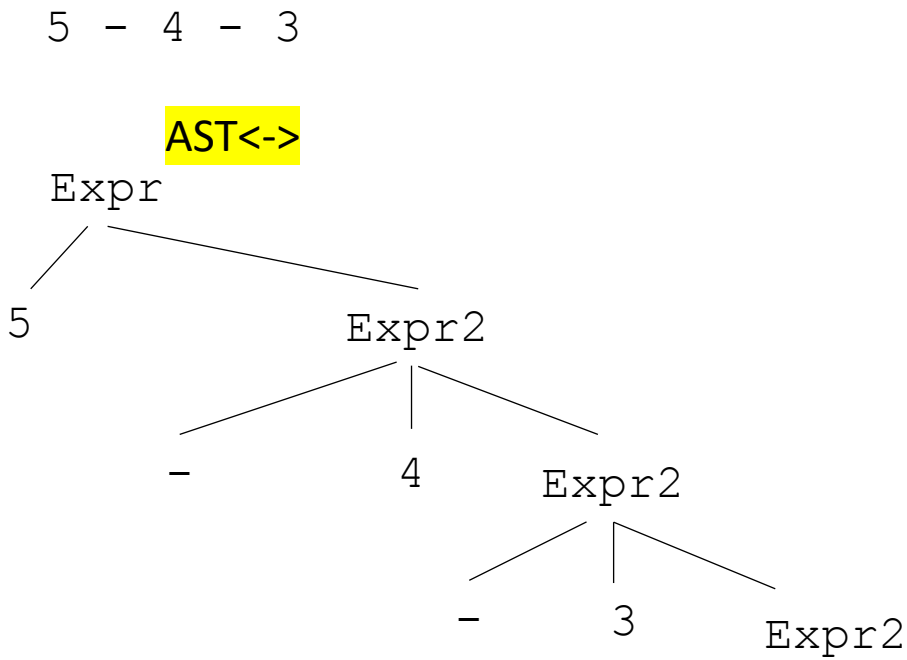


pass down the new node

Creating an AST from top down grammar

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





return the node when there is nothing left to parse

Creating an AST from top down grammar

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

```
def parse_expr(self):
    #get the value from the lexeme
    value = self.to_match.value
    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
       |   “”
```

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

how to adapt?

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

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

Creating an AST from top down grammar

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

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

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

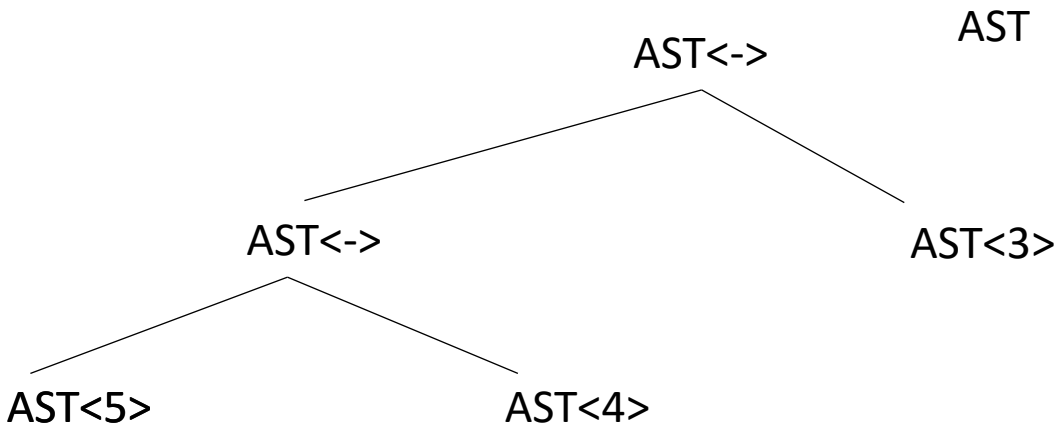
how to adapt?

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

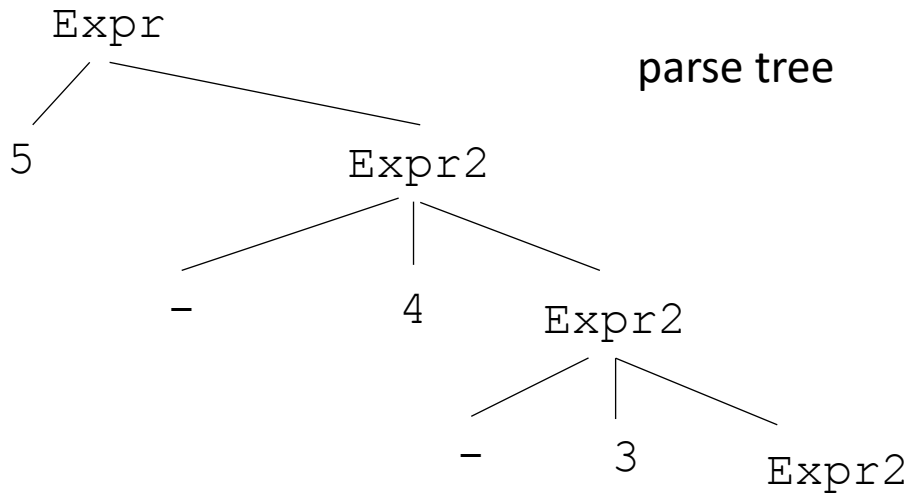
The parse_term will figure out how to get you an AST node for that term.

Evaluate an AST by doing a post order traversal

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



5 - 4 - 3



parse tree

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

Example

- Python AST

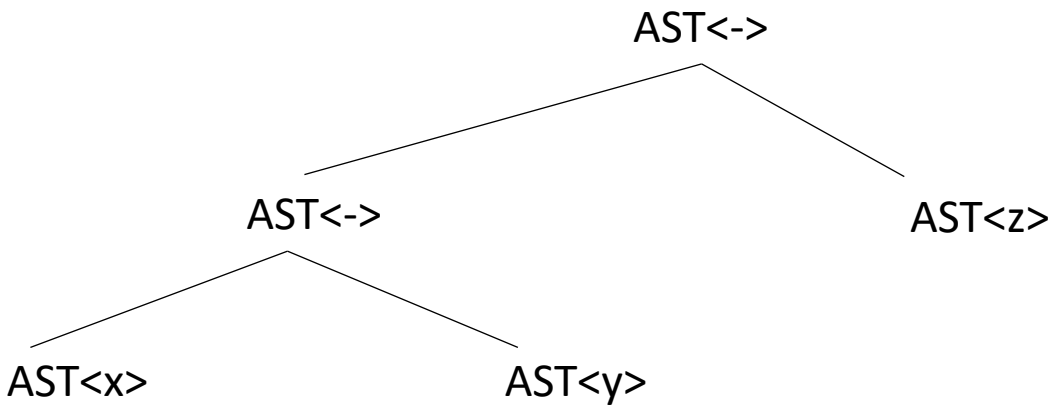
```
import ast

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
      |   “”
```

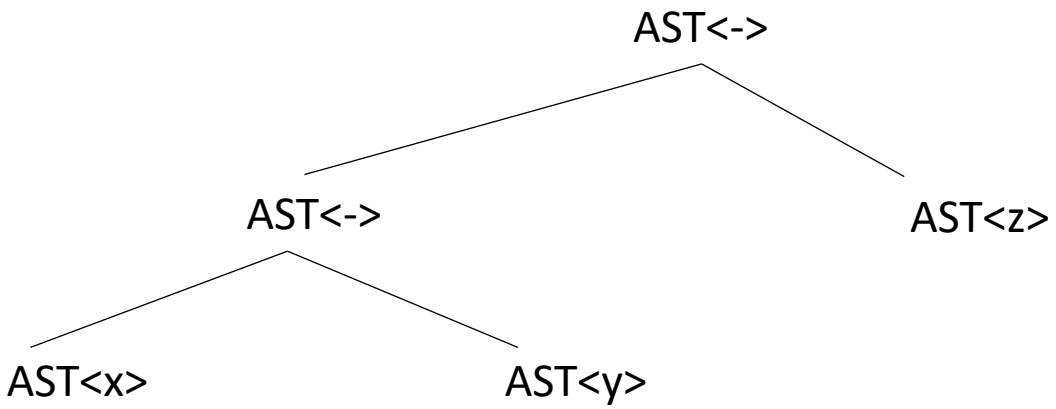


What if you cannot evaluate it?
What else might you do?

```
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
```

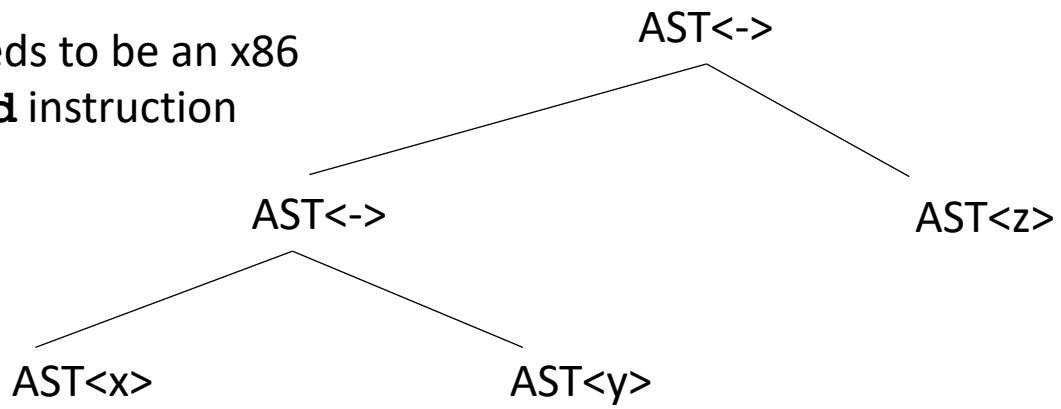
How does this change things?

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?

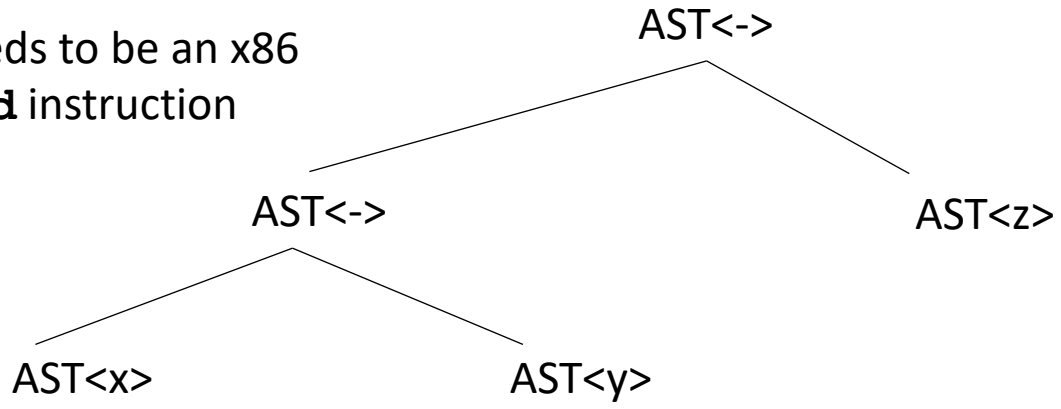
Evaluate an AST by doing a post order traversal

```
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?

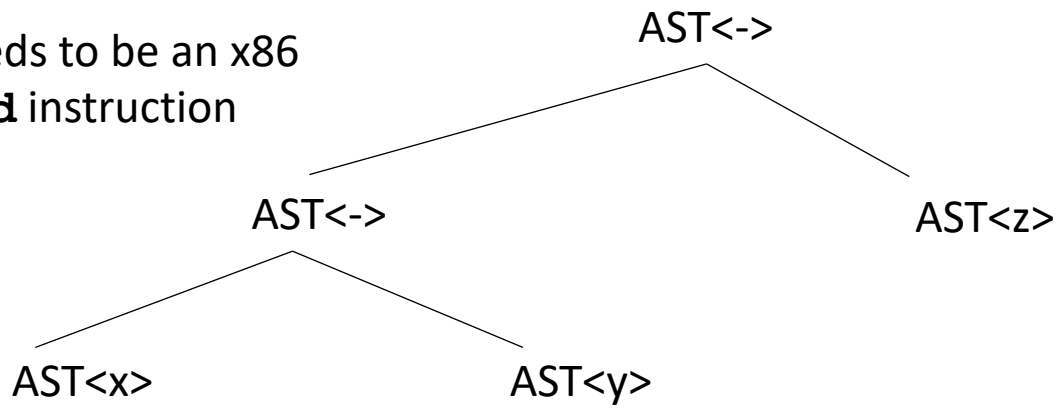
Evaluate an AST by doing a post order traversal

```
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

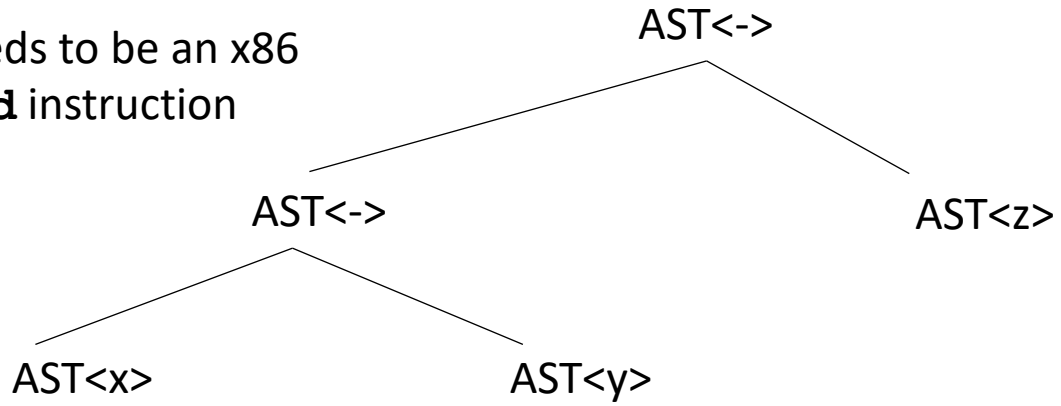
Evaluate an AST by doing a post order traversal

```
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 subtract them!

Is this all?

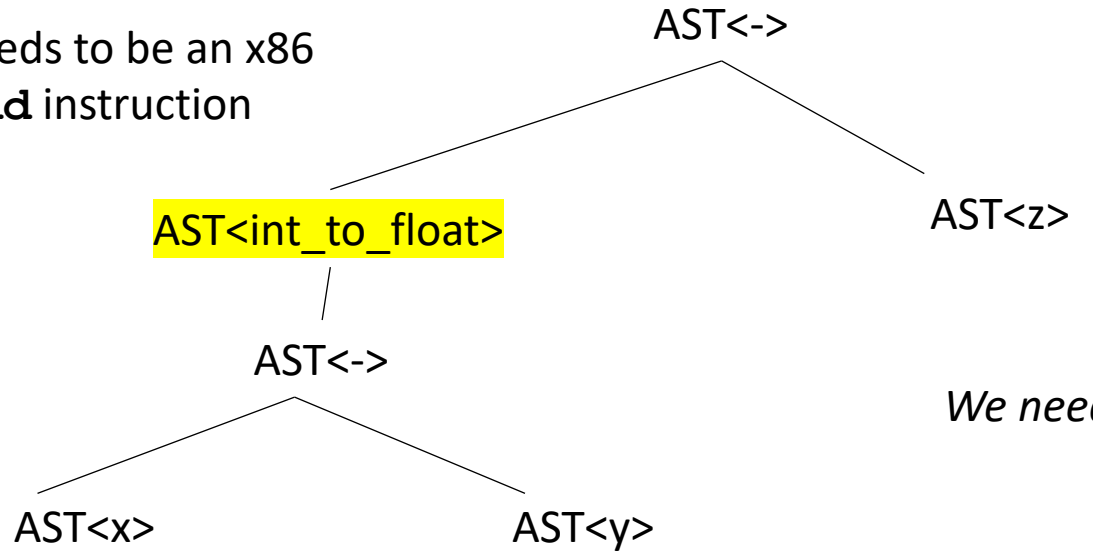
Evaluate an AST by doing a post order traversal

```
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

See everyone on Monday

- Study for the test!