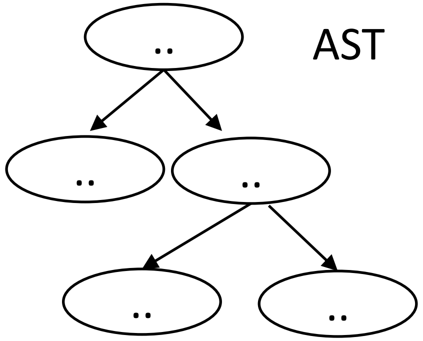
CSE110A: Compilers

April 27, 2022

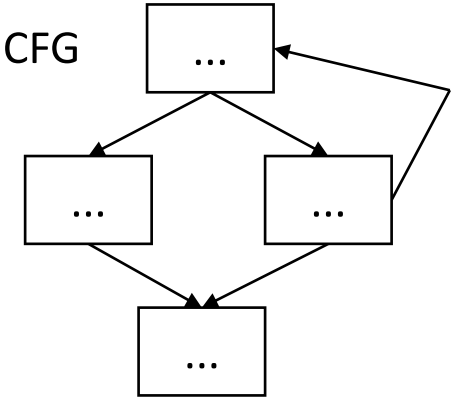
AST



Topics:

- *Module 3: Intermediate representations*
  - *Intro to intermediate representations*
  - *ASTs*
    - *parse trees into ASTs*

CFG



3 address code

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

Announcements

- HW 2
  - Due on Monday by midnight
  - Still have lots of chances for help
  - If you haven’t started yet, I highly suggest that you start!

- Midterm will be given on May 2
  - Take home midterm.
  - Assigned on Monday morning and due on Friday by midnight
  - No late midterms are accepted

Announcements

- HW 2
  - Due on Monday by midnight
  - Still have lots of chances for help
  - If you haven't started yet, I highly suggest that you start!

- Midterm will be given on May 2
  - Take home midterm.
  - Assigned on Monday morning and due on Friday by midnight
  - No late midterms are accepted

- HW 1 grades
  - Hoping to get them by Monday

Announcements

- Neal wrote a recursive descent primer on Piazza, check it out!

Homework 2 clarifications

- Tip for starting on statement rules

- A statement can be one of the following:
  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.

Simply translate the English:

- A statement can be one of the following:
  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword "if" followed by an expression enclosed in ()s. Next is a statement, followed by the "else" keyword. Following "else" is another statement.

Simply translate the English:

```
Statement ::= variable_declaration
           |   assignment_statement
           |   if_else_statement
```

- A statement can be one of the following:
  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.

Simply translate the English:

```
Statement ::= variable_declaration
           |   assignment_statement
           |   if_else_statement
```

```
variable_declaration ::= TYPE ID SEMI
```

- A statement can be one of the following:
  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.

Simply translate the English:

```
Statement ::= variable_declaration
          |   assignment_statement
          |   if_else_statement
```

```
variable_declaration ::= TYPE ID SEMI
```

- A statement can be one of the following:
  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.

Simply translate the English:

```
Statement ::= variable_declaration
           |   assignment_statement
           |   if_else_statement
```

```
variable_declaration ::= type ID SEMI
```

```
type ::= FLOAT
      |   INT
```

Homework 2 clarifications

- Statement precedence

- Do we need to encode statement precedence? Or associativity?

Homework 2 clarifications

```
Statement_list ::= Statement_list Statement
                |   Statement
```

Which one do we want?

```
Statement_list ::= Statement Statement_list
                |   Statement
```

Homework 2 clarifications

```
Statement_list ::= Statement_list Statement
               |   Statement
```

We don't want left recursion for top-down parsing

```
Statement_list ::= Statement Statement_list
               |   Statement
```

We might want left recursion for left associativity

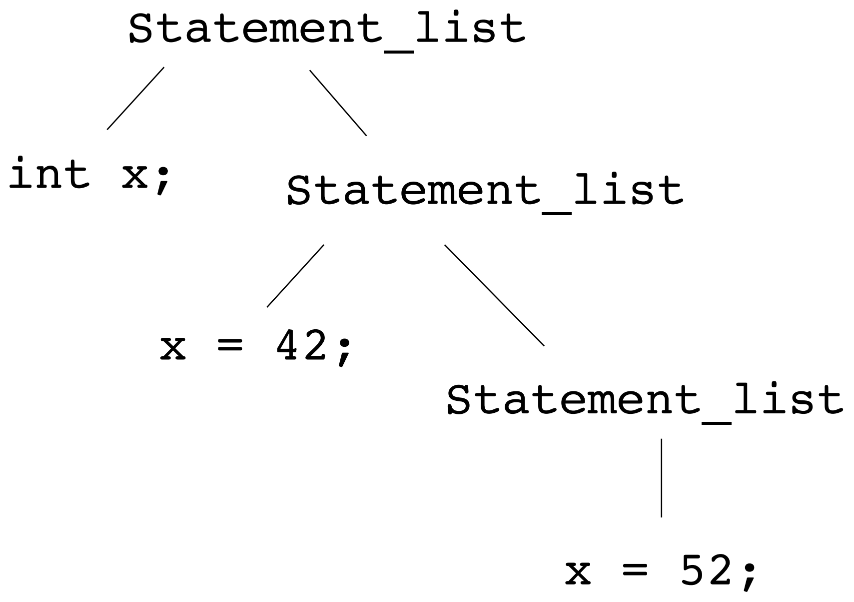
```
int x; x = 42; x = 52;
```

think about this program. We want to evaluate it left to right.

Homework 2 clarifications

```
Statement_list ::= Statement Statement_list
                |   Statement
```

```
int x; x = 42; x = 52;
```



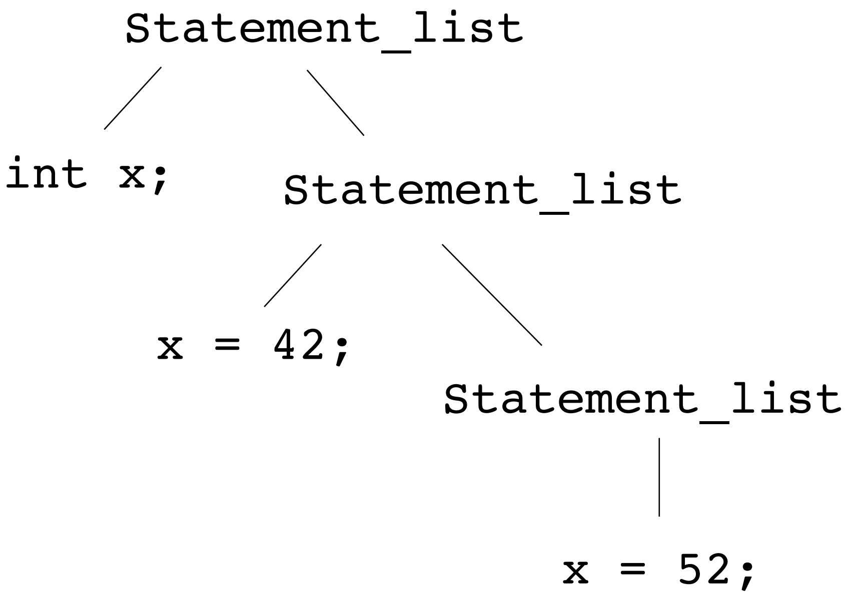
Homework 2 clarifications

```
Statement_list ::= Statement Statement_list
                |   Statement
```

```
int x; x = 42; x = 52;
```

there is no evaluation associated with a statement list. The evaluation should occur at the statement

Thus we can use the right recursive form with no issue. We also don't have to worry about statement precedence



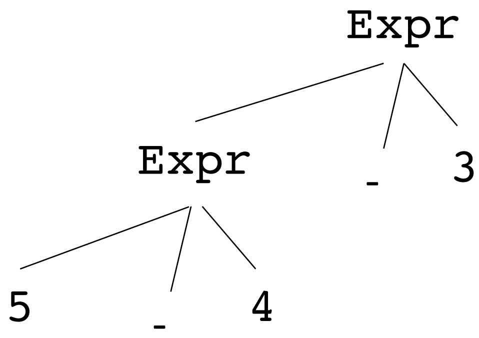
Homework 2 clarifications

- Left associativity and left recursion expressions

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
      |   NUM
```

5 - 4 - 3

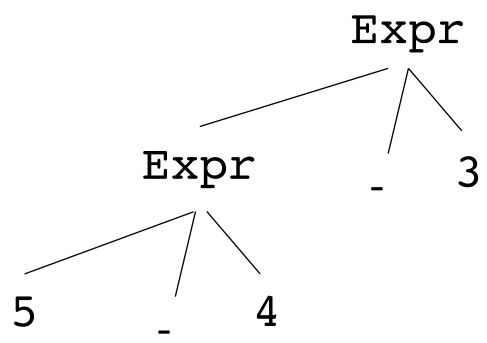


Left recursive grammar makes this parse tree. It encodes associativity

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
       |    NUM
```

5 – 4 – 3



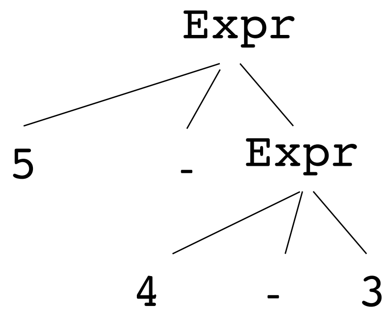
Left recursive grammar makes this parse tree. It encodes associativity.

But left recursion won't work for top-down parsers!

What if we do it right recursive

```
Expr ::= NUM MINUS Expr
       |    NUM
```

5 – 4 – 3

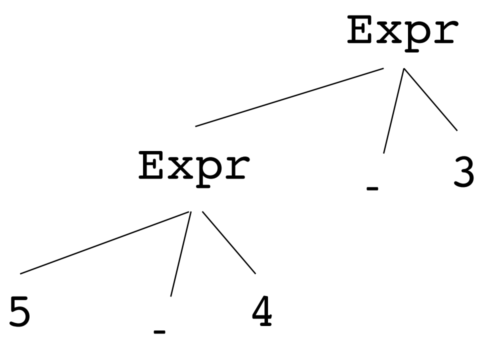


We can use this grammar in a top-down parser, but it doesn't encode associativity

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
      |   NUM
```

5 – 4 – 3



Left recursive grammar makes this parse tree. It encodes associativity.

But left recursion won't work for top-down parsers!

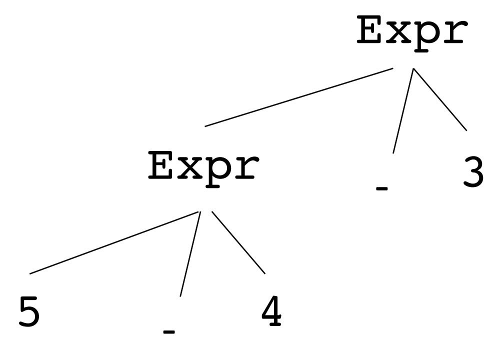
What if we follow the recipe

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

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
       |    NUM
```

5 – 4 – 3

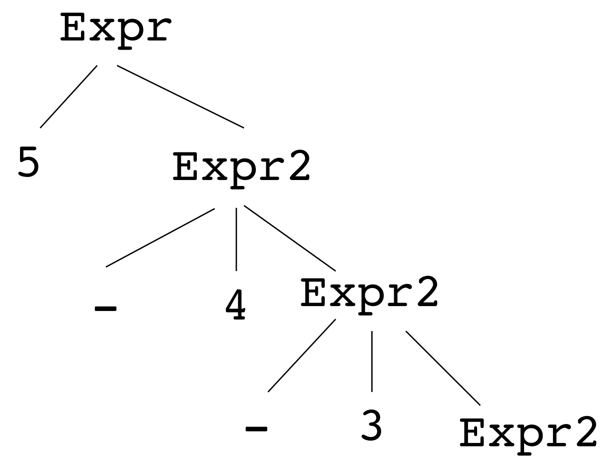


Left recursive grammar makes this parse tree. It encodes associativity.

But left recursion won't work for top-down parsers!

What if we follow the recipe

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

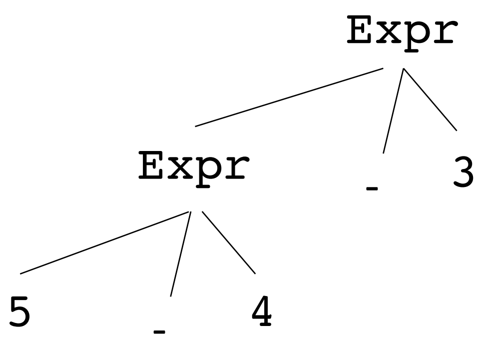


How about this one?

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM
      |    NUM
```

5 – 4 – 3

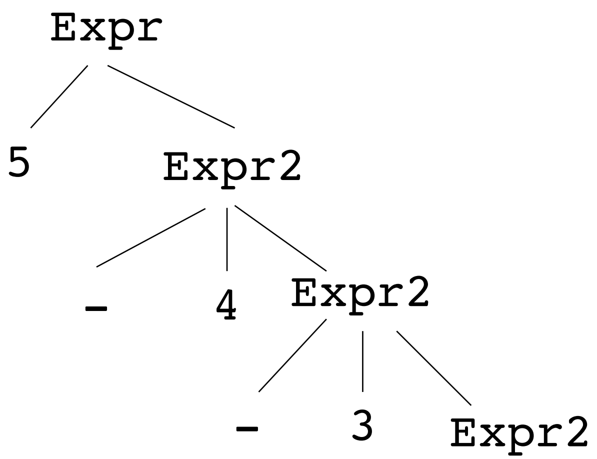


Left recursive grammar makes this parse tree. It encodes associativity.

But left recursion won't work for top-down parsers!

What if we follow the recipe

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



How about this one?

It isn't really clear...

We will talk about it more today but for your homework, encode associativity in your original grammar (1.1) and use the recipe for eliminating left recursion for the rest.

Homework 2 clarifications

- Scopes for symbol table

- In which cases do you need to start a new scope?

  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword "if" followed by an expression enclosed in ()s. Next is a statement, followed by the "else" keyword. Following "else" is another statement.
  - A for statement, which is the keyword "for" followed by an assignment statement, an expression, and an assignment statement all enclosed in ()s. The final assignment statement does not require a semicolon. After the ()s is a statement.
  - A block statement, which is a sequence of statements enclosed in braces {}s.

Homework 2 clarifications

- Scopes for symbol table

- In which cases do you need to start a new scope?

  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.
  - A for statement, which is the keyword ”for” followed by an assignment statement, an expression, and an assignment statement all enclosed in ()s. The final assignment statement does not require a semicolon. After the ()s is a statement.
  - A block statement, which is a sequence of statements enclosed in braces {}s.

Homework 2 clarifications

- Scopes for symbol table

- In which cases do you need to start a new scope?

  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword "if" followed by an expression enclosed in ()s. Next is a statement, followed by the "else" keyword. Following "else" is another statement.
  - A for statement, which is the keyword "for" followed by an assignment statement, an expression, and an assignment statement all enclosed in ()s. The final assignment statement does not require a semicolon. After the ()s is a statement.
  - A block statement, which is a sequence of statements enclosed in braces {}s.

Homework 2 clarifications

- Scopes for symbol table

- In which cases do you need to start a new scope?

  – A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  – An assignment statement, which is ID followed by = followed by an expression.
  – An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.
  – A for statement, which is the keyword ”for” followed by an assignment statement, an expression, and an assignment statement all enclosed in ()s. The final assignment statement does not require a semicolon. After the ()s is a statement.
  – A block statement, which is a sequence of statements enclosed in braces {}s.

Homework 2 clarifications

- Scopes for symbol table
- In which cases do you need to start a new scope?

```
int foo() {
  if (1)
    int x;
  else
    int y;

  return 0;
}
```

Is this allowed in C-simple?

Is it allowed in C?

Homework 2 clarifications

- Scopes for symbol table

- In which cases do you need to start a new scope?

I have failed ☹ C-simple is not a strict subset of C

We won't test this case.

```
int foo() {
  if (1)
    int x;
  else
    int y;

  return 0;
}
```

Is this allowed in C-simple? Yes!

Is it allowed in C? No!

Homework 2 clarifications

- Scopes for symbol table

- In which cases do you need to start a new scope?

  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword ”if” followed by an expression enclosed in ()s. Next is a statement, followed by the ”else” keyword. Following ”else” is another statement.
  - A for statement, which is the keyword ”for” followed by an assignment statement, an expression, and an assignment statement all enclosed in ()s. The final assignment statement does not require a semicolon. After the ()s is a statement.
  - A block statement, which is a sequence of statements enclosed in braces {}s.

Homework 2 clarifications

- Scopes for symbol table
- In which cases do you need to start a new scope?

```
int foo() {
  int i;
  for (i = 0; i < 100; i = i + 1)
    int y;

  return 0;
}
```

Homework 2 clarifications

- Scopes for symbol table

- In which cases do you need to start a new scope?

We won't test this case either.

```
int foo() {
  int i;
  for (i = 0; i < 100; i = i + 1)
    int y;

  return 0;
}
```

Is this allowed in C-simple? Yes!

Is it allowed in C? No!

Homework 2 clarifications

- Scopes for symbol table

- In which cases do you need to start a new scope?

How about this one?

```
int foo() {

  for (int i = 0; i < 100; i = i + 1)
    i = i + 1;

  return 0;
}
```

Homework 2 clarifications

- Scopes for symbol table
- In which cases do you need to start a new scope?

starts a new scope in C. But you don't have to worry about it in C-simple

```
int foo() {

  for (int i = 0; i < 100; i = i + 1)
    i = i + 1;

  return 0;
}
```

Is this allowed in C-simple? No!

Is it allowed in C? Yes!

Homework 2 clarifications

- Scopes for symbol table
- In which cases do you need to start a new scope?

  - A variable declaration, which is a type name followed by an ID, followed by a semi colon. Types for C-simple are ints or floats.
  - An assignment statement, which is ID followed by = followed by an expression.
  - An if-else statement, which is the keyword "if" followed by an expression enclosed in ()s. Next is a statement, followed by the "else" keyword. Following "else" is another statement.
  - A for statement, which is the keyword "for" followed by an assignment statement, an expression, and an assignment statement all enclosed in ()s. The final assignment statement does not require a semicolon. After the ()s is a statement.
  - A block statement, which is a sequence of statements enclosed in braces {}s.

Homework 2 clarifications

- Scopes for symbol table
- In which cases do you need to start a new scope?

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

Homework 2 clarifications

- Scopes for symbol table
- In which cases do you need to start a new scope?

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

block statement needs a new scope

Quiz

Quiz

Error messages about undeclared variables are printed by

- ○ Scanner
- ○ Parser
- ○ Symbol Table
- ○ Code Generator

Quiz

Error messages about undeclared variables are printed by

- ◯ Scanner
- ◯ Parser
- ◯ Symbol Table
- ◯ Code Generator

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

Quiz

Thinking about scoping rules for Python and C (constrained to a single function): Please write a few sentences about the differences in how each language should utilize a symbol table, e.g. to catch variables that are used before they are defined.

Quiz

Thinking about scoping rules for Python and C (constrained to a single function): Please write a few sentences about the differences in how each language should utilize a symbol table, e.g. to catch variables that are used before they are defined.

```python
if (1):
    x = 5
print(x)
```

is this allowed?

```c
int main() {
  if (1) {
    int x = 5;
  }
  printf("%d\n",x);
}
```

is this allowed?

Quiz

Thinking about scoping rules for Python and C (constrained to a single function): Please write a few sentences about the differences in how each language should utilize a symbol table, e.g. to catch variables that are used before they are defined.

```
if (1):
    x = 5
print(x)
```

is this allowed? yes

```
int main() {
  if (1) {
    int x = 5;
  }
  printf("%d\n",x);
}
```

is this allowed? no

Quiz

We can always evaluate arithmetic computations during parsing using parser actions.

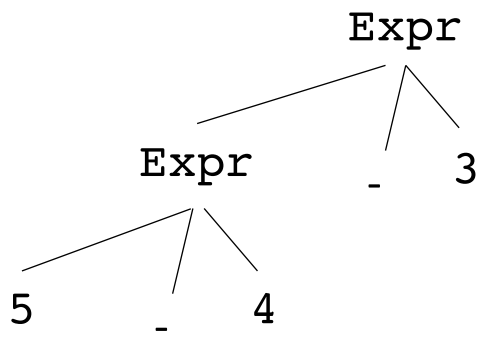
- ○ True
- ○ False

Quiz

We can always evaluate arithmetic computations during parsing using parser actions.

- ◯ True
- ◯ False

5 - 4 - 3



Simple grammar for minus expressions

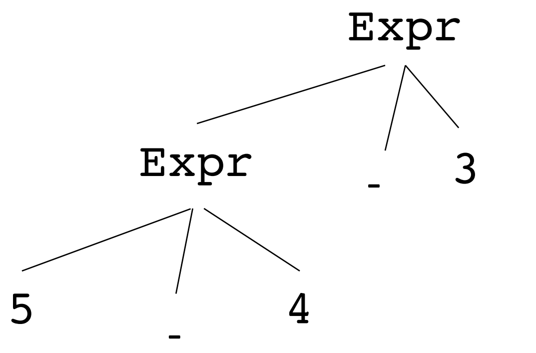
```
Expr ::= Expr MINUS NUM
      |   NUM
```

Quiz

We can always evaluate arithmetic computations during parsing using parser actions.

- ○ True
- ○ False

5 - 4 - 3



Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM {return $1 - $3}
      |   NUM            {return $1}
```

Quiz

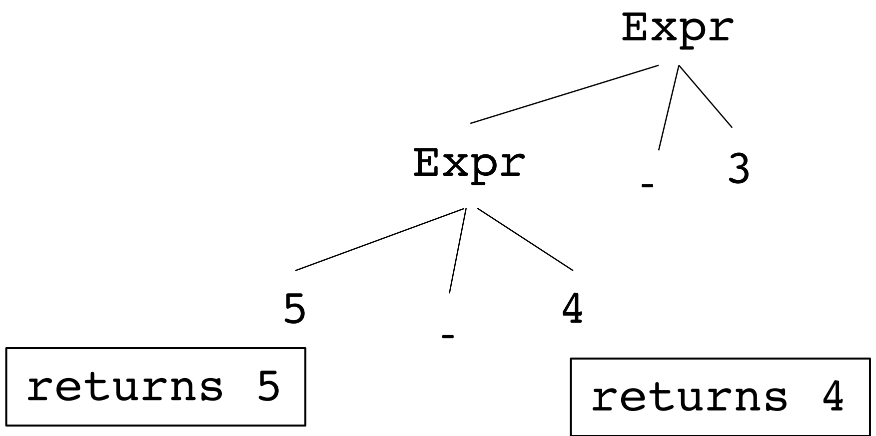
We can always evaluate arithmetic computations during parsing using parser actions.

- ○ True
- ○ False

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM {return $1 - $3}
       |   NUM             {return $1}
```

5 - 4 - 3

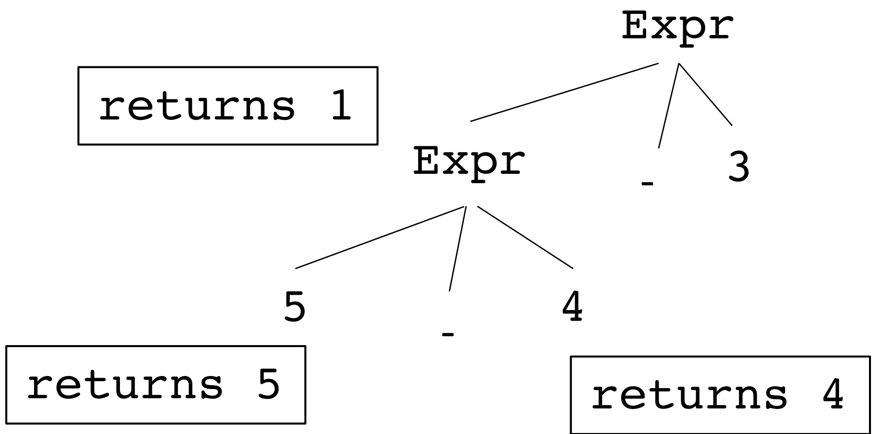


Quiz

We can always evaluate arithmetic computations during parsing using parser actions.

- ◯ True
- ◯ False

5 - 4 - 3



Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM {return $1 - $3}
      |   NUM            {return $1}
```

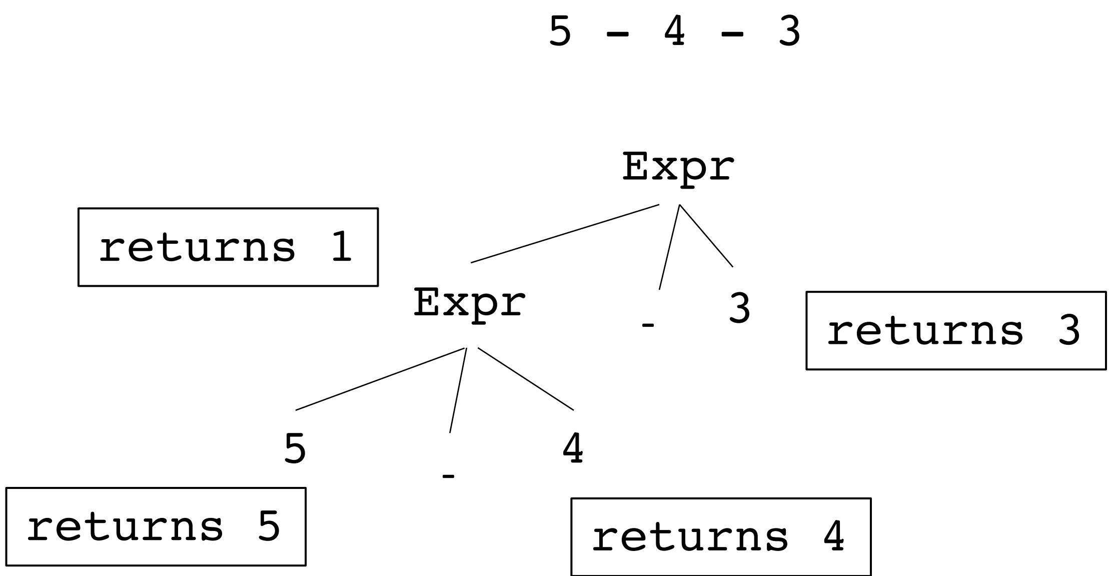
Quiz

We can always evaluate arithmetic computations during parsing using parser actions.

- ◯ True
- ◯ False

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM {return $1 - $3}
      |   NUM               {return $1}
```



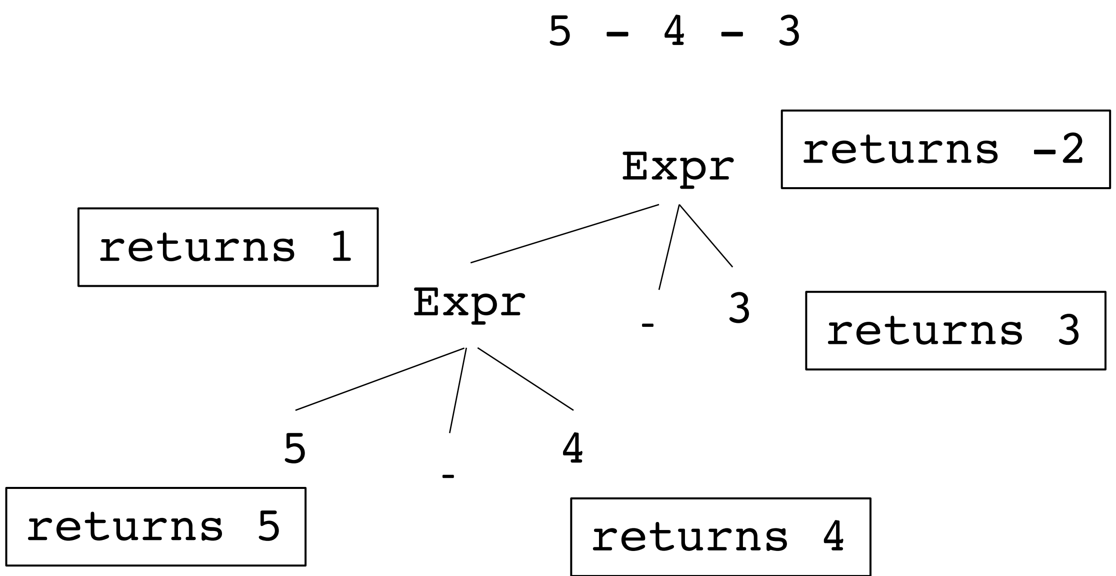
Quiz

We can always evaluate arithmetic computations during parsing using parser actions.

- ◯ True
- ◯ False

Simple grammar for minus expressions

```
Expr ::= Expr MINUS NUM {return $1 - $3}
      |    NUM              {return $1}
```



Quiz

We can always evaluate arithmetic computations during parsing using parser actions.

- ○ True
- ○ False

So why can't we always evaluate arithmetic expressions during parsing?

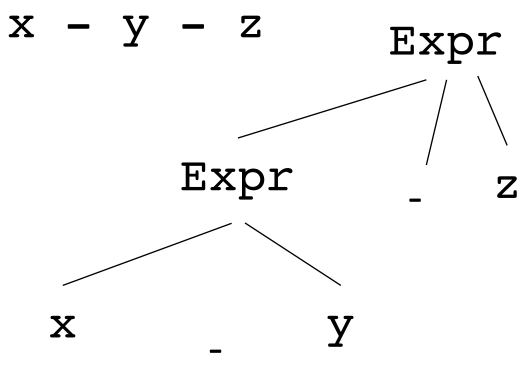
Quiz

We can always evaluate arithmetic computations during parsing using parser actions.

- ◯ True
- ◯ False

So why can't we always evaluate arithmetic expressions during parsing?

```
Expr ::= Expr MINUS UNIT {return $1 - $3}
      |  UNIT            {return $1}
UNIT ::= NUM             {return $1}
      |  ID              {return $1}
```



Quiz

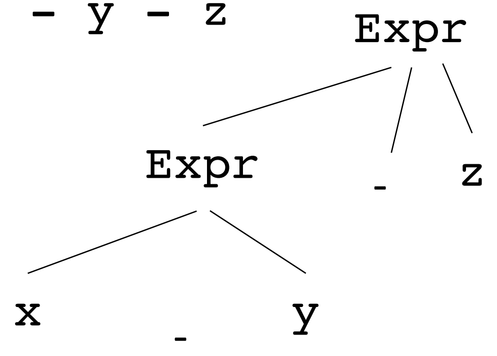
We can always evaluate arithmetic computations during parsing using parser actions.

- ◯ True
- ◯ False

We cannot evaluate the program unless we know the value of x,y,z. What are some examples when we wouldn't know the values?

```
Expr ::= Expr MINUS UNIT {return $1 - $3}
       |   UNIT          {return $1}
UNIT ::= NUM             {return $1}
       |   ID            {return $1}
```

x - y - z



Quiz

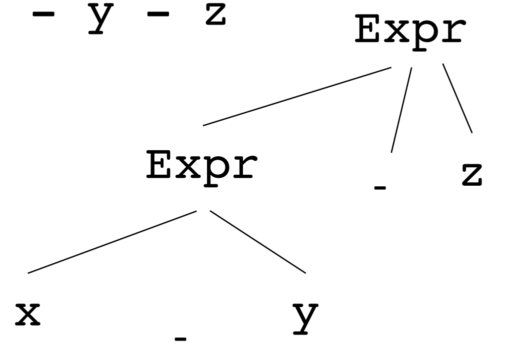
We can always evaluate arithmetic computations during parsing using parser actions.

- ◯ True
- ◯ False

But we might be able to do some optimizations...

```
Expr ::= Expr MINUS UNIT {return $1 - $3}
      |   UNIT             {return $1}
UNIT ::= NUM               {return $1}
      |   ID               {return $1}
```

x - y - z



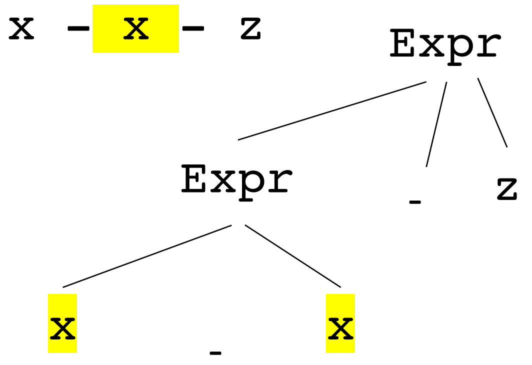
Quiz

We can always evaluate arithmetic computations during parsing using parser actions.

- ◯ True
- ◯ False

But we might be able to do some optimizations...

```
Expr ::= Expr MINUS UNIT  {return $1 - $3}
      |  UNIT             {return $1}
UNIT ::= NUM              {return $1}
      |  ID               {return $1}
```



Quiz

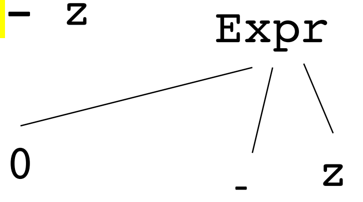
We can always evaluate arithmetic computations during parsing using parser actions.

- ◯ True
- ◯ False

But we might be able to do some optimizations…

```
Expr ::= Expr MINUS UNIT {if $1 == $3 then 0 else ...}
      |   UNIT            {return $1}
UNIT ::= NUM              {return $1}
      |   ID              {return $1}
```

x - x - z



Quiz

It is the last lecture of Module 2; please let me know any feedback you might have about the module: e.g. what you enjoyed or what you think could be improved.

Thanks for your feedback! Apologies again about the disorganization caused by the technical failure!

Review

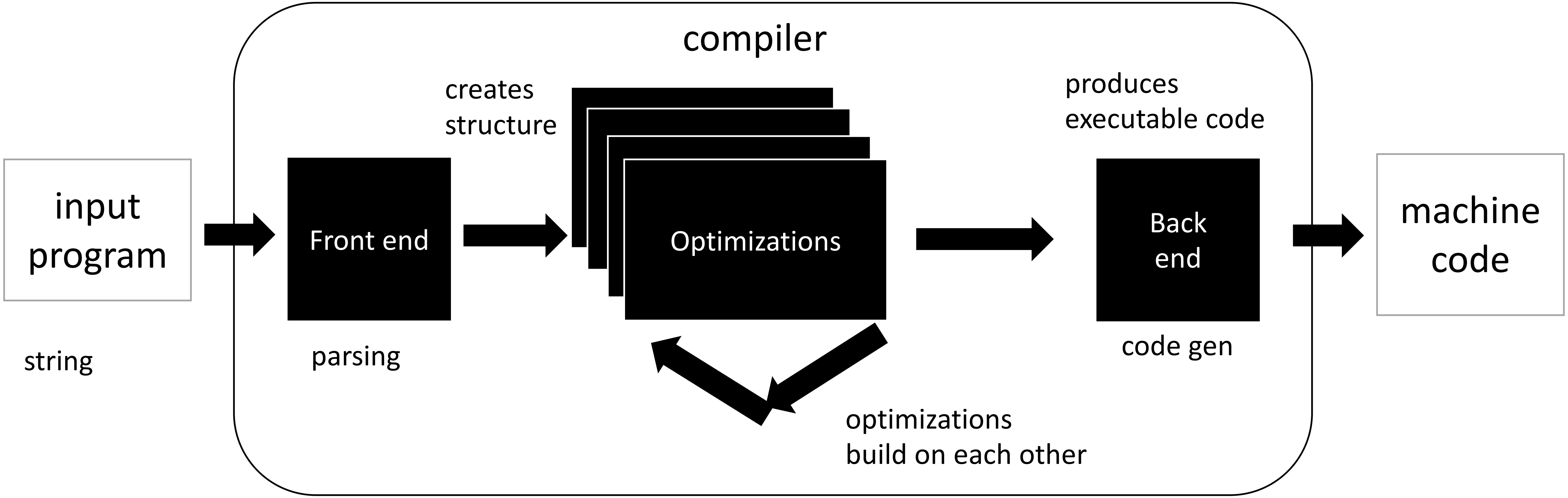
- We covered most of last lecture’s material in the quiz

New module!

- Intermediate representations

- Where are we at in our compiler flow?

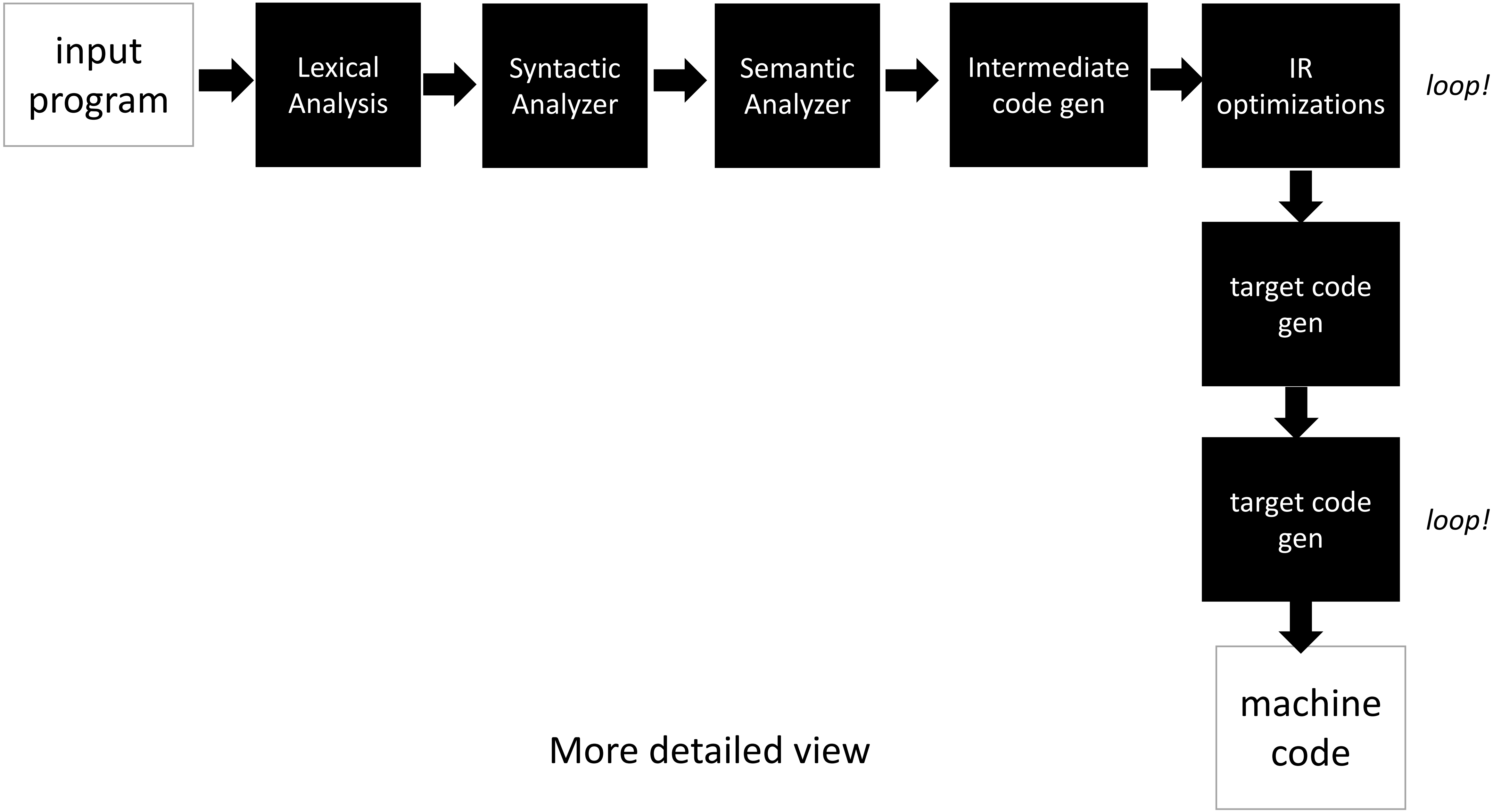
Compiler Architecture



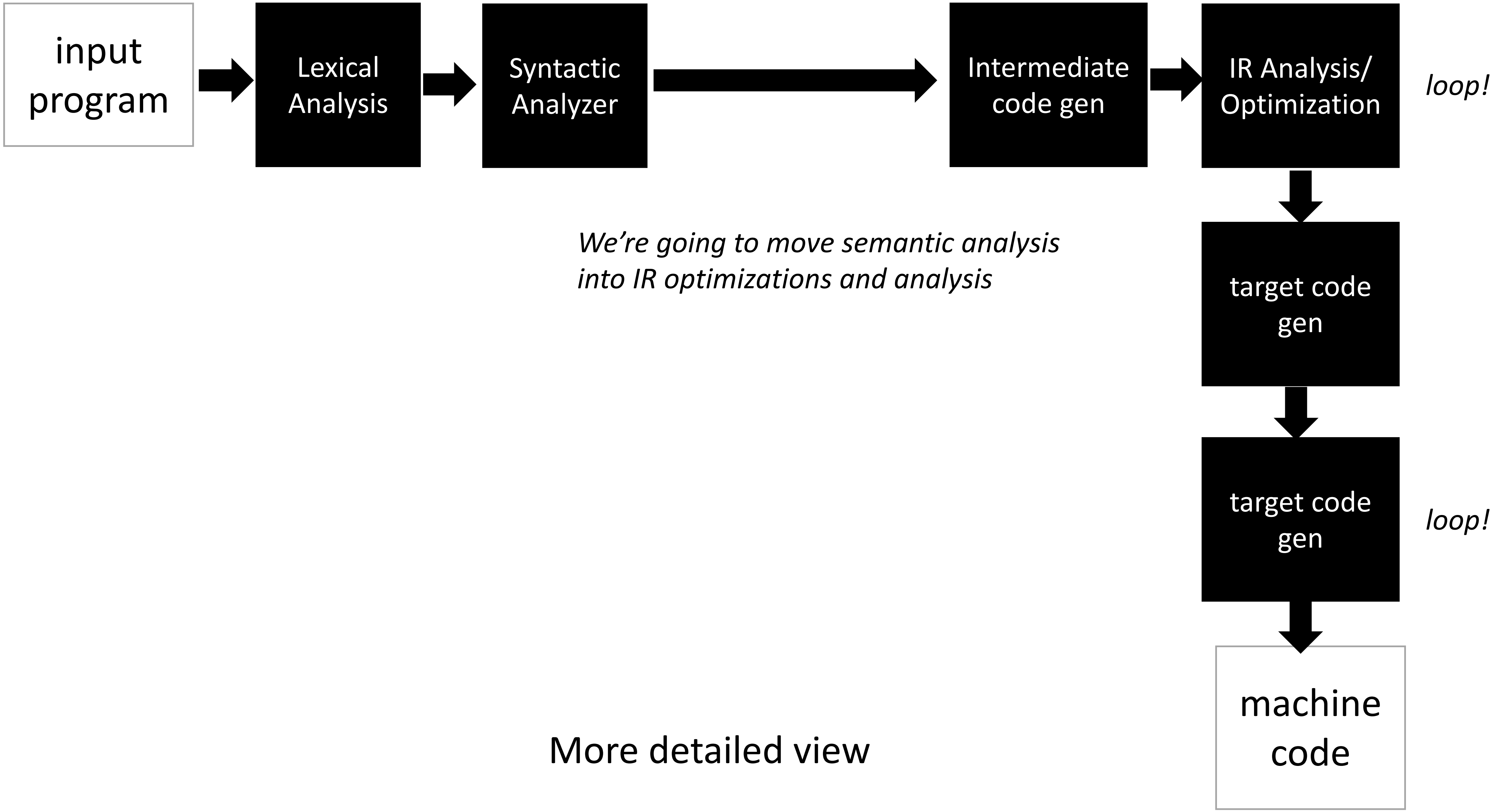
Medium detailed view

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

More detailed view

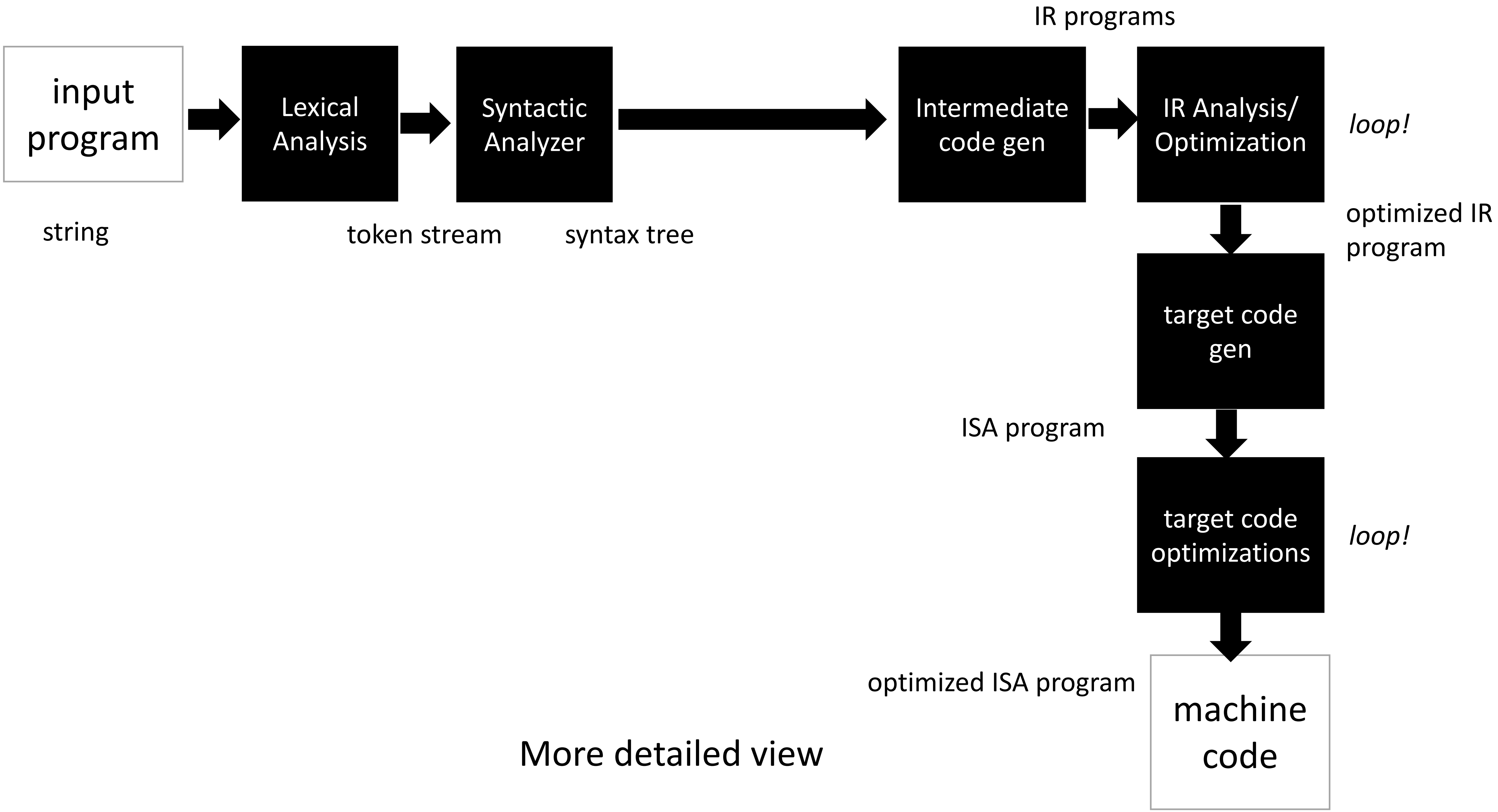


More detailed view

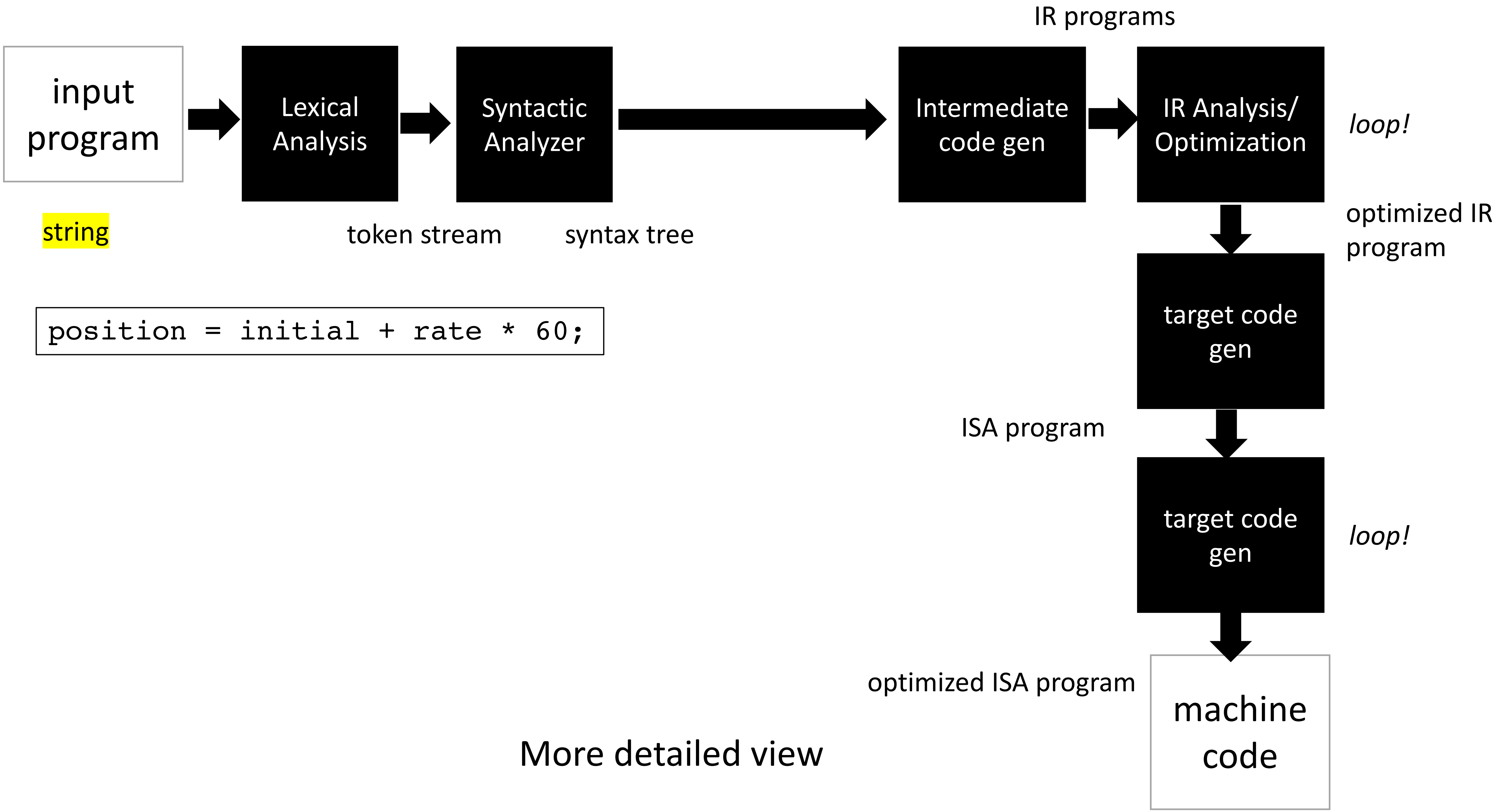


We’re going to move semantic analysis into IR optimizations and analysis

More detailed view

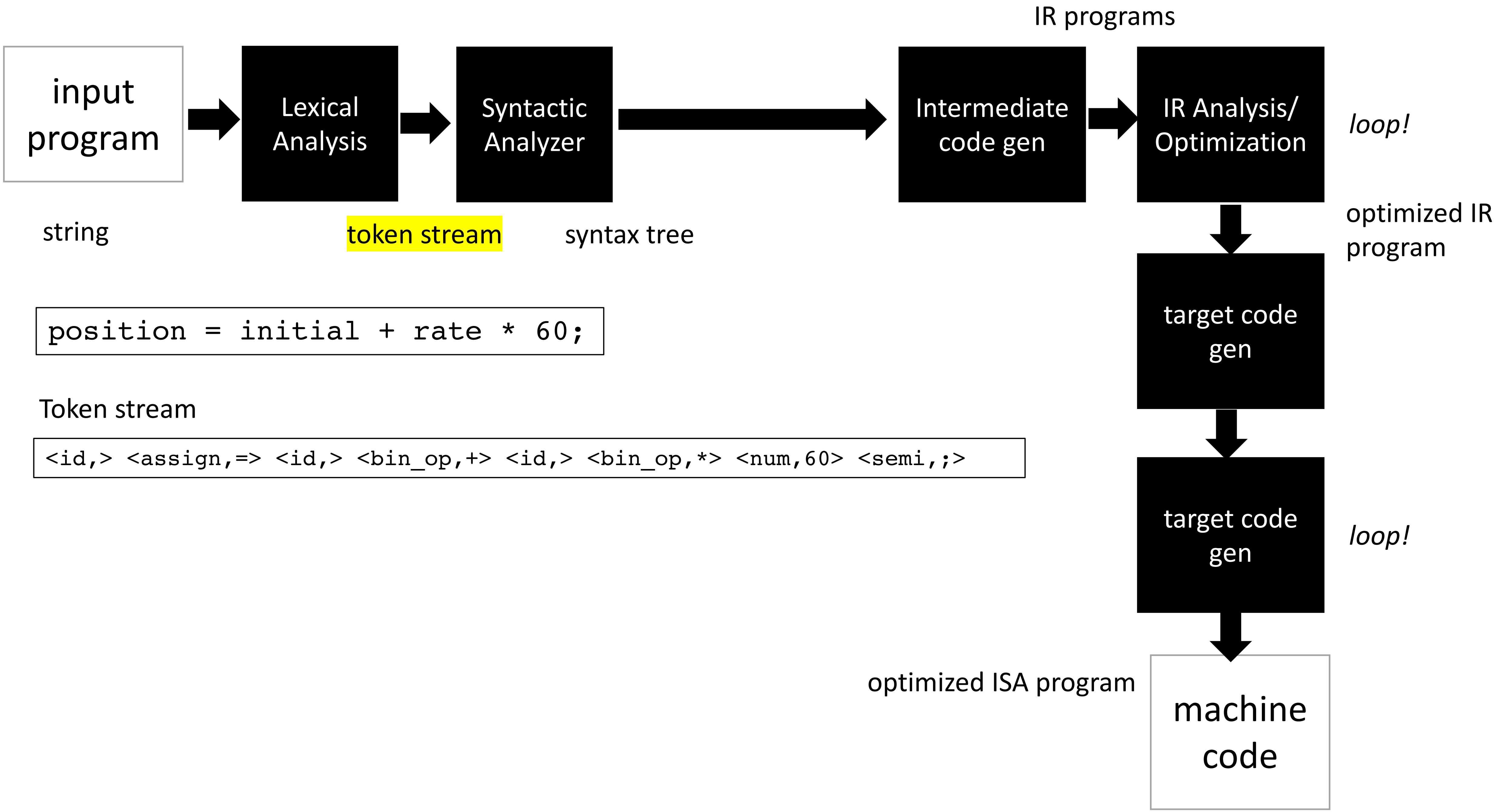


More detailed view




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

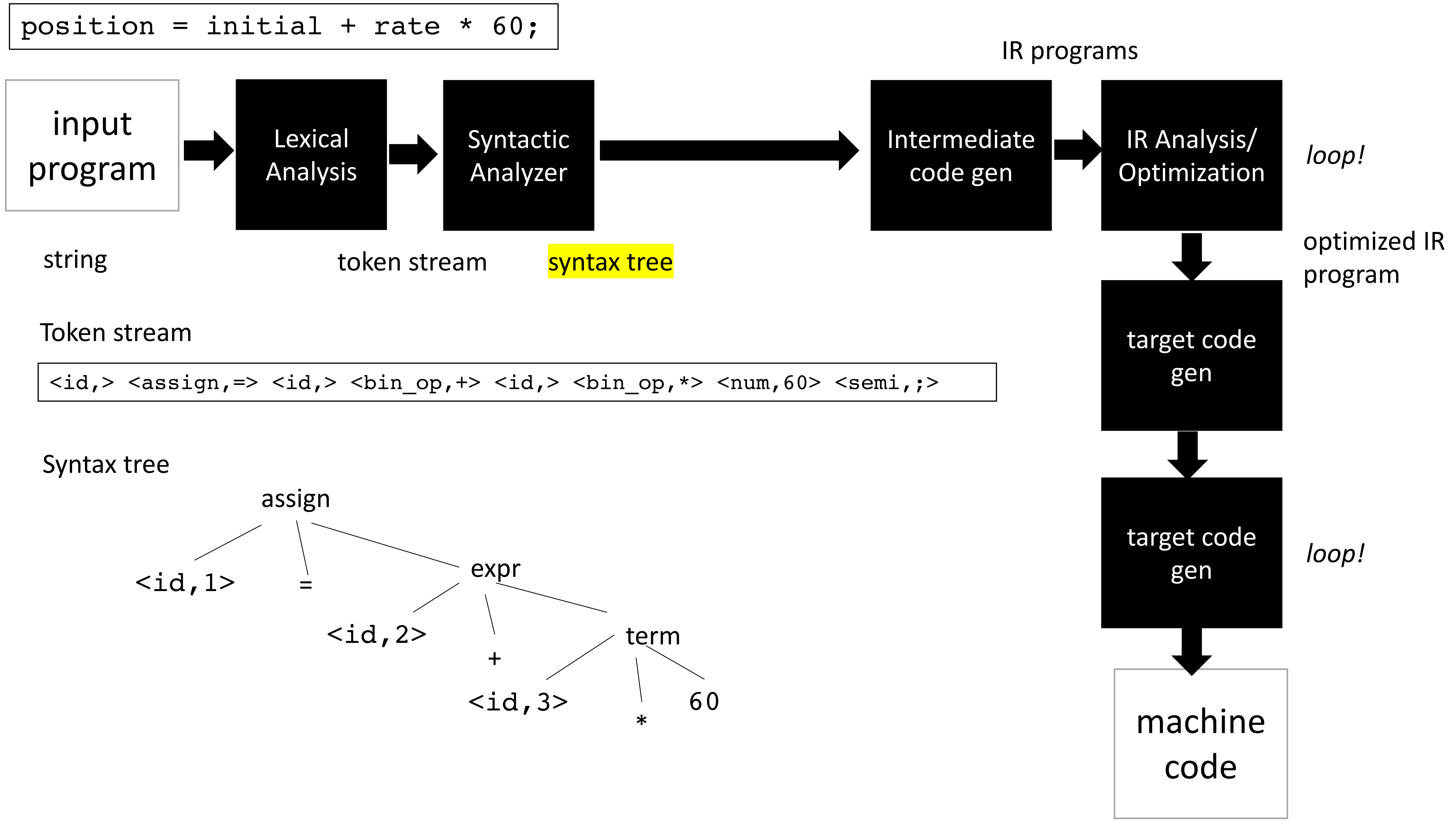
More detailed view




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

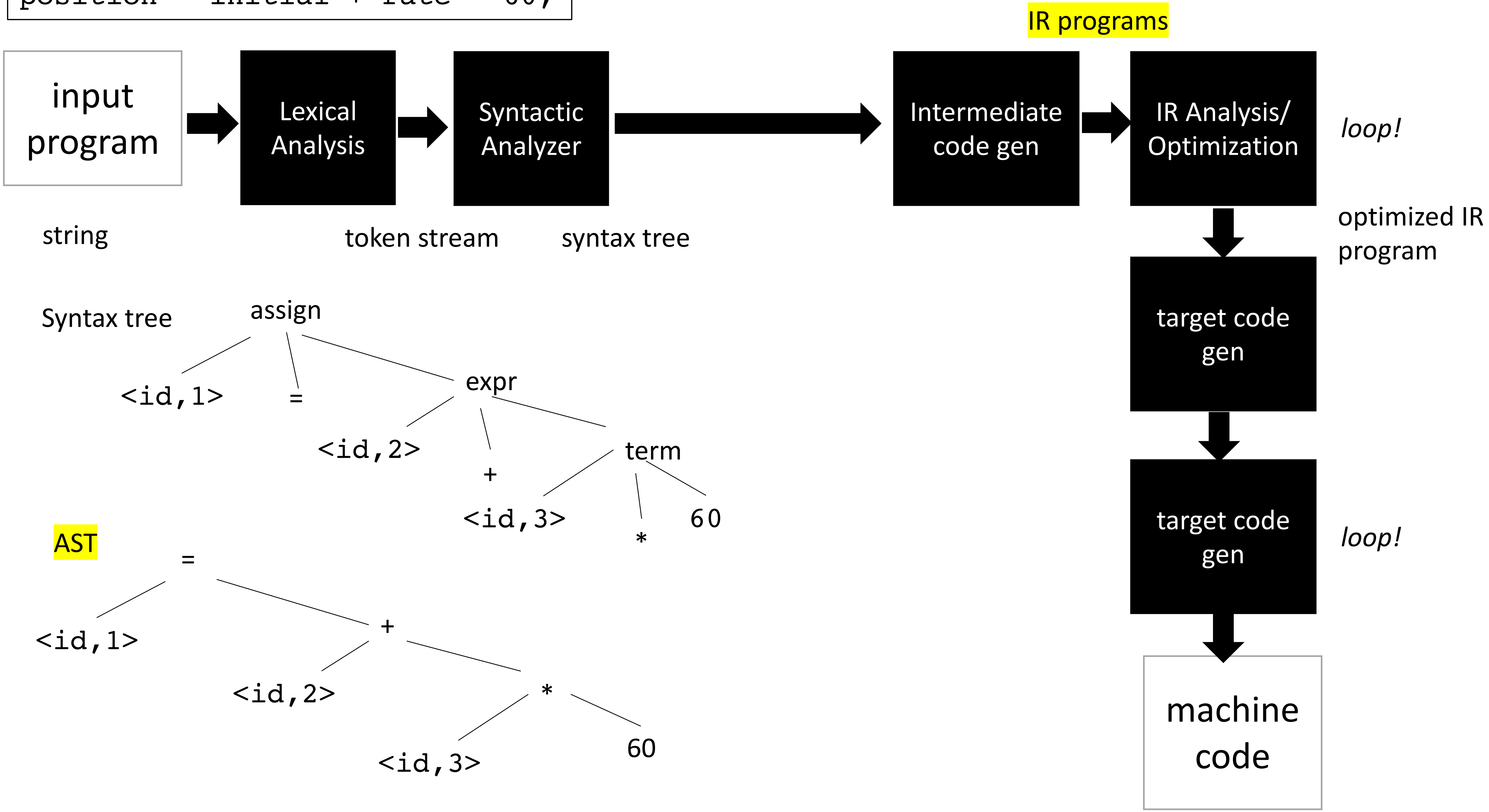
Token stream

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

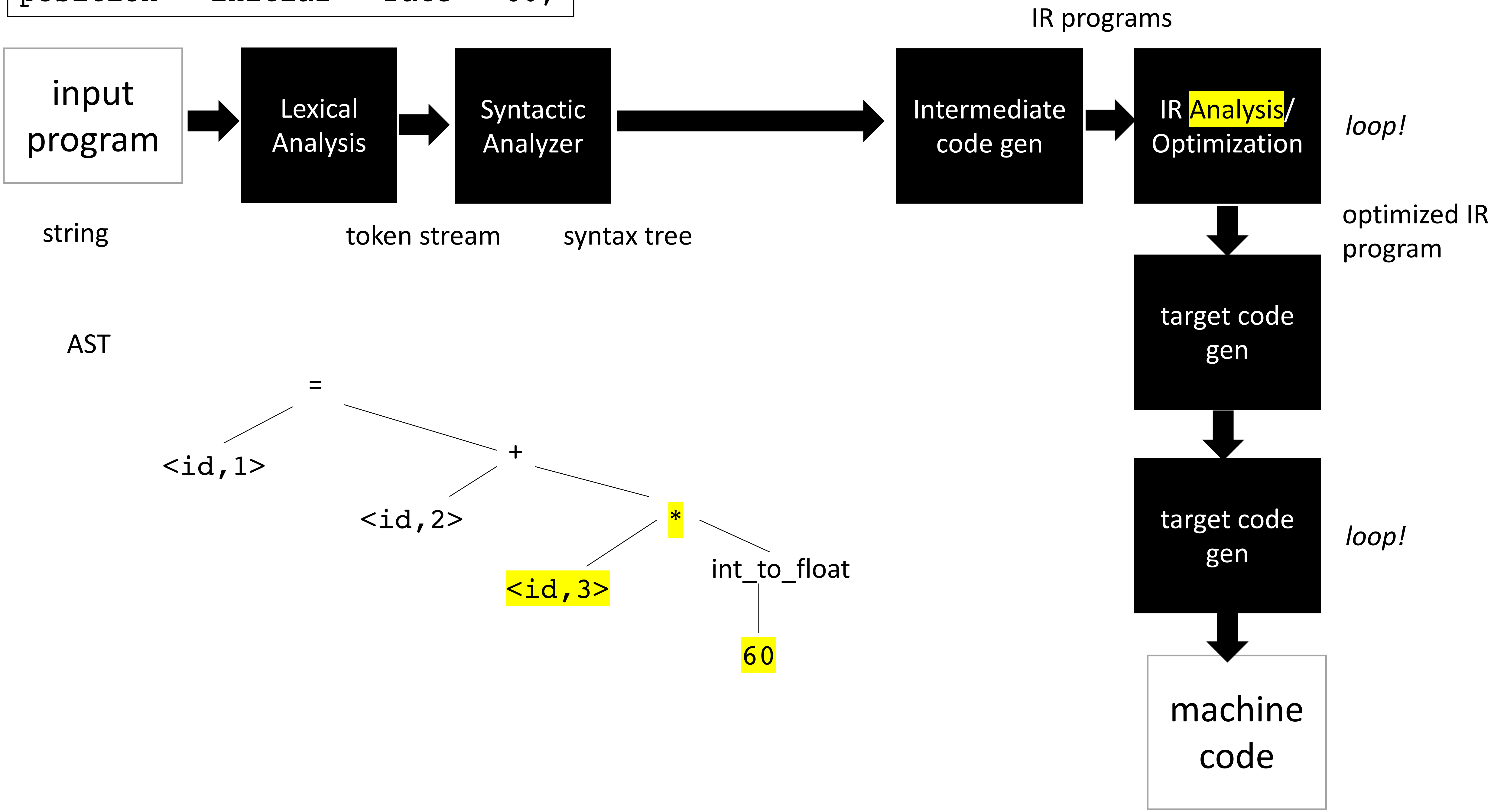


position = initial + rate * 60;
IR programs
input program
Lexical Analysis
Syntactic Analyzer
Intermediate code gen
IR Analysis/ Optimization
loop!
string
token stream
syntax tree
optimized IR program
Token stream
<id,> <assign,=> <id,> <bin_op,+> <id,> <bin_op,*> <num,60> <semi,;>
target code gen
Syntax tree
assign
<id,1>
=
expr
<id,2>
+
term
<id,3>
*
60
target code gen
loop!
machine code

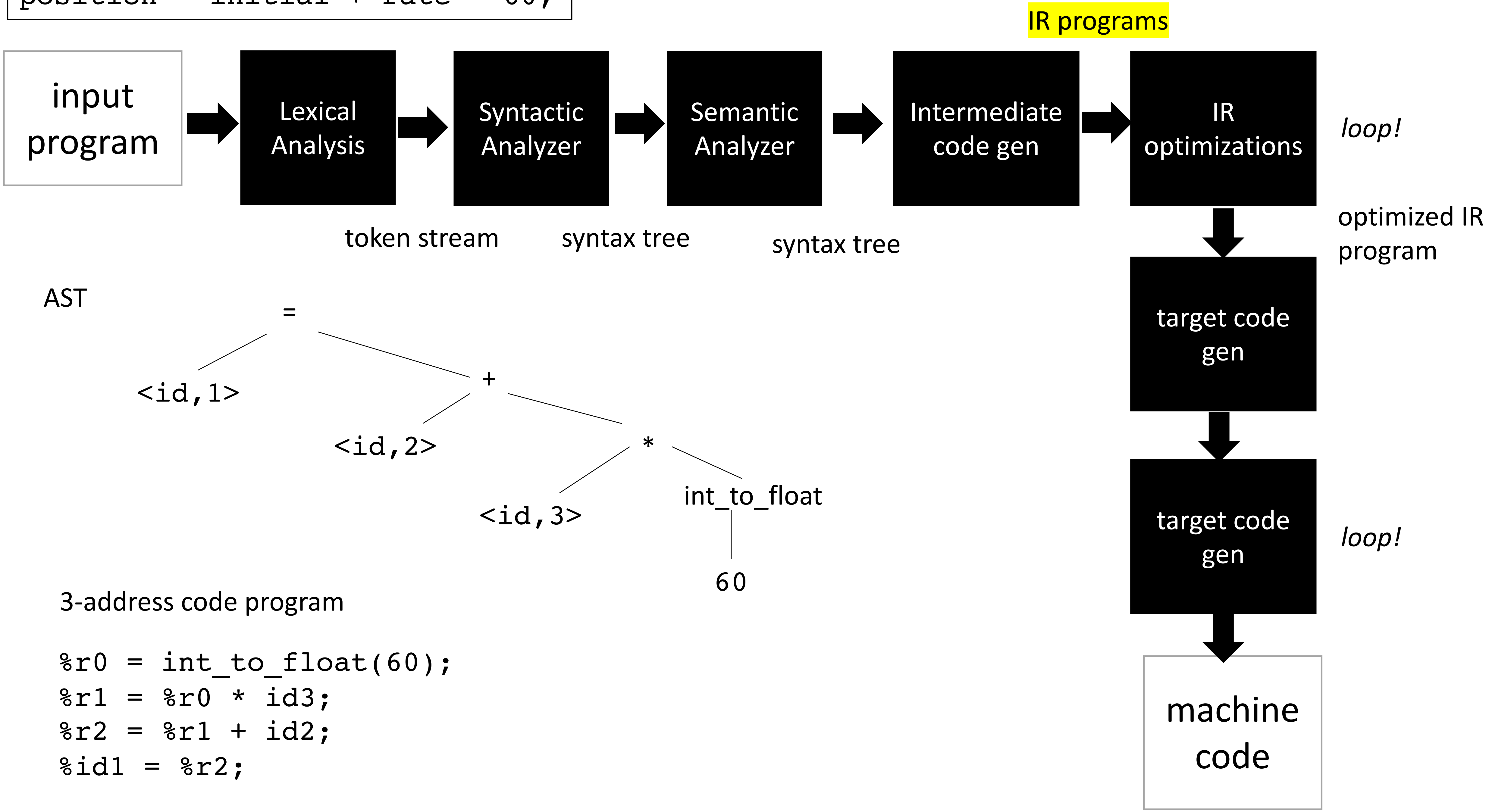
```
position = initial + rate * 60;
```



```
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

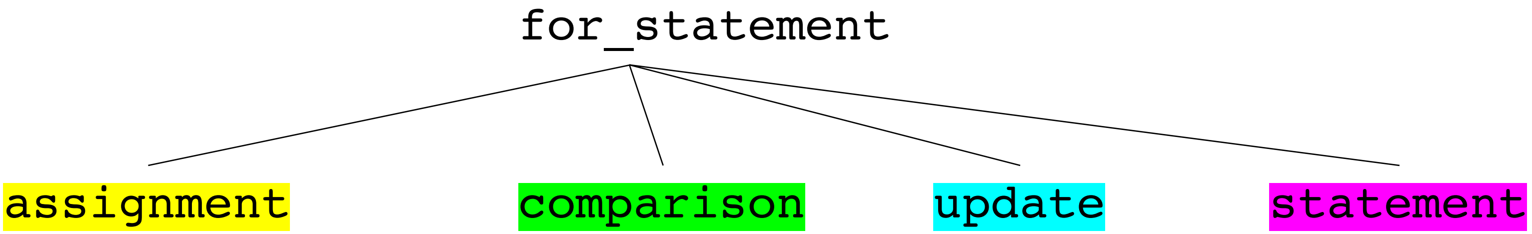
- Several forms:
  - tree - abstract syntax tree
  - graphs - control flow graph
  - linear program - 3 address code
- Often times the program is represented as a hybrid
  - graphs where nodes are a linear program
  - linear program where expressions are ASTs
- Progression:
  - start close to a parse tree
  - move closer to an ISA

Intermediate representations

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

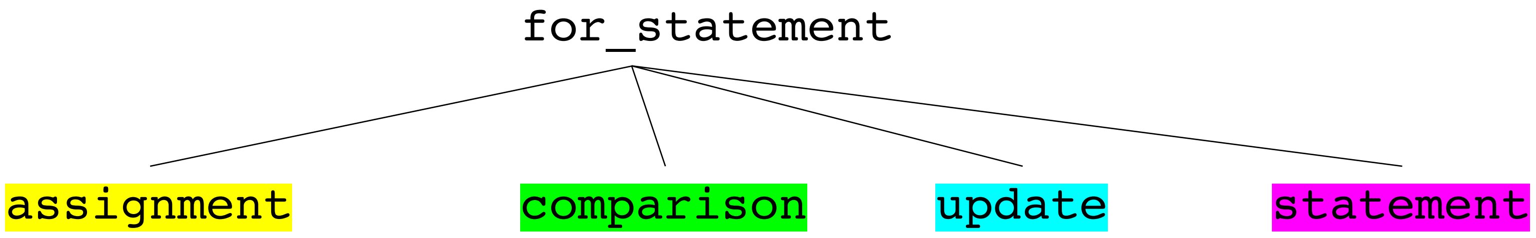
- Different optimizations and analysis are more suitable for IRs in different forms.

Example: loop unrolling




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

Example: loop unrolling




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

Check:

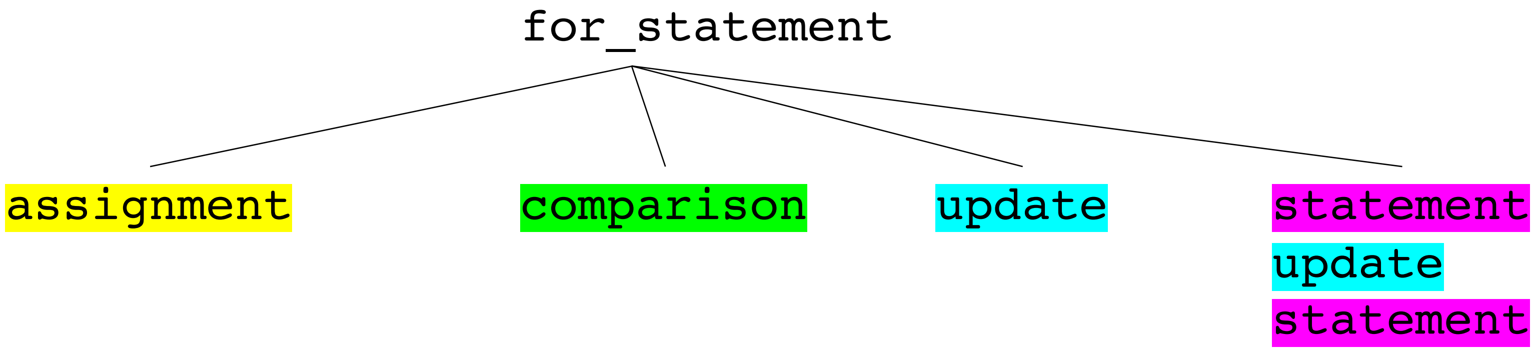
1. Find iteration variable by examining assignment, comparison and update.

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

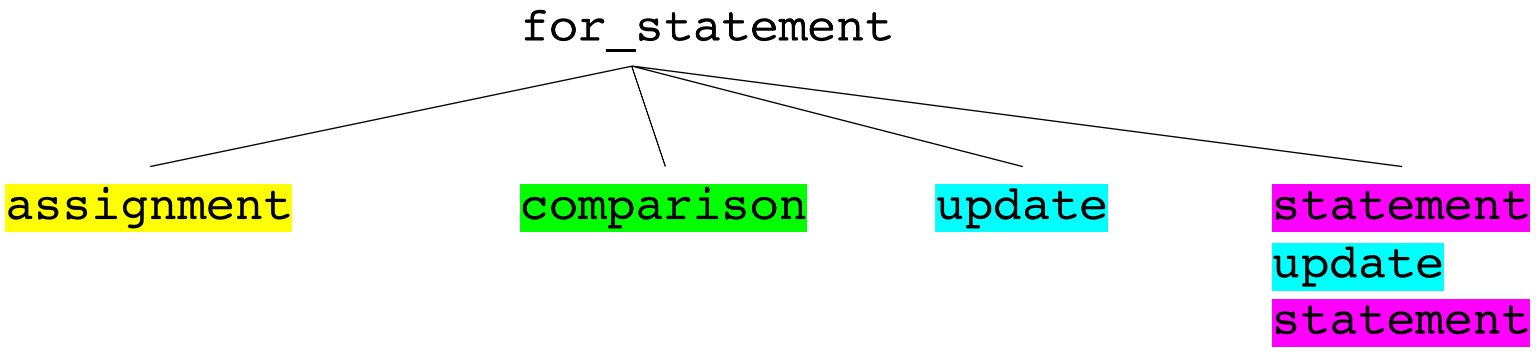
3. check that statement doesn't change i.

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

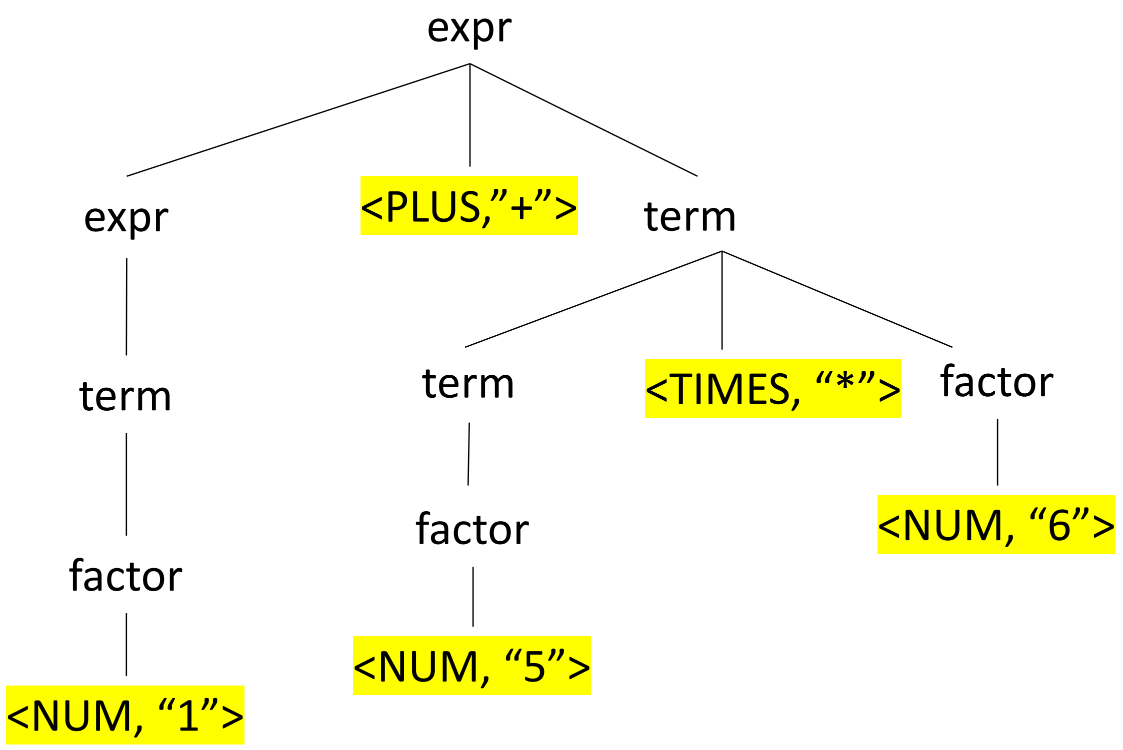
- One step away from parse trees
- Great representation for expressions
- Natural representation to apply type checking

What is an AST?

`input: 1+5*6`

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`

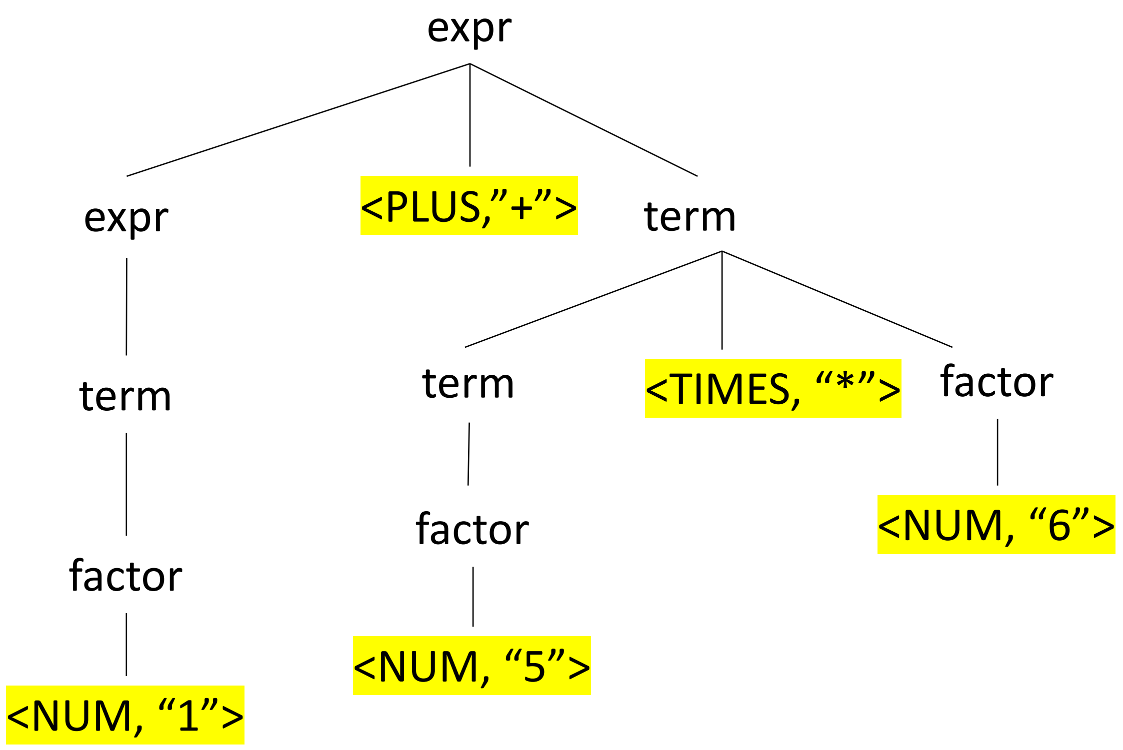


What is an AST?

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

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`



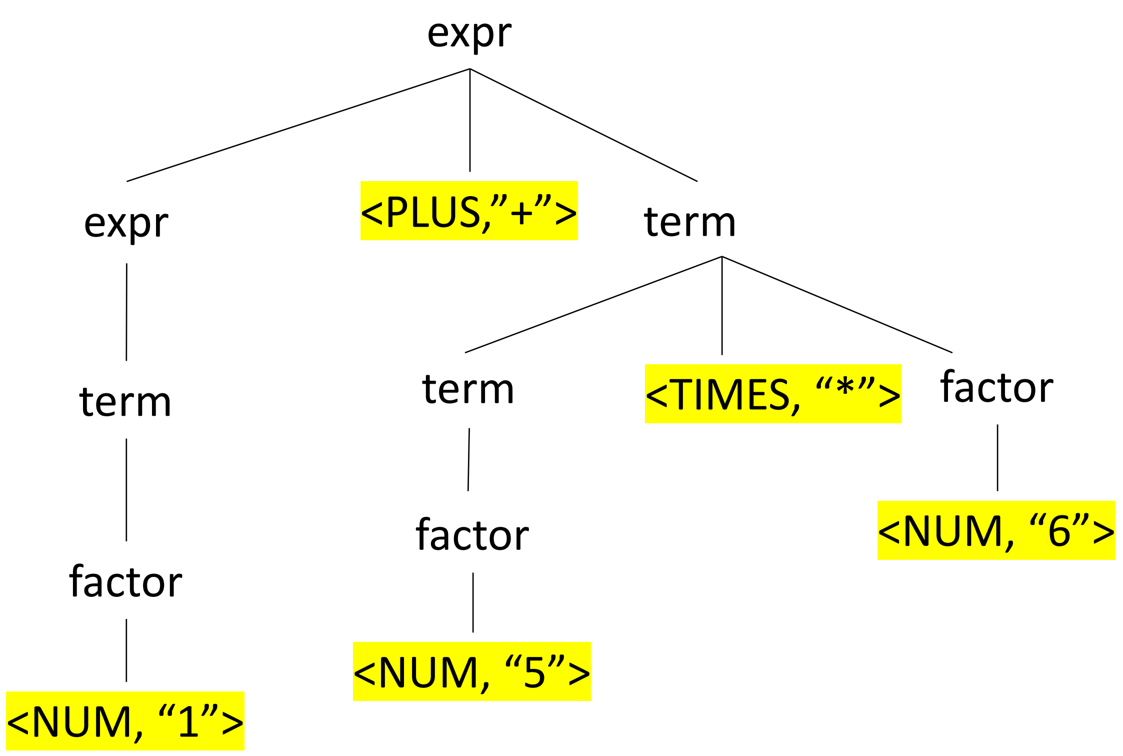
What are leaves?

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`



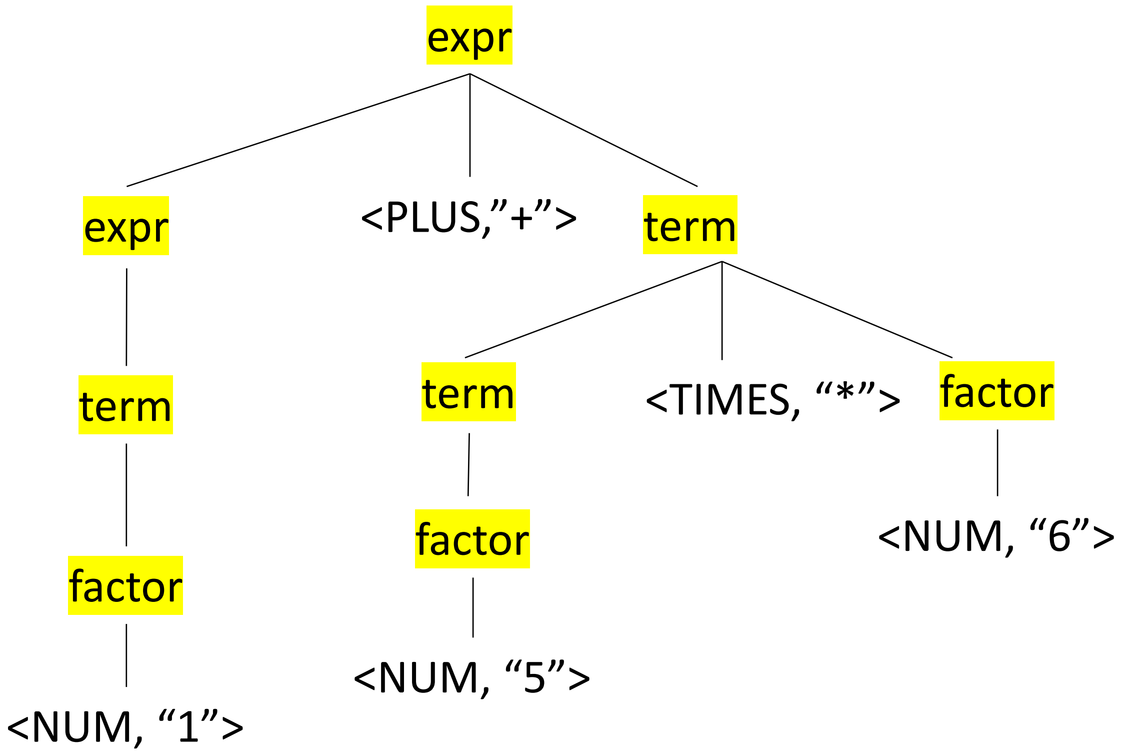
What are leaves? lexemes

What is an AST?

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

We’ll start by looking at a parse tree:

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>+</td><td><code>expr</code></td><td><code>: expr PLUS term</code>
<code>| term</code></td></tr>
<tr><td>*</td><td><code>term</code></td><td><code>: term TIMES factor</code>
<code>| factor</code></td></tr>
<tr><td>()</td><td><code>factor</code></td><td><code>: LPAREN expr RPAREN</code>
<code>| NUM</code></td></tr>
</table>



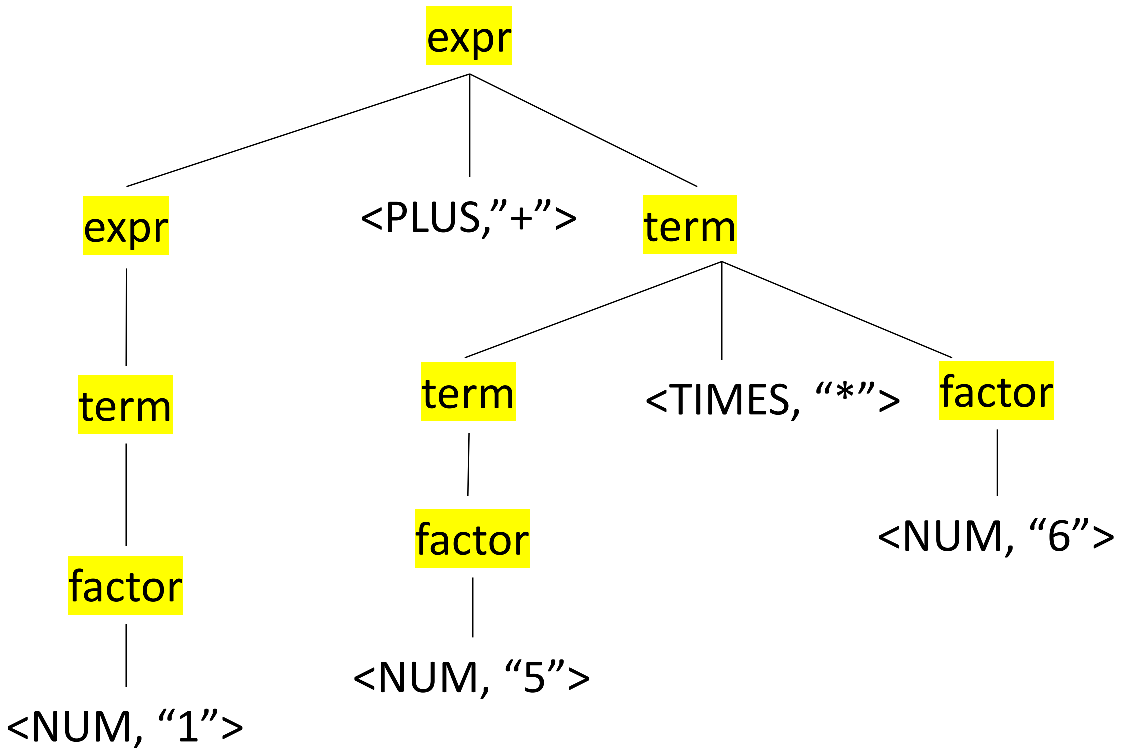
What are nodes?

What is an AST?

We’ll start by looking at a parse tree:

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>+</td><td><code>expr</code></td><td><code>: expr PLUS term</code>
<code>| term</code></td></tr>
<tr><td>*</td><td><code>term</code></td><td><code>: term TIMES factor</code>
<code>| factor</code></td></tr>
<tr><td>()</td><td><code>factor</code></td><td><code>: LPAREN expr RPAREN</code>
<code>| NUM</code></td></tr>
</table>

`input: 1+5*6`



What are nodes? non-terminals

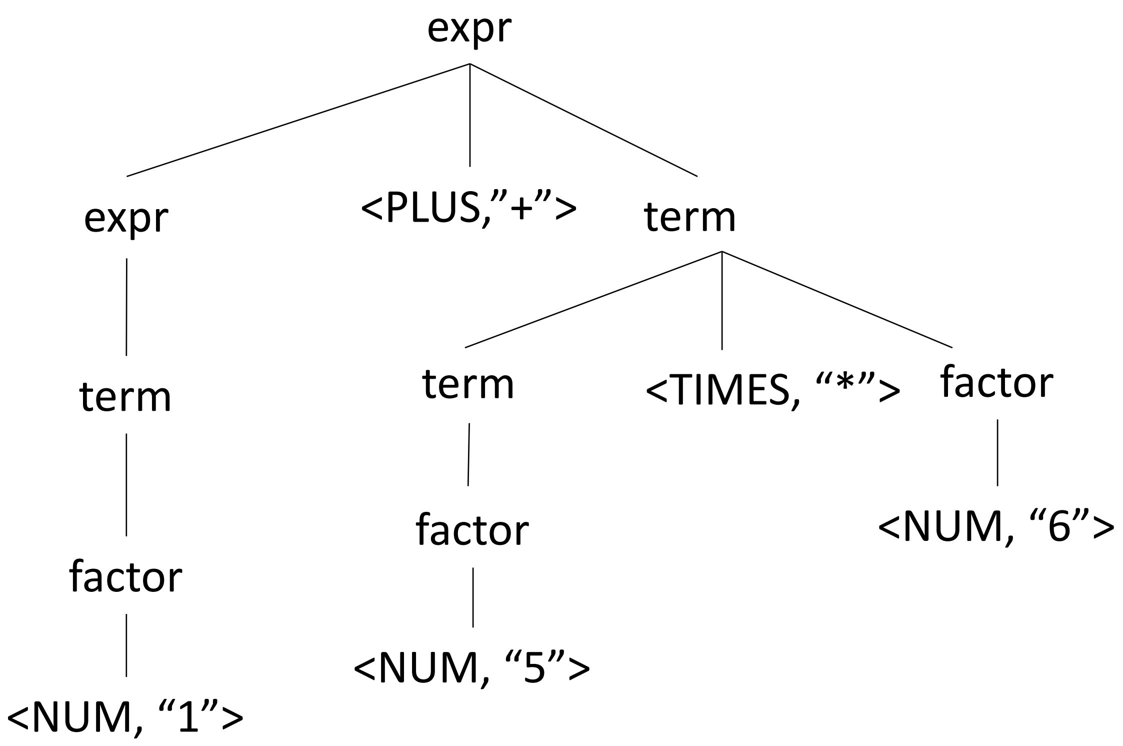
What is an AST?

`input: 1+5*6`

Parse trees are defined by the grammar

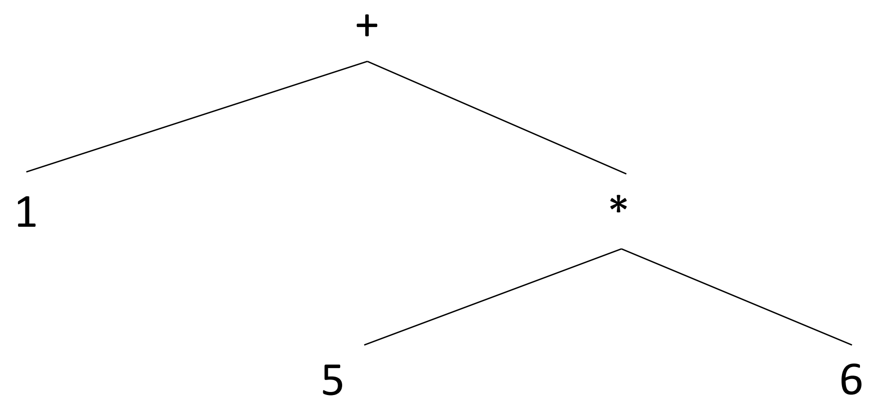
- **Tokens**
- **Production rules**

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



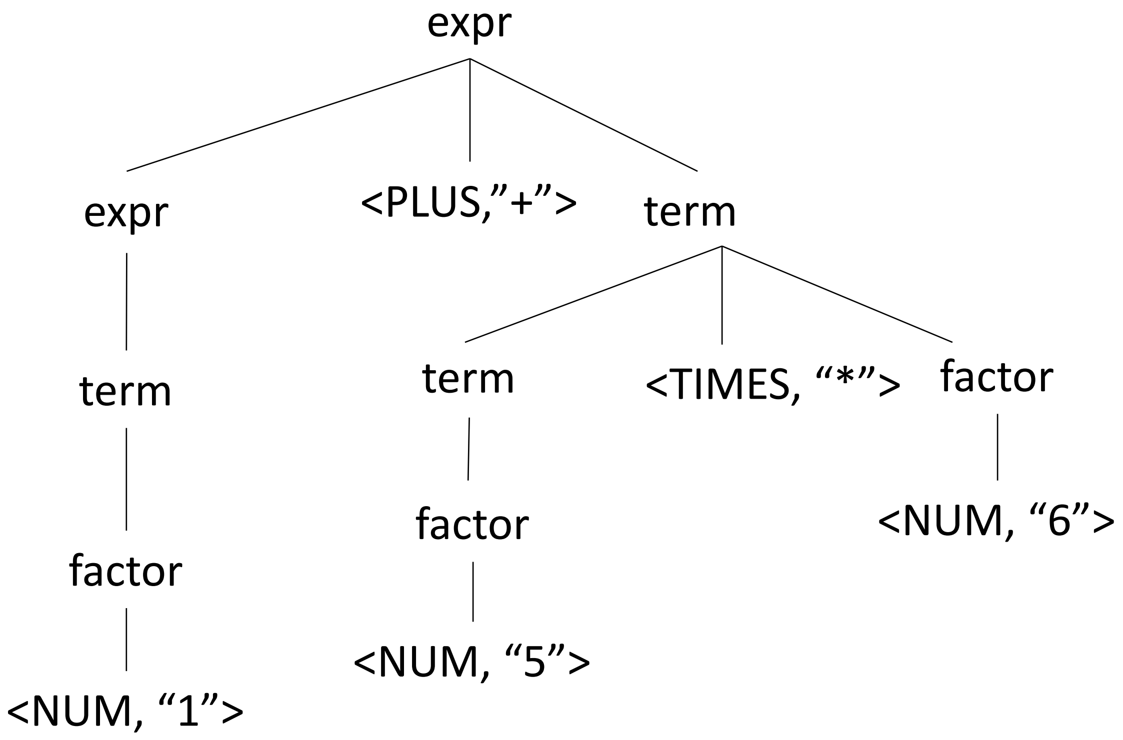
What is an AST?

`input: 1+5*6`

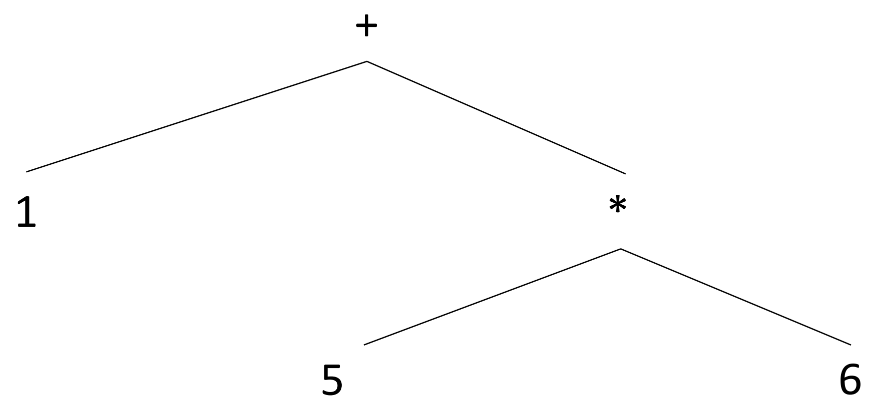


AST

What are some differences?



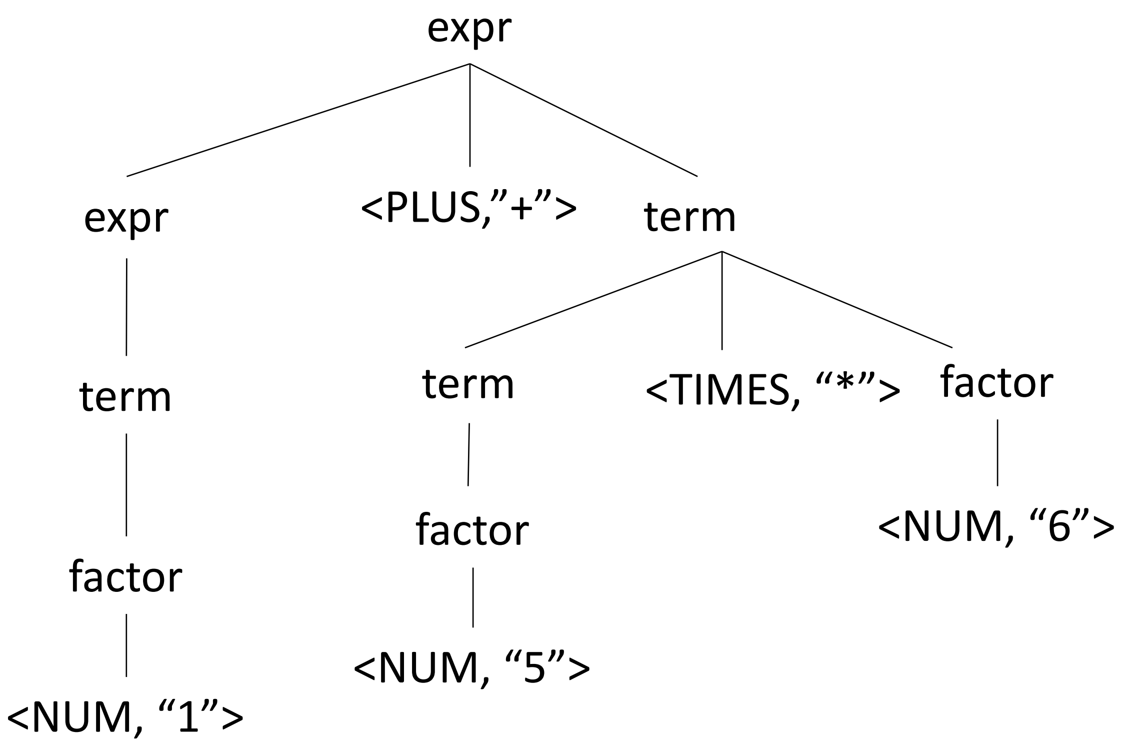
What is an AST?



AST

What are some differences?
- disjoint from the grammar
- leaves are data, not lexemes
- nodes are operators, not non-terminals

`input: 1+5*6`

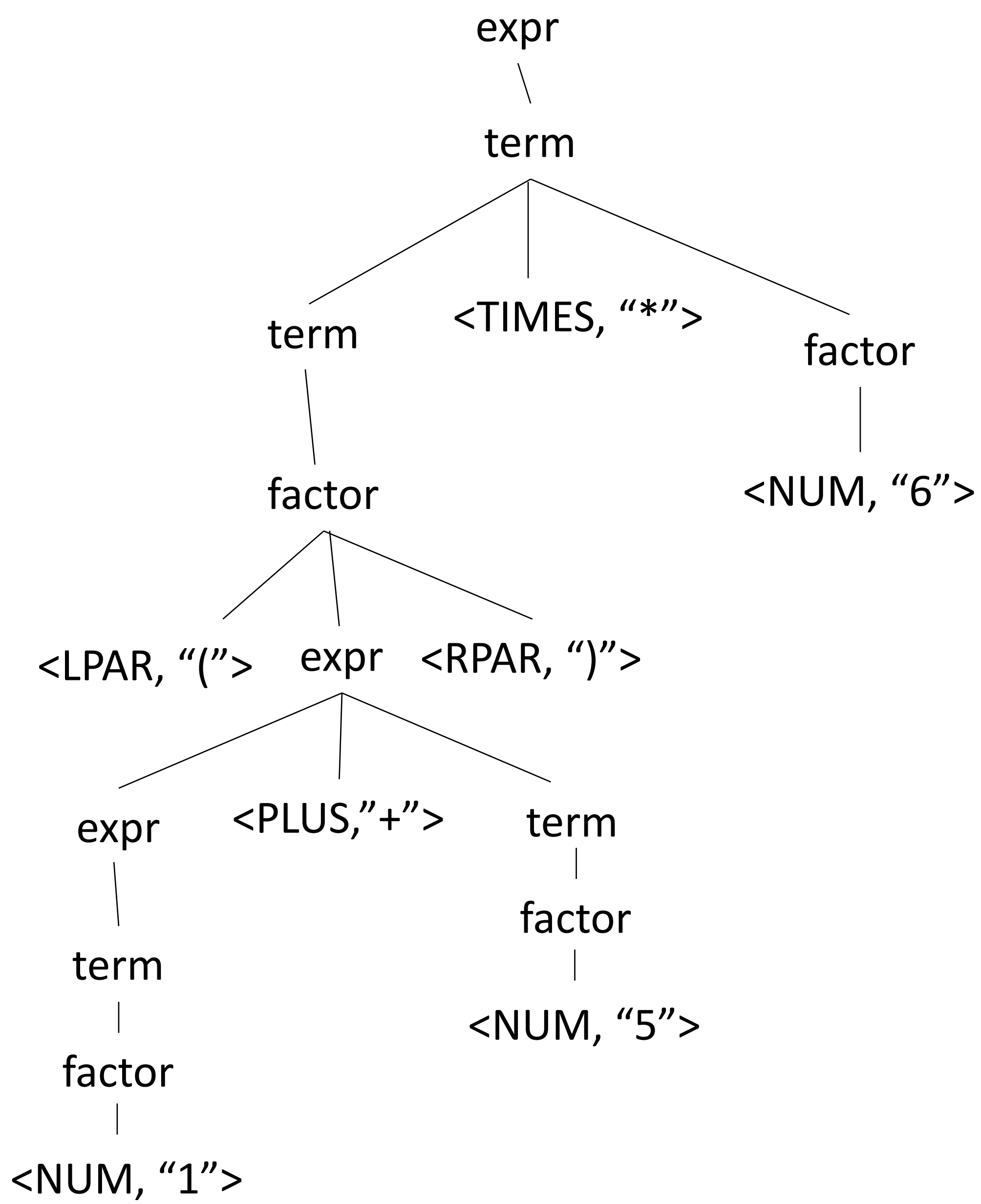


Example

`input: (1+5)*6`

what happens to ()s in an AST?

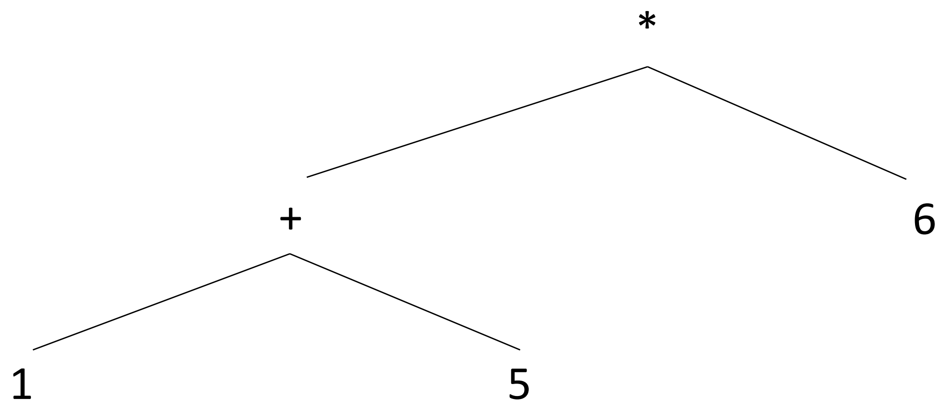
Operator	Name	Productions
+	`expr`	`: expr PLUS term` `| term`
*	`term`	`: term TIMES factor` `| factor`
()	`factor`	`: LPAR expr RPAR` `| NUM`



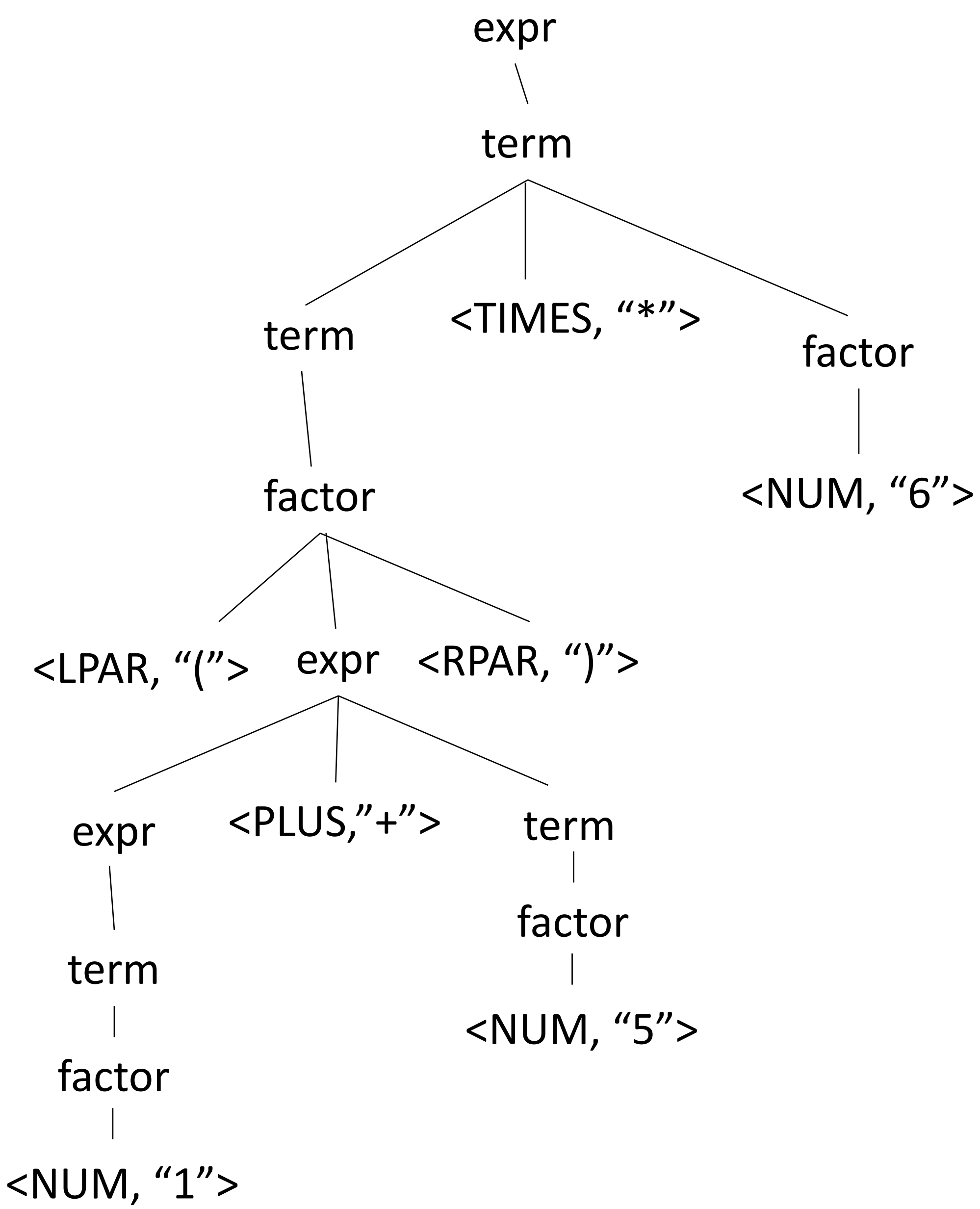
Example

`input: (1+5)*6`

what happens to ()s in an AST?



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



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 (`+`,`*`)

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 (`+`,`*`)

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

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


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


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

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


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


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

Creating an AST from production rules

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th><th>Production action</th></tr>
<tr><td>+</td><td>expr</td><td>: expr PLUS term
| term</td><td>{}
{}</td></tr>
<tr><td>*</td><td>term</td><td>: term TIMES factor
| factor</td><td>{}
{}</td></tr>
<tr><td>()</td><td>factor</td><td>: LPAR expr RPAR
| NUM
| ID</td><td>{}
{}
{}</td></tr>
</table>

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}`

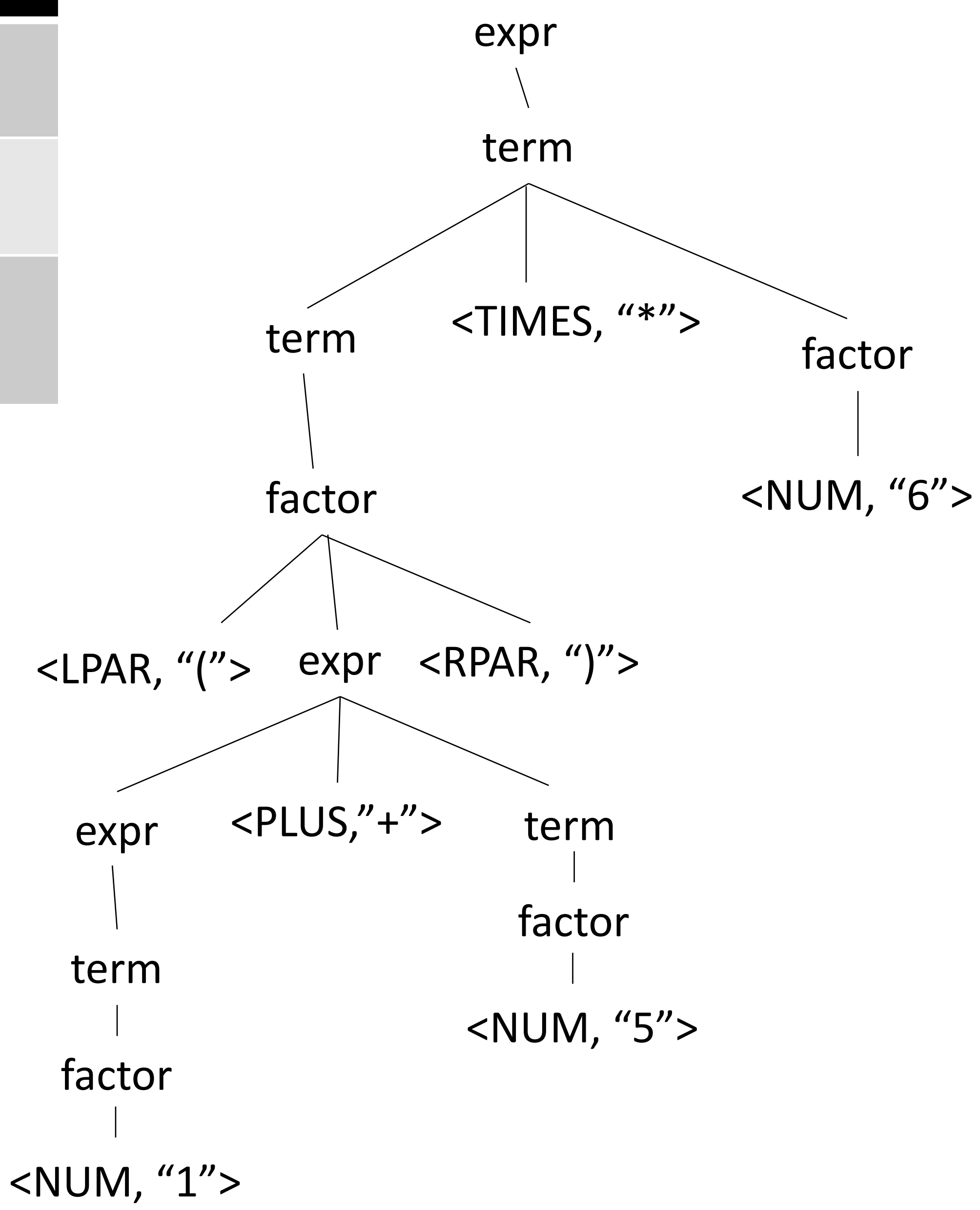
()	`factor`	`: LPAR expr RPAR`
		`| NUM`
		`| ID`
			`{return $2}`
			`{return ASTNumNode($1)}`
			`{return ASTIDNode($1)}`

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)}`

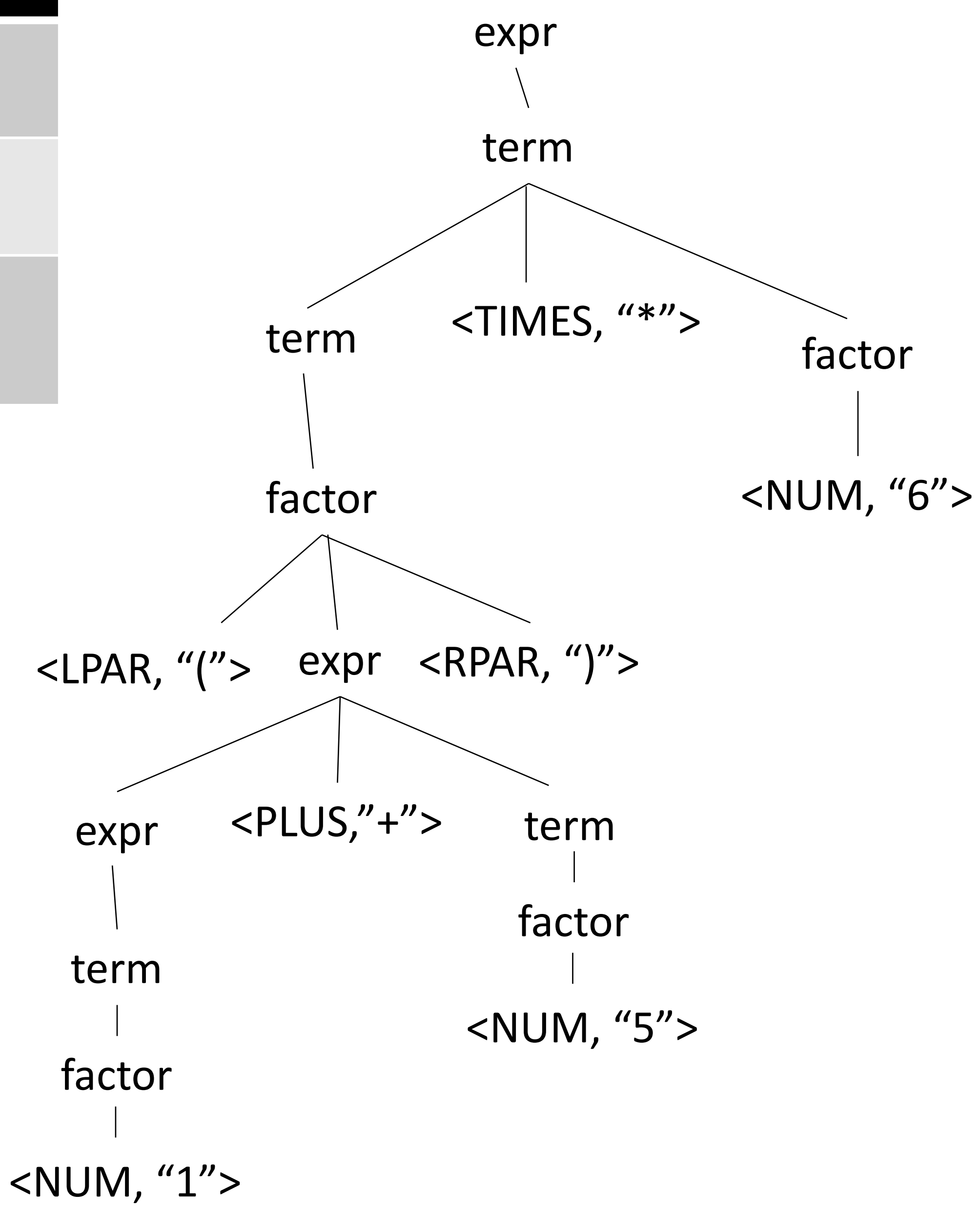
Lets build the AST

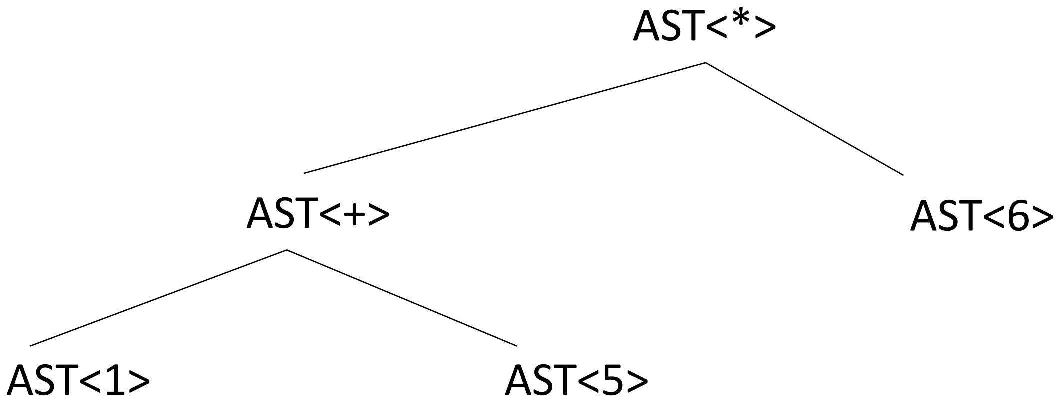
`input: (1+5)*6`



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

input: (1+5)*6

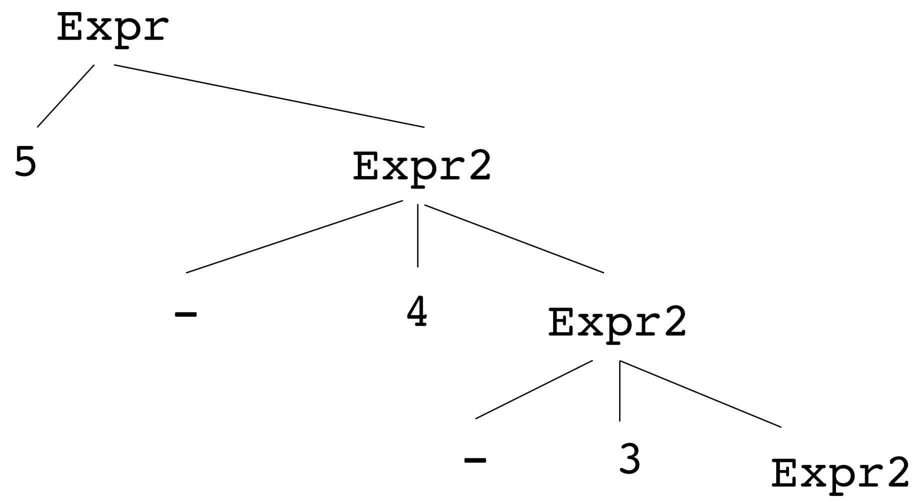




Creating an AST from top down grammar

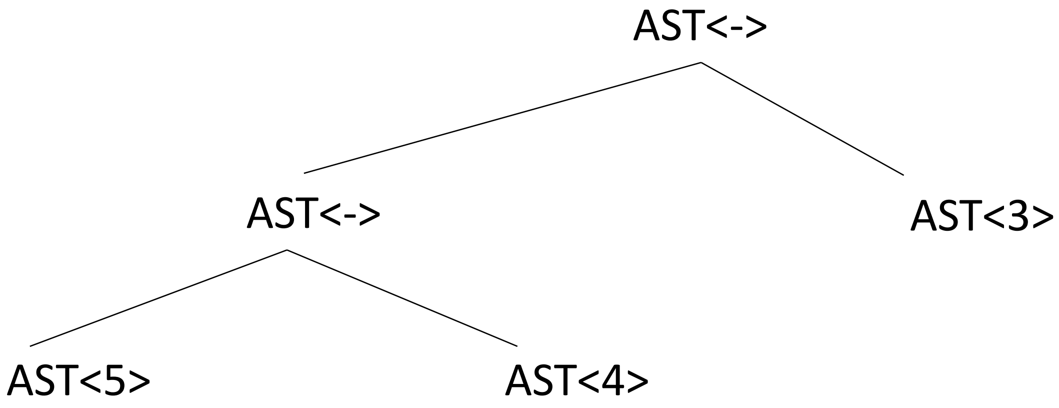
```
Expr  ::= NUM Expr2
Expr2 ::= MINUS NUM Expr2
       |    ""
```

5 - 4 - 3

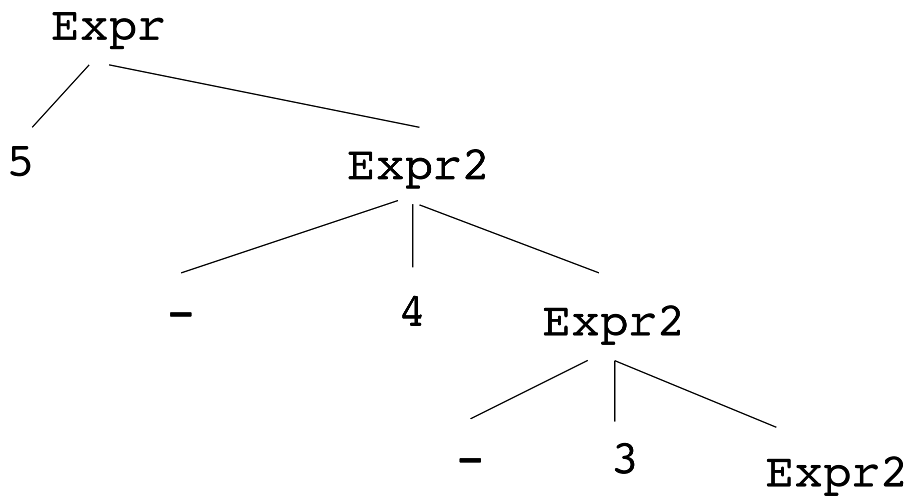


Creating an AST from top down grammar

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



`5 - 4 - 3`

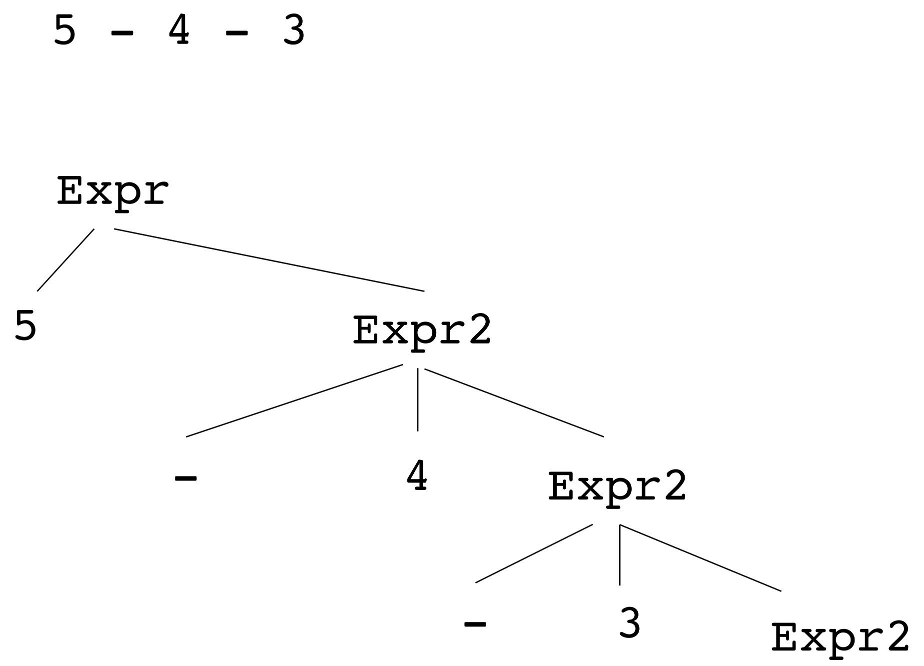


How do we get to the desired parse tree?

Creating an AST from top down grammar

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

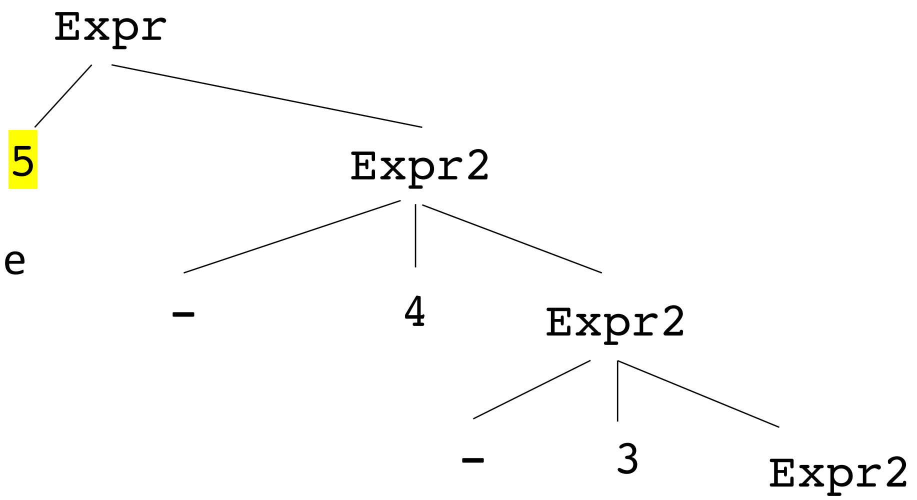
Keep in mind that because we wrote our own parser, we can inject code at any point during the parse.



Creating an AST from top down grammar

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

5 - 4 - 3

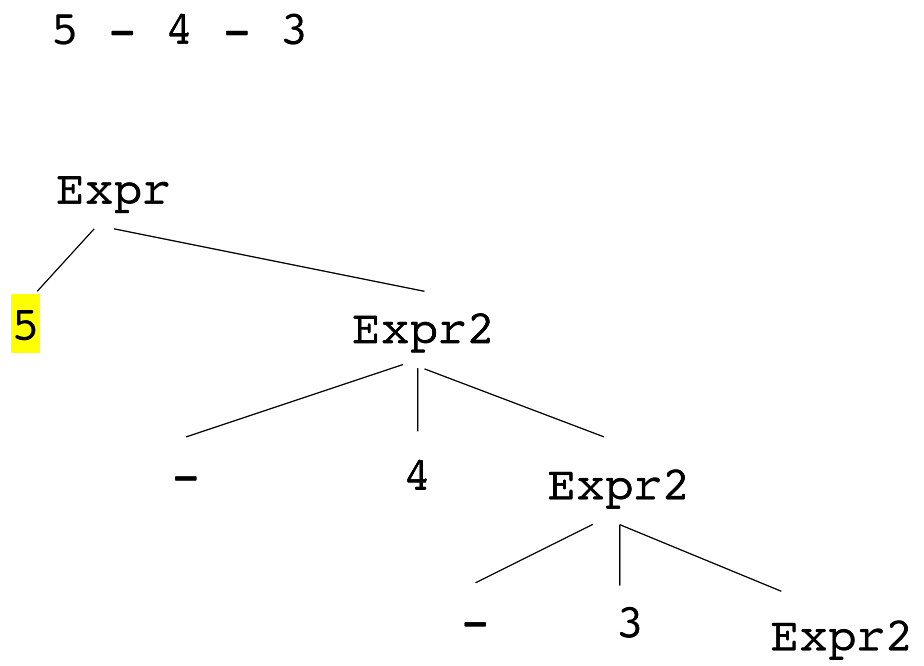


Get number node after we see a number

AST<5>

Creating an AST from top down grammar

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



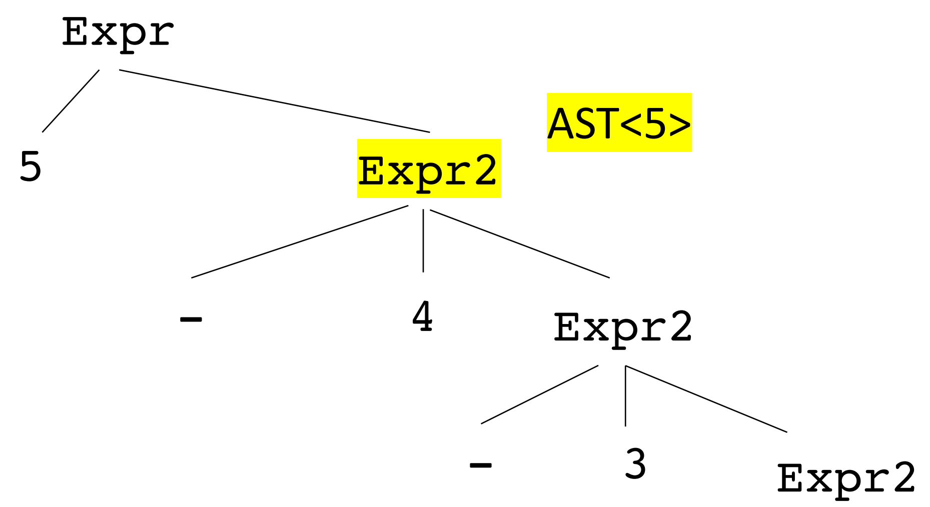
Pass the node down

AST<5>

Creating an AST from top down grammar

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

5 - 4 - 3



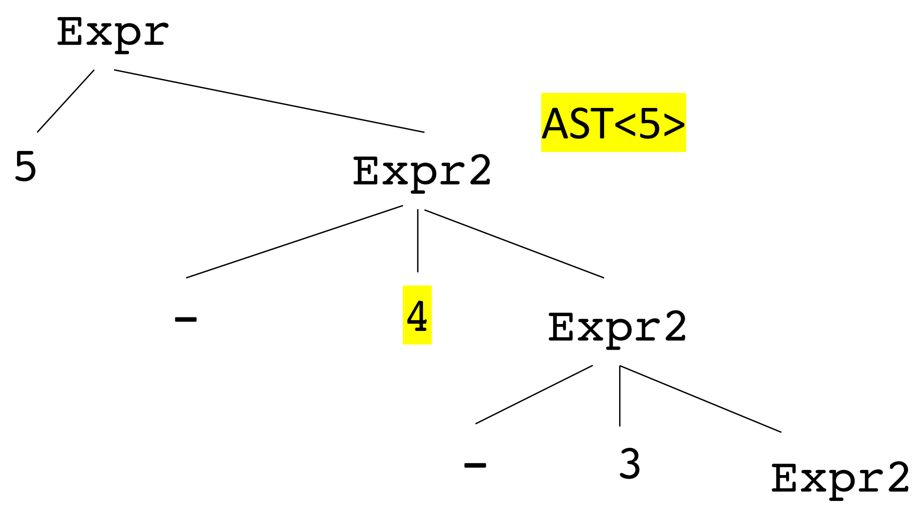
Pass the node down

AST<5>

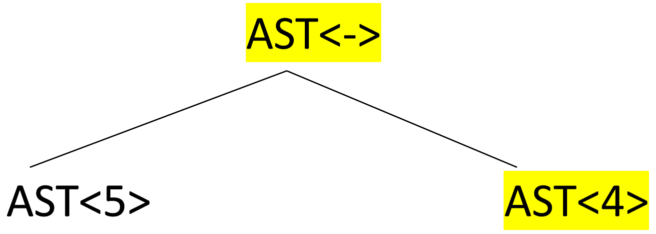
Creating an AST from top down grammar

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

5 - 4 - 3



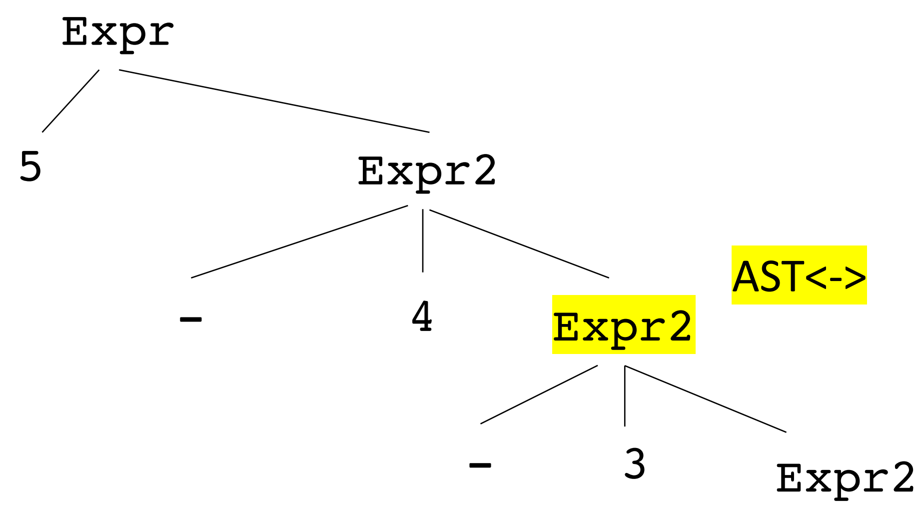
In Expr2, after 4 is parsed, create a number node and a minus node



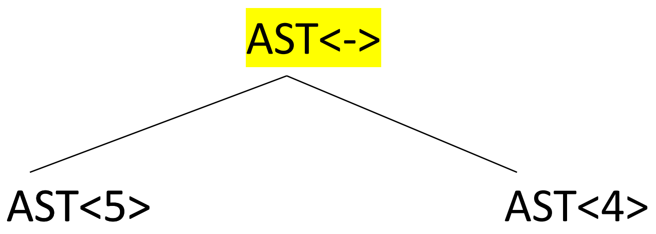
Creating an AST from top down grammar

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

5 - 4 - 3



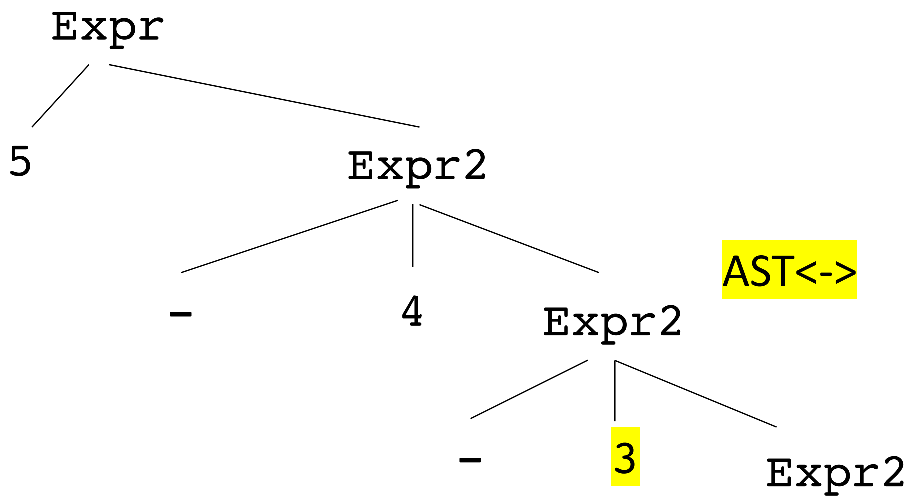
pass the new node down

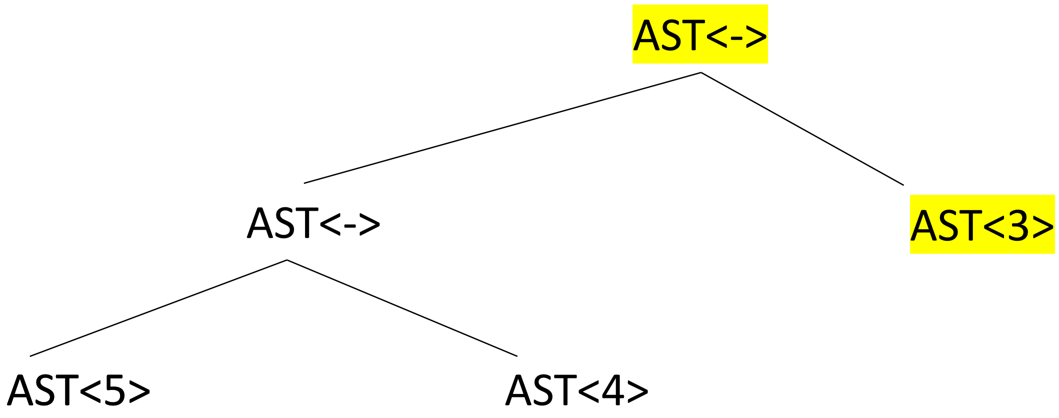


Creating an AST from top down grammar

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

5 - 4 - 3



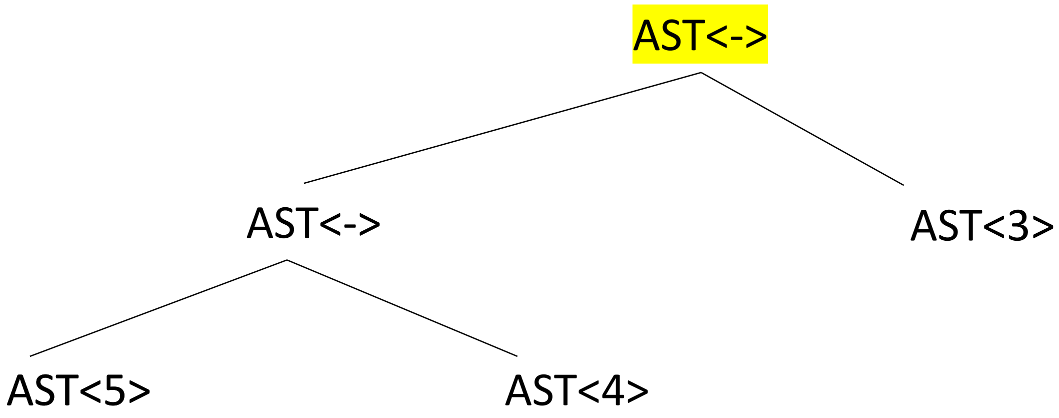


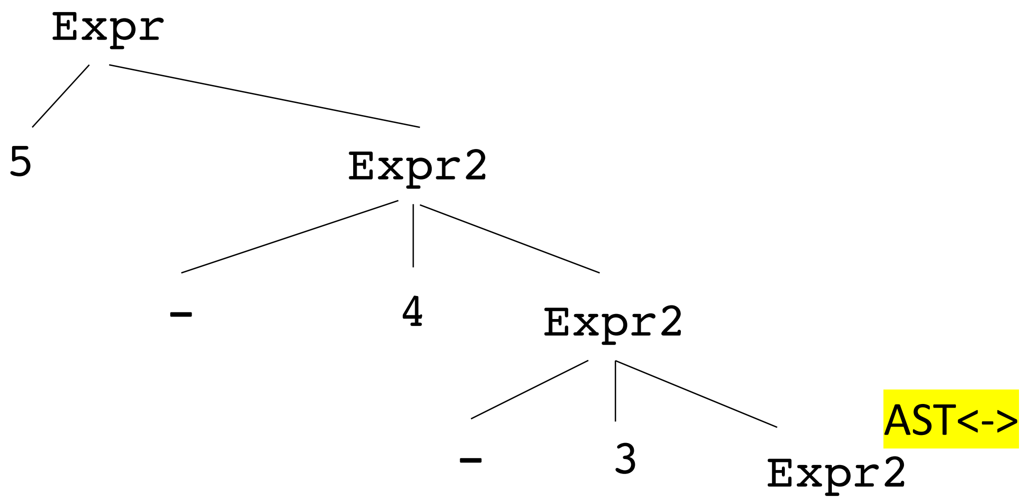
In Expr2, after 3 is parsed, create a number node and a minus node

Creating an AST from top down grammar

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

5 - 4 - 3



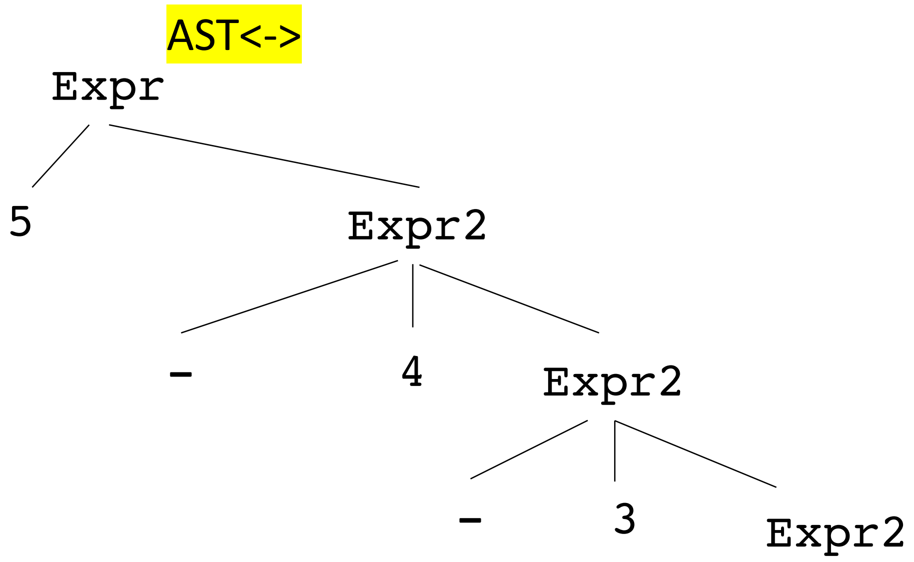


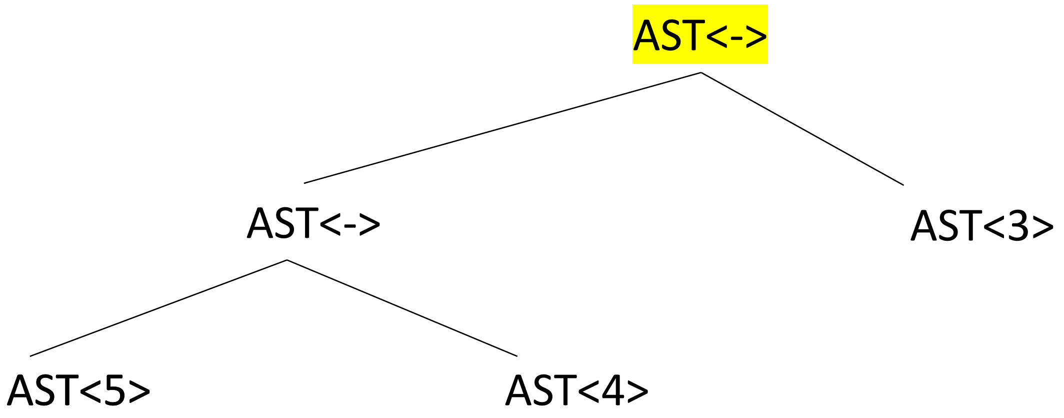
pass down the new node

Creating an AST from top down grammar

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

5 - 4 - 3





return the node when there is nothing left to parse

Creating an AST from top down grammar

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

```
def parse_expr(self):
    #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 top down 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 top down 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 top down grammar

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

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

how to adapt?

```
def parse_expr2(self, lhs_node):
        # ... for applying the first production rule
        self.eat("MINUS")
        value = self.next_word[1]
        rhs_node = ASTNumNode(value)
        self.eat("NUM")
        node = ASTMinusNode(lhs_node, rhs_node)
        return self.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.

See everyone on Friday

- We will discuss type checking on ASTs