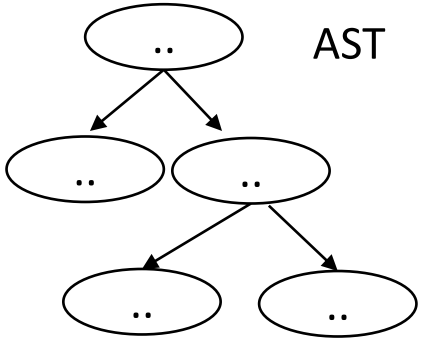
CSE110A: Compilers

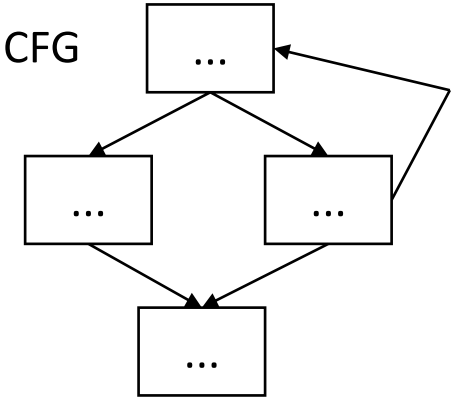


AST

Topics: Final Review

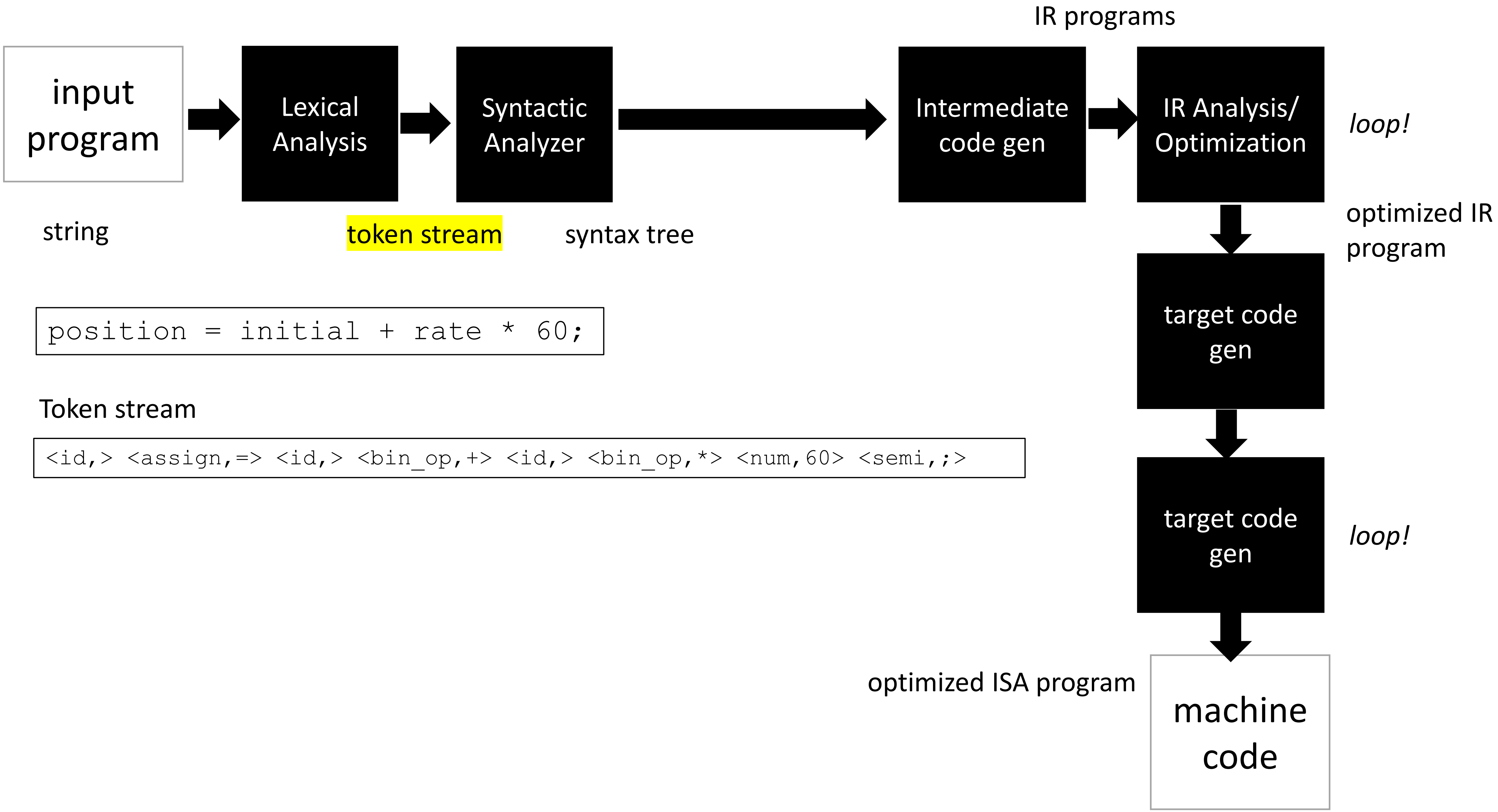
- *ASTs and Type Inferencing*
- *Intermediate Representations*
- *Basic Blocks*
- *Control Flow Graphs*
- *Optimizations: Local Value Numbering*
- *Optimizations: Loop Unrolling*

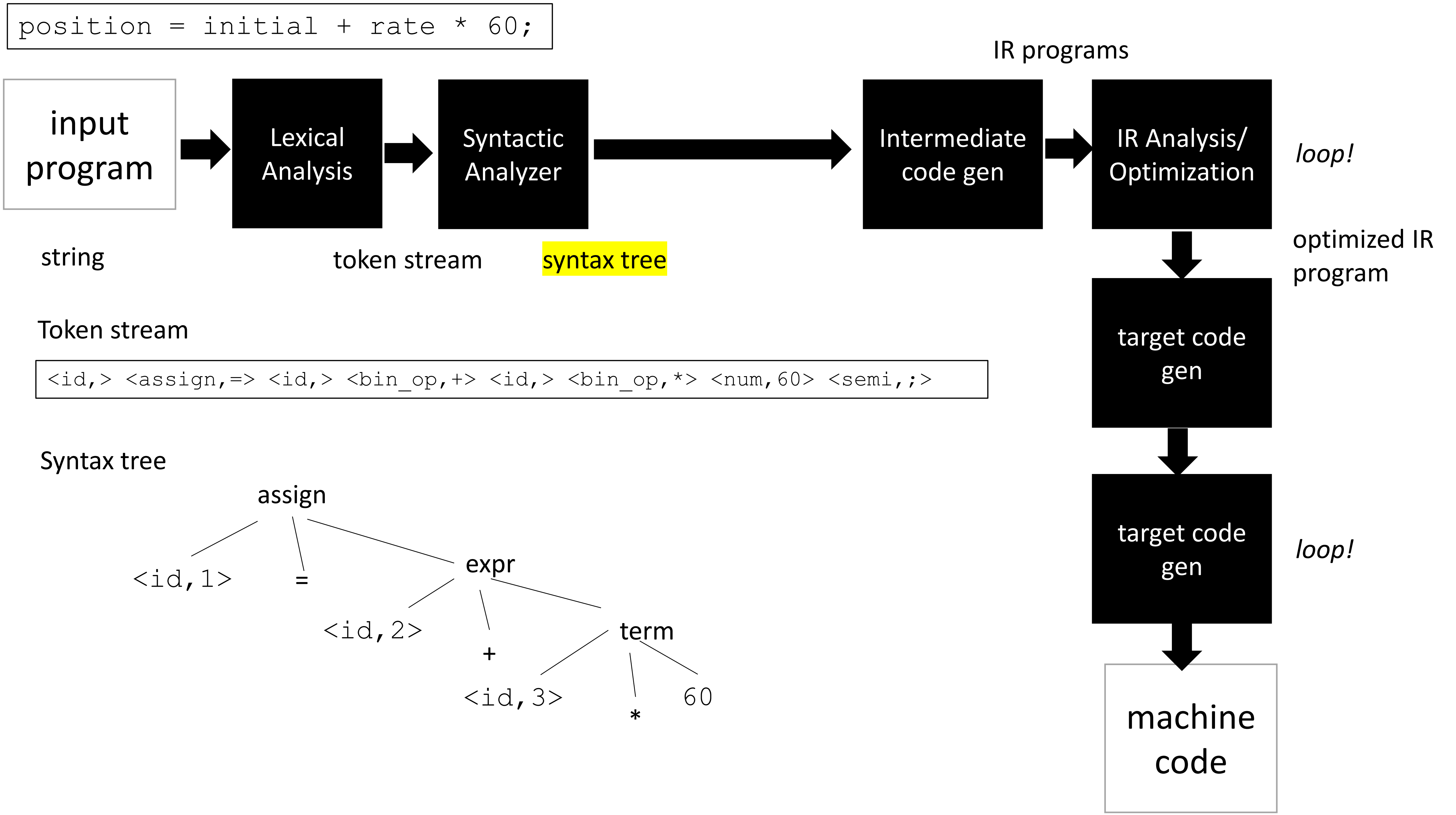
CFG

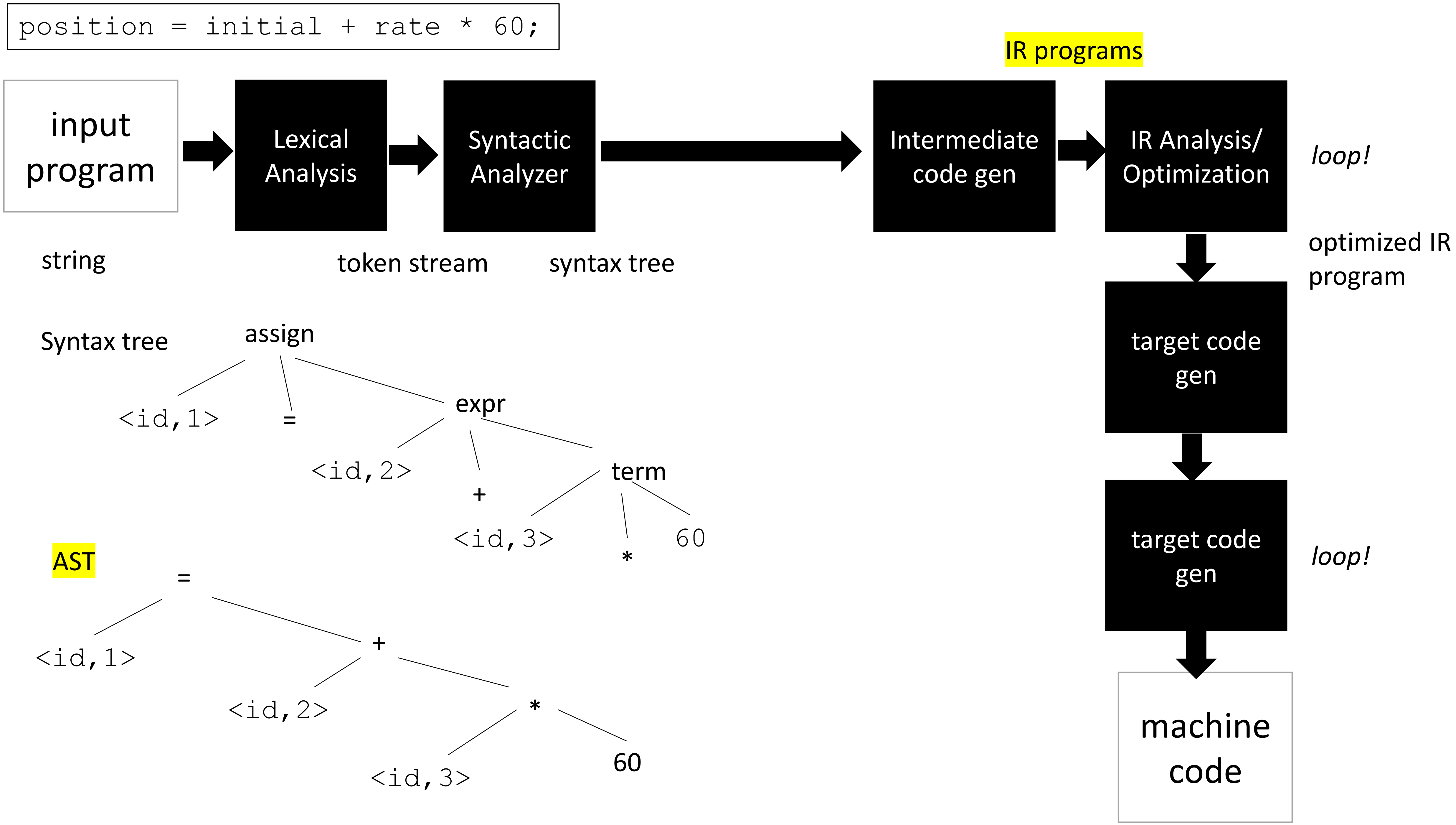


3 address code

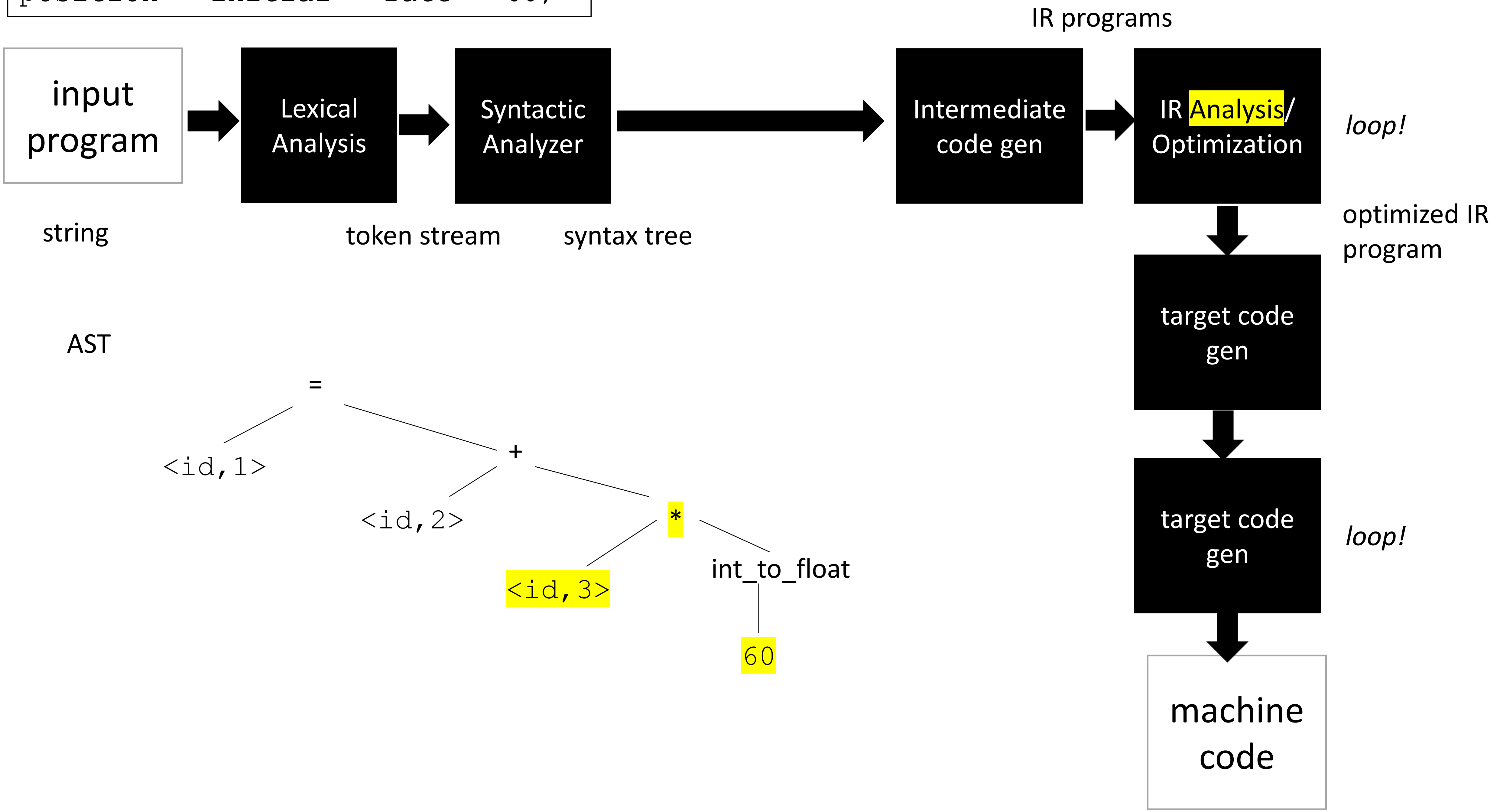
```
store i32 0, ptr %2
%3 = load i32, ptr %1
%4 = add nsw i32 %3, 1,
store i32 %4, ptr %1
%5 = load i32, ptr %2
```

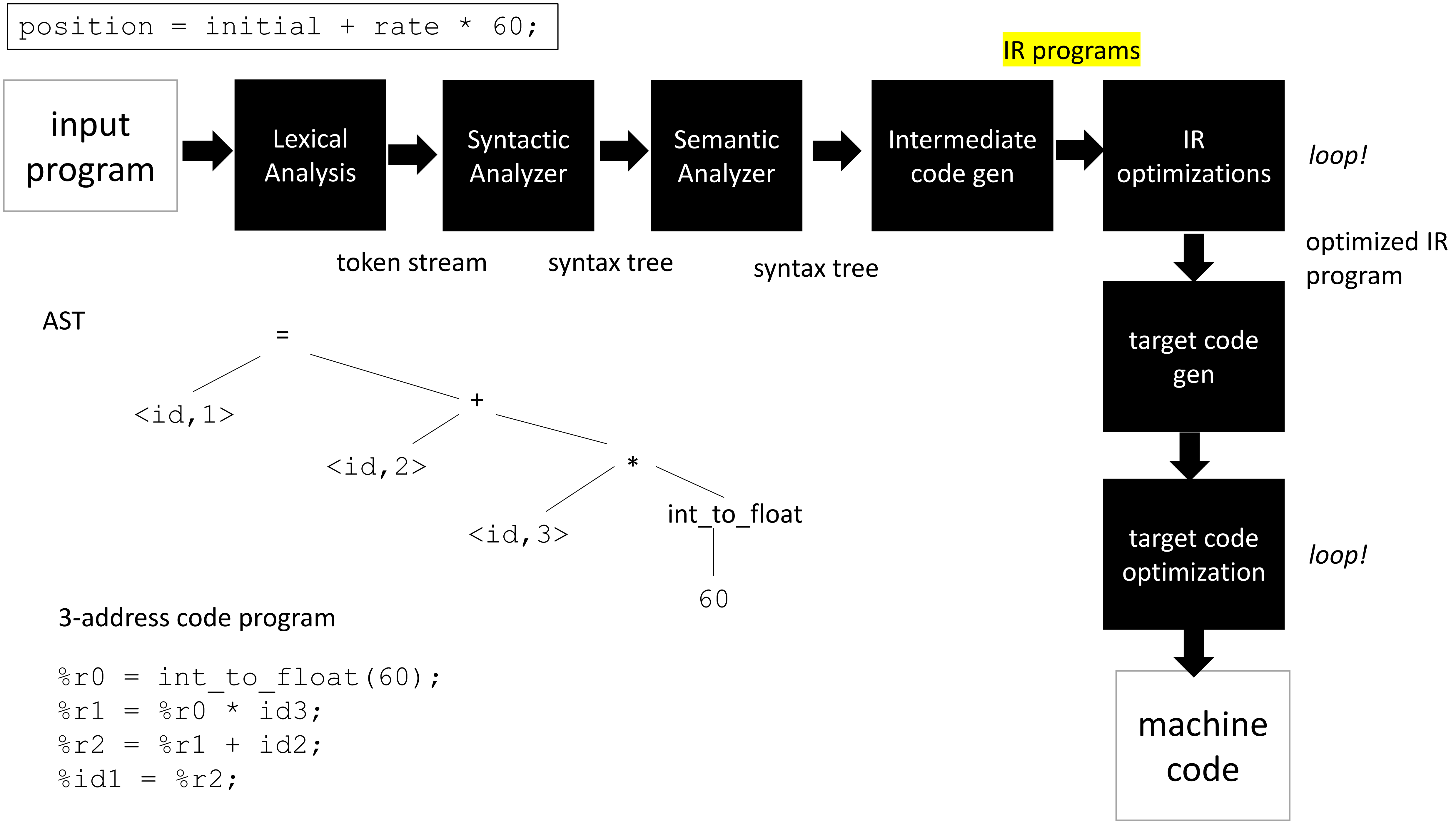

IR programs
input program
Lexical Analysis
Syntactic Analyzer
Intermediate code gen
IR Analysis/ Optimization
loop!
string
token stream
syntax tree
optimized IR program
position = initial + rate * 60;
Token stream
<id,> <assign,=> <id,> <bin_op,+> <id,> <bin_op,*> <num,60> <semi,;>
target code gen
target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


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

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




position = initial + rate * 60;
IR programs
input program
Lexical Analysis
Syntactic Analyzer
Semantic Analyzer
Intermediate code gen
IR optimizations
loop!
token stream
syntax tree
syntax tree
optimized IR program
target code gen
target code optimization
loop!
machine code
AST
=
<id,1>
+
<id,2>
*
<id,3>
int_to_float
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

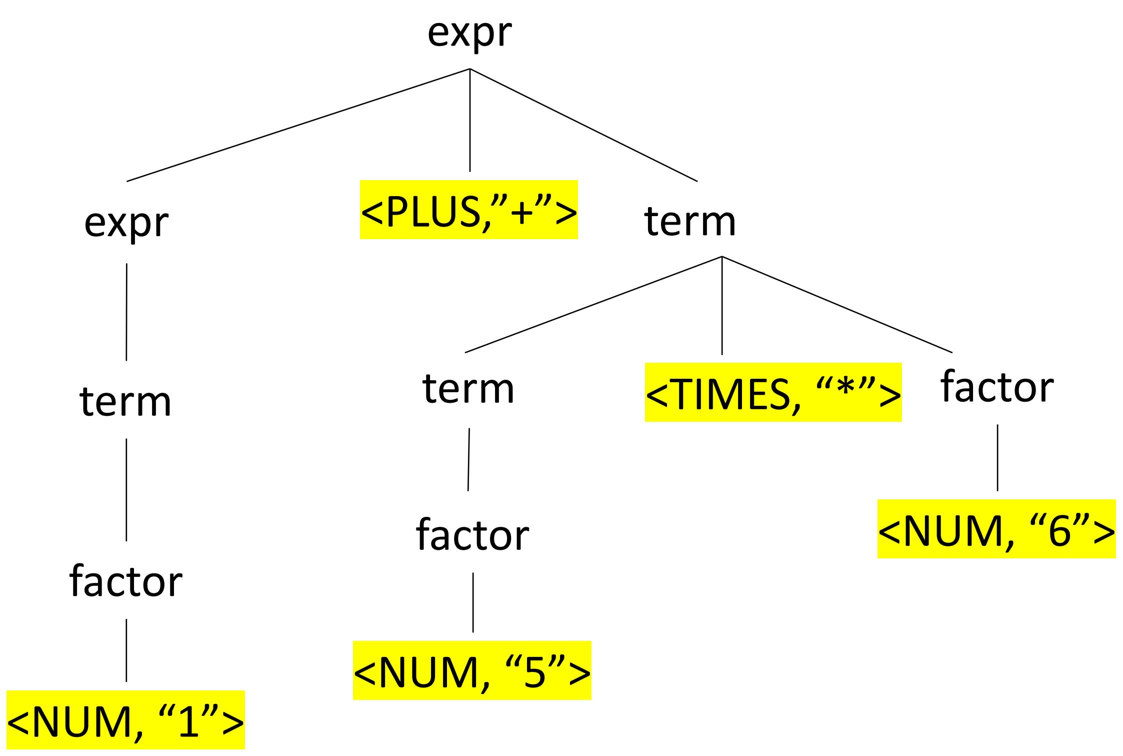
Abstract Syntax Trees

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`

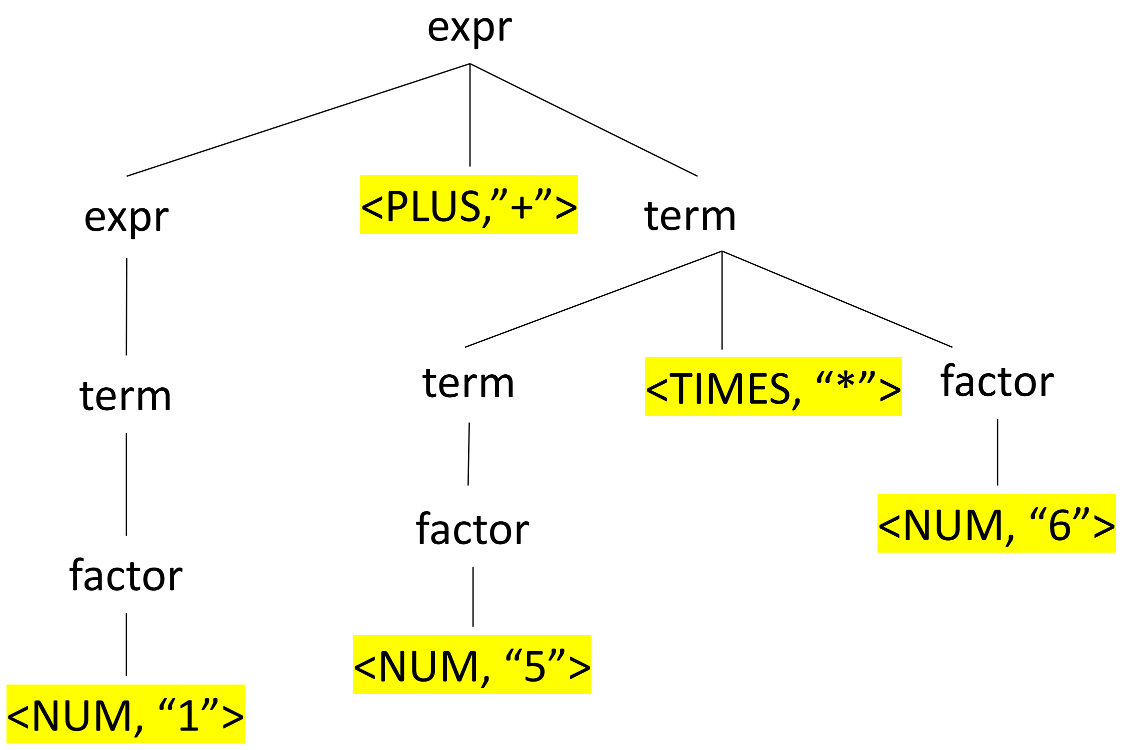


What is an AST?

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`



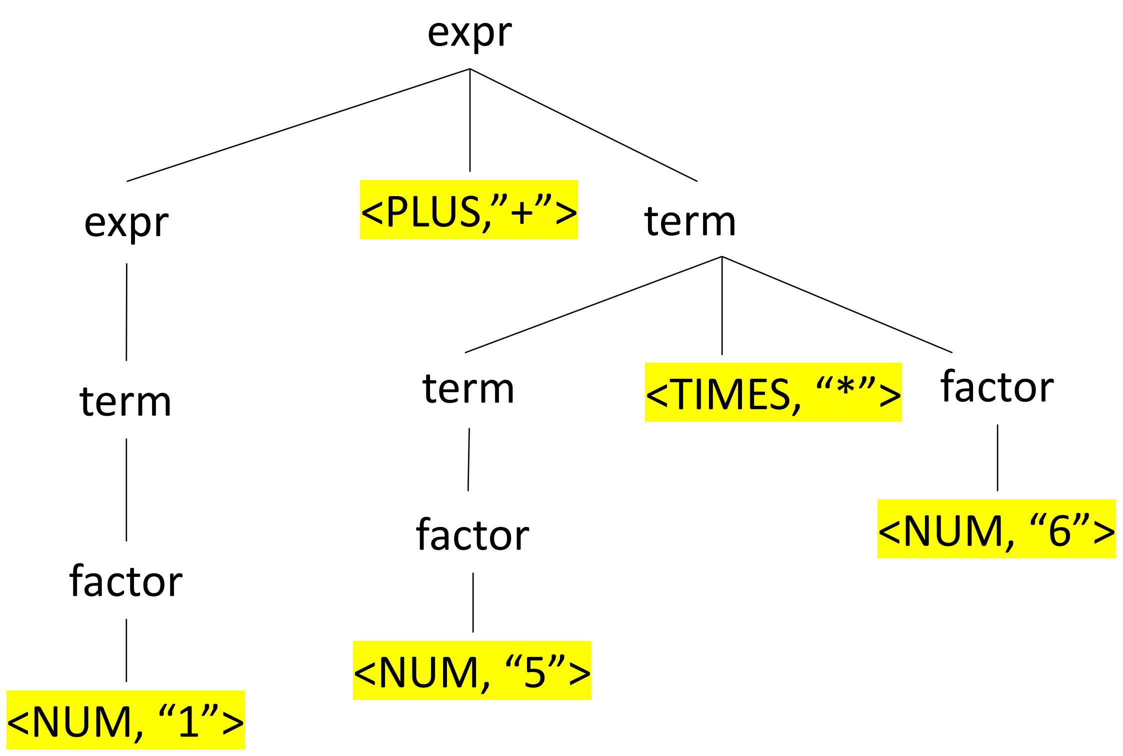
What are leaves?

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



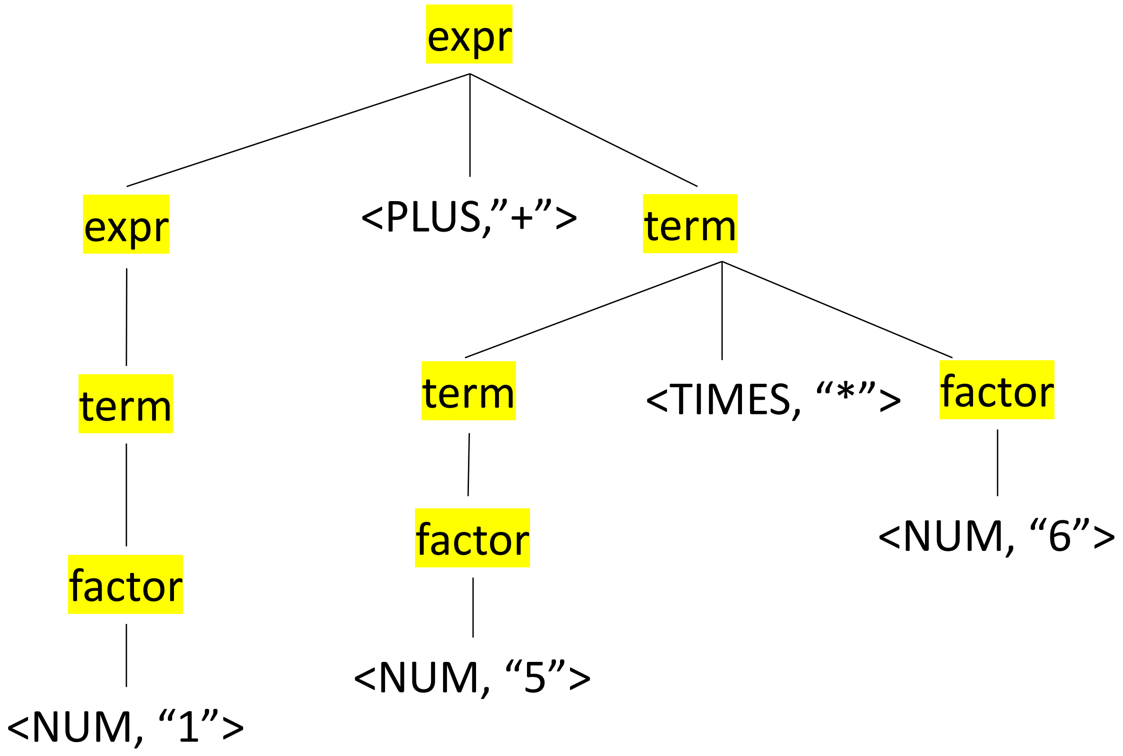
What are leaves? lexemes

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>



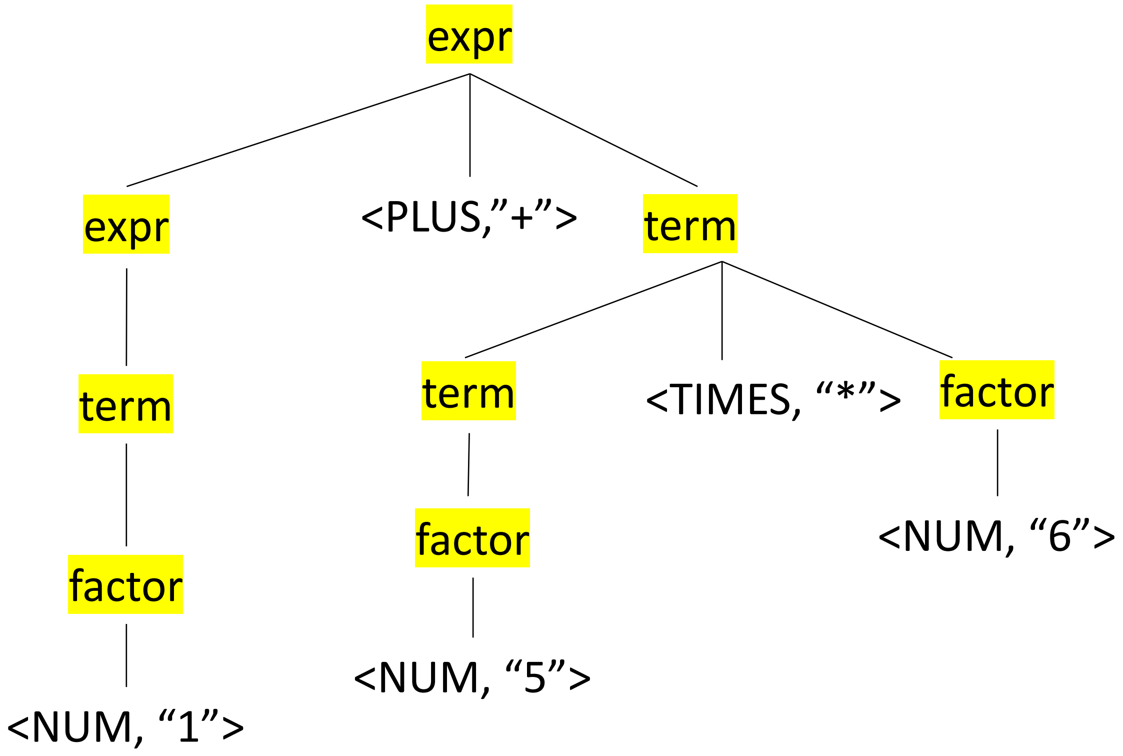
What are nodes?

What is an AST?

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



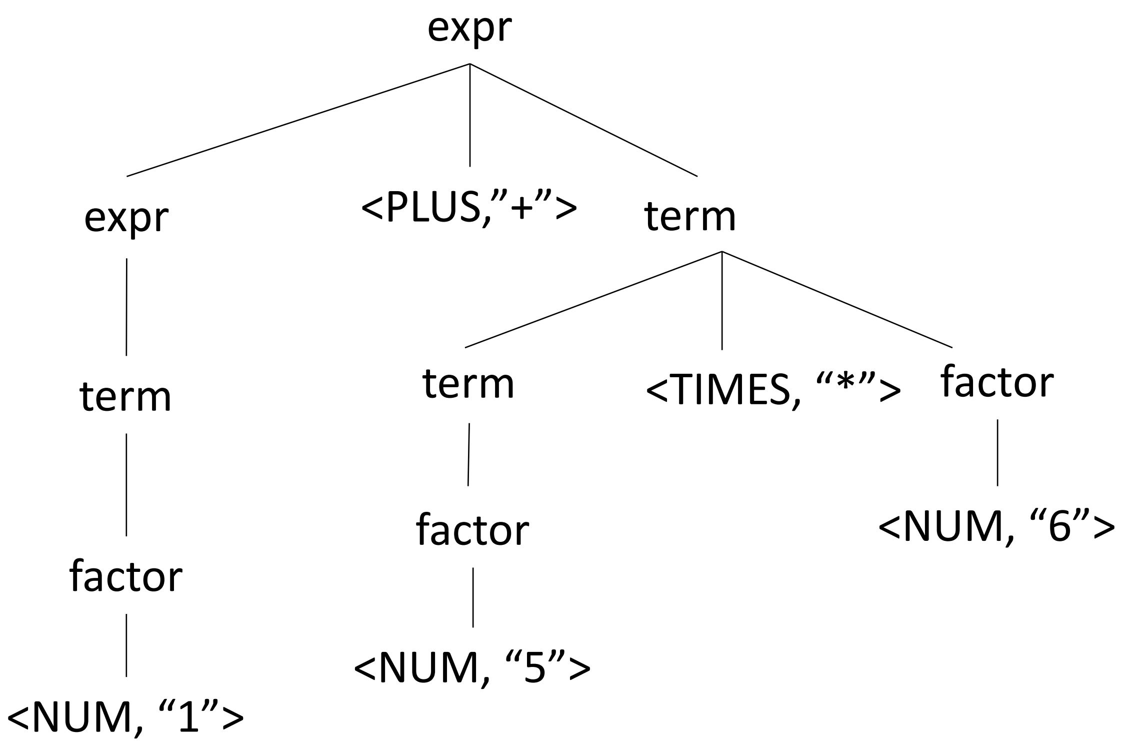
What are nodes? non-terminals

What is an AST?

`input: 1+5*6`

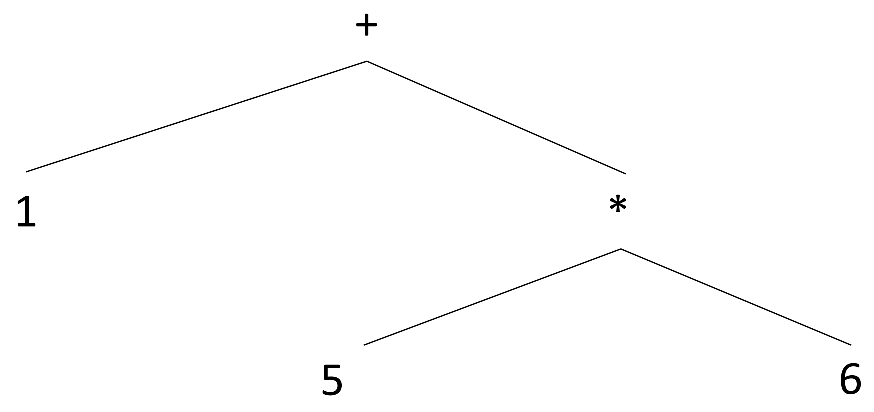
Parse trees are defined by the grammar
- **Tokens**
- **Production rules**

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



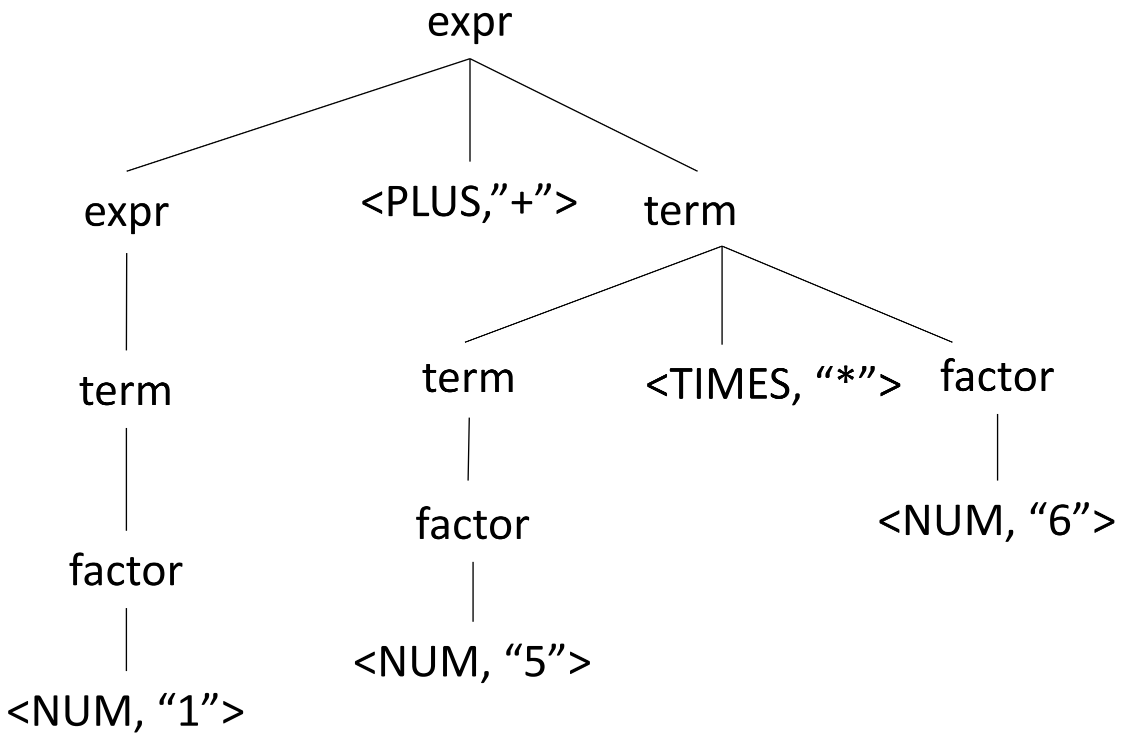
What is an AST?

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



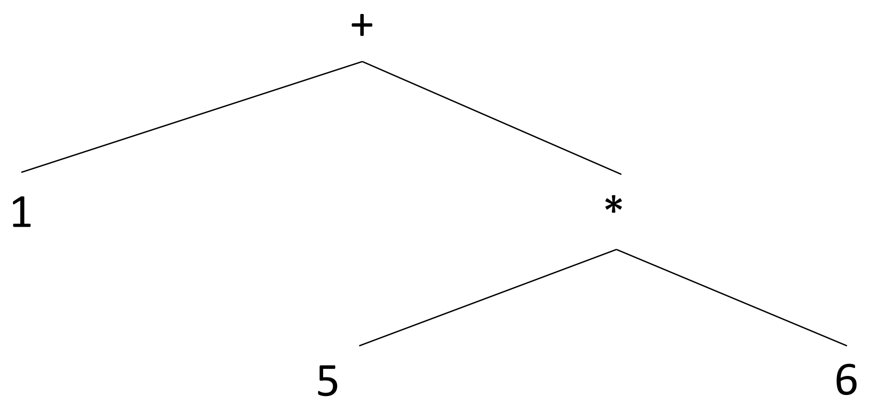
AST

What are some differences?

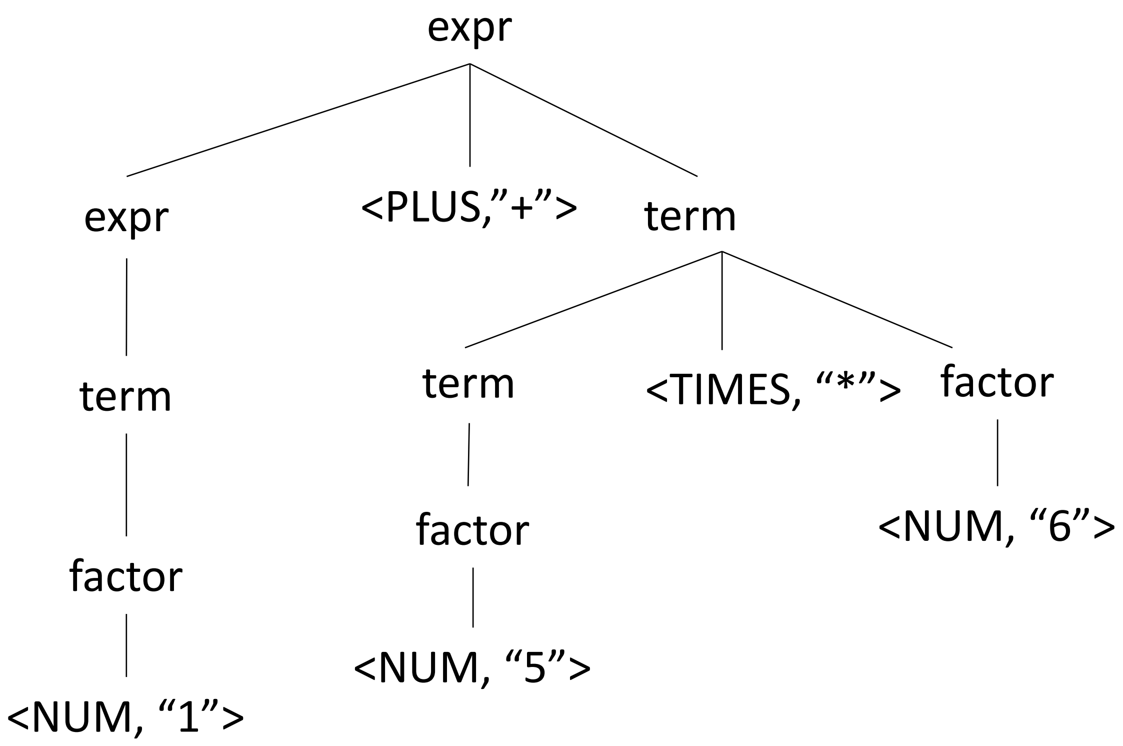


What is an AST?

`input: 1+5*6`



AST



What are some differences?

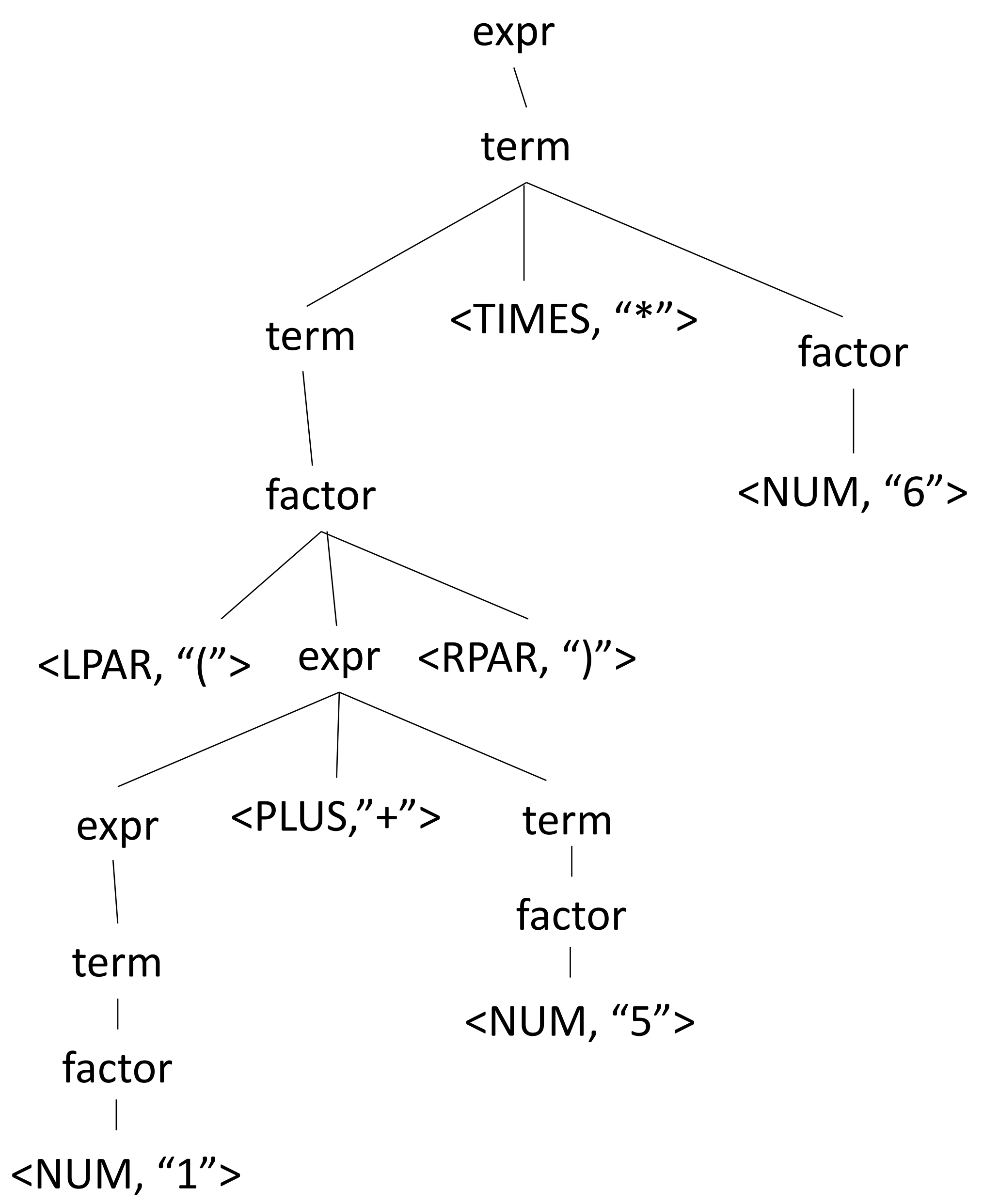
- disjoint from the grammar
- leaves are data, not lexemes
- nodes are operators, not non-terminals

Example

what happens to ()s in an AST?

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

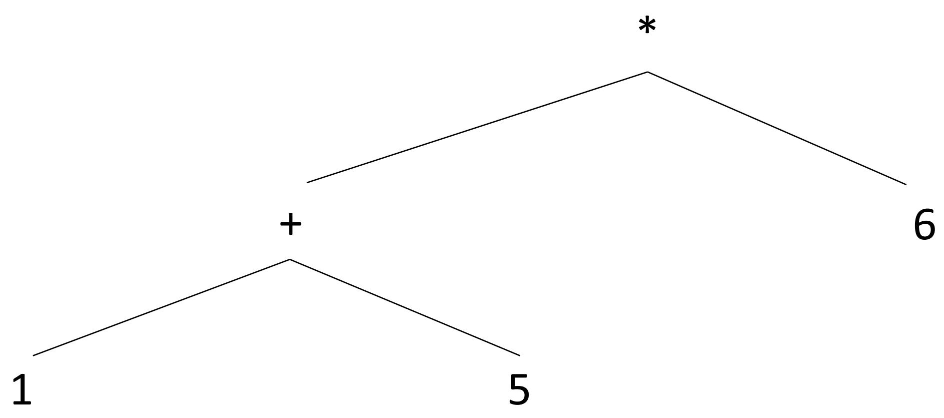
`input: (1+5)*6`

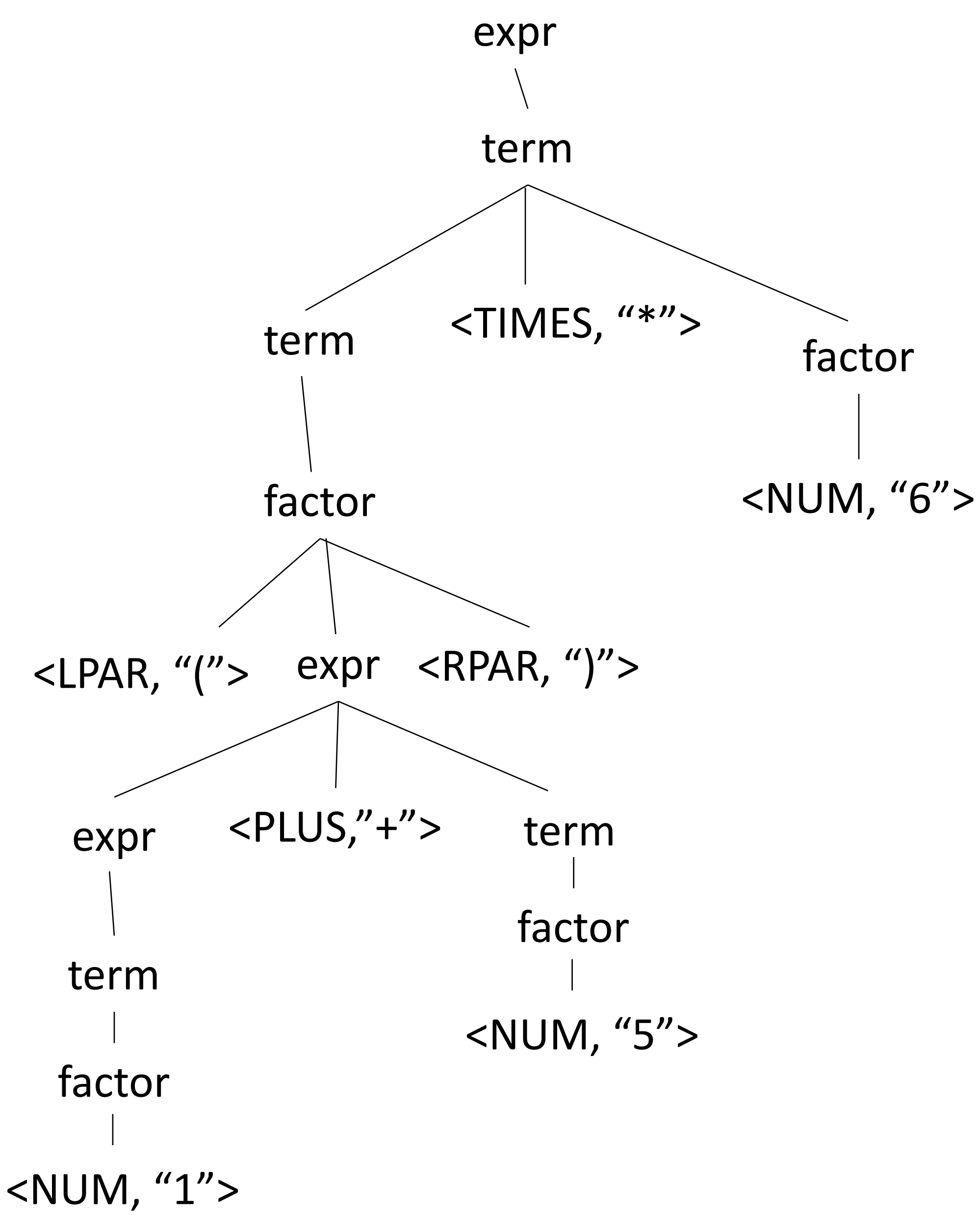


Example

`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

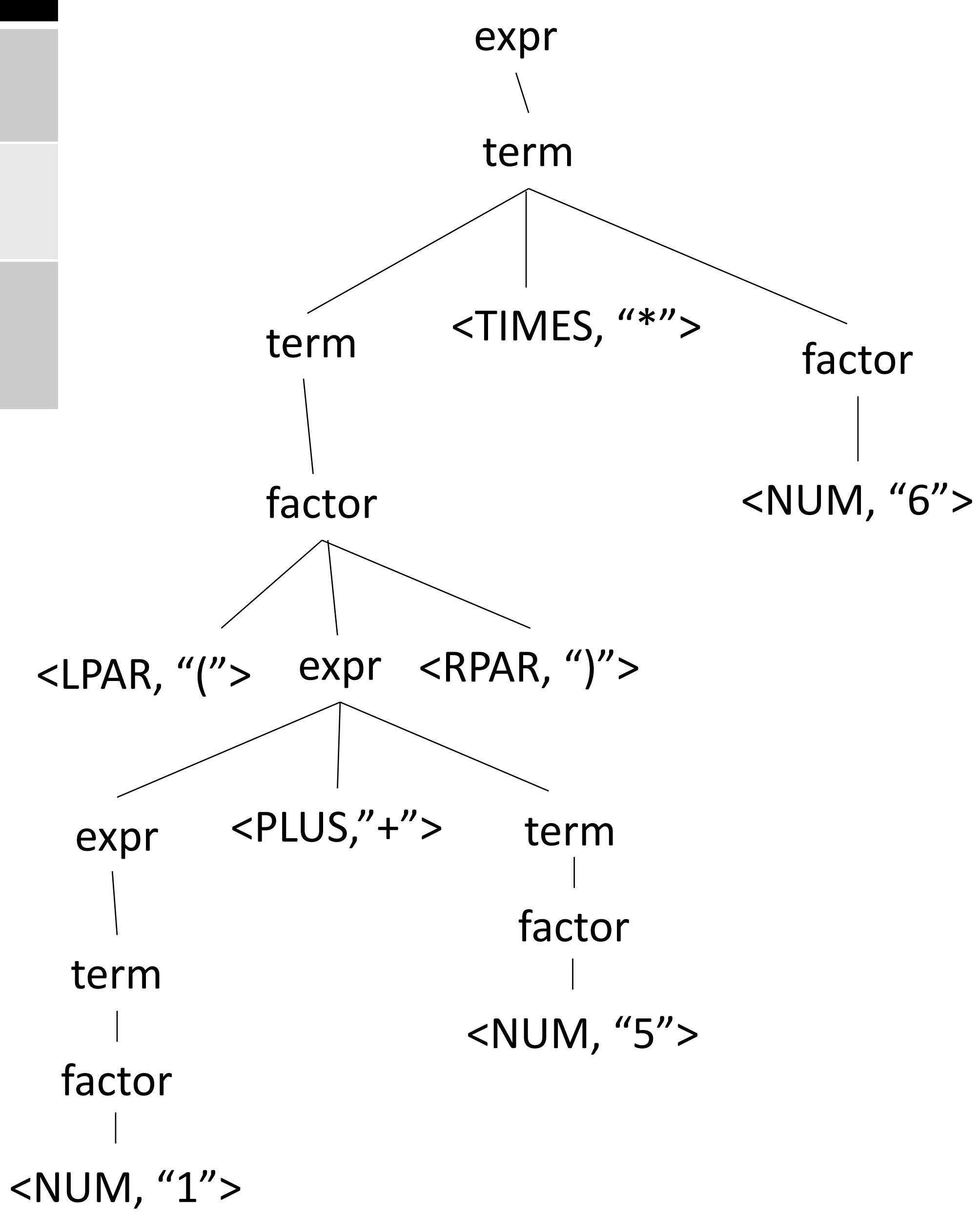
- A tree based data structure, used to represent expressions

- 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 (`+`, `*`)

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`

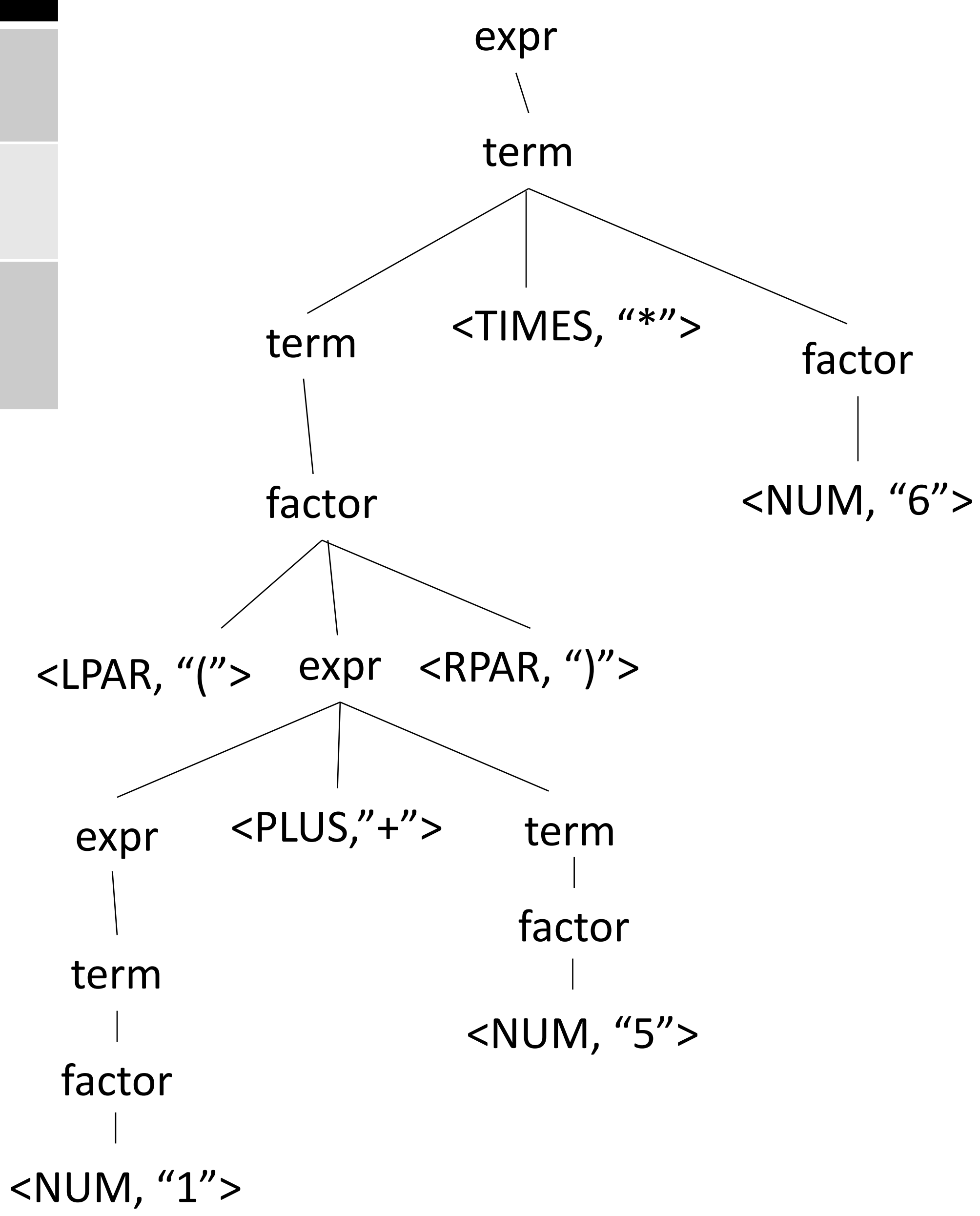


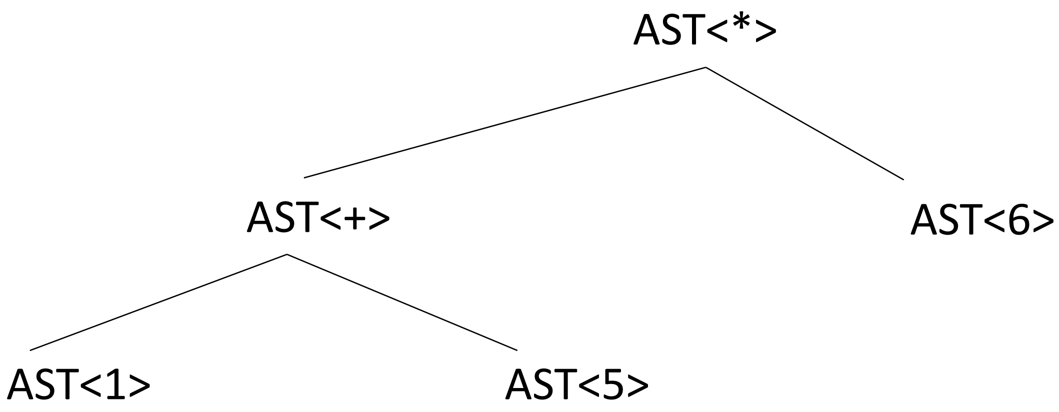
Lets build the AST

AST<?>

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

`input: (1+5)*6`

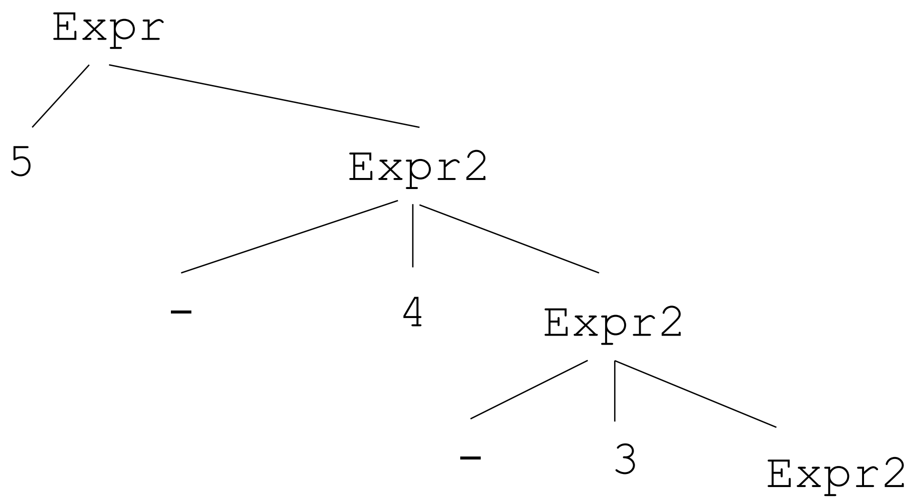




Creating an AST from top down grammar

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

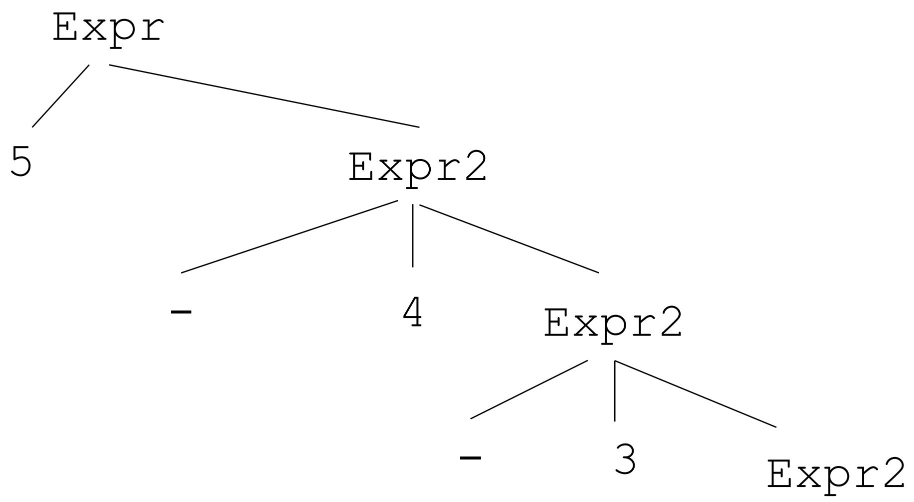
5 - 4 - 3

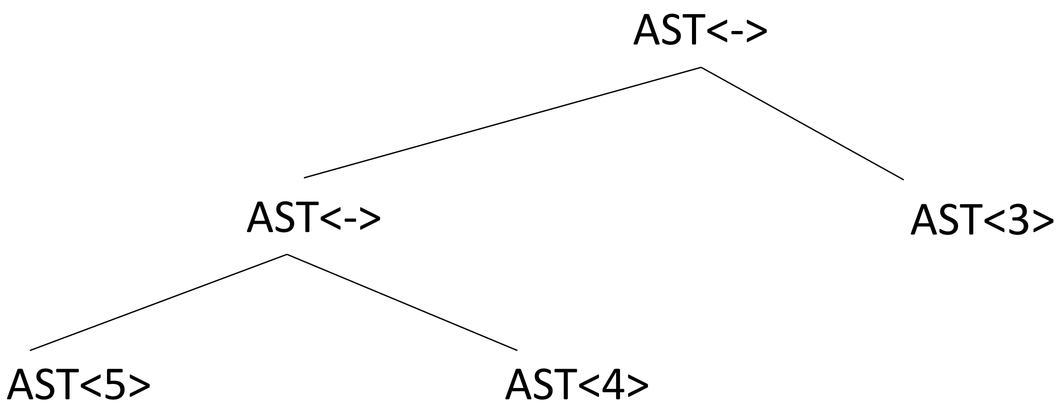


Creating an AST from top down grammar

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

`5 - 4 - 3`





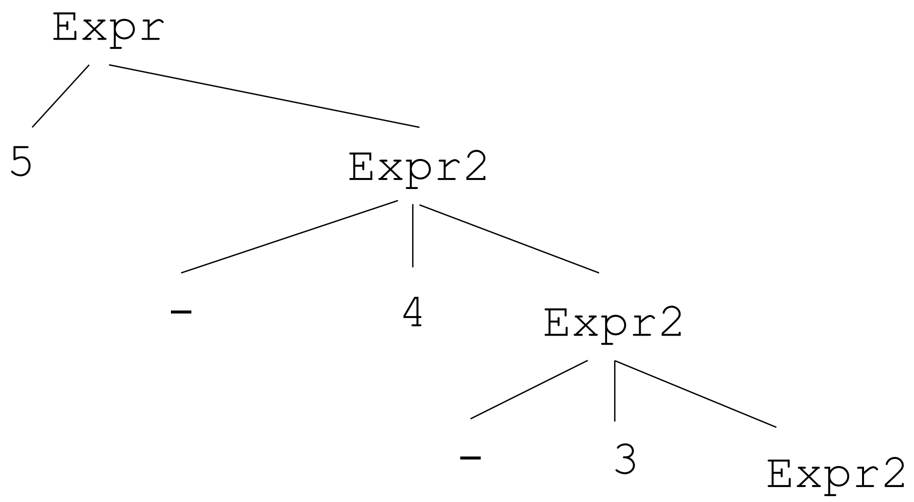
How do we get to the desired parse tree?

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.

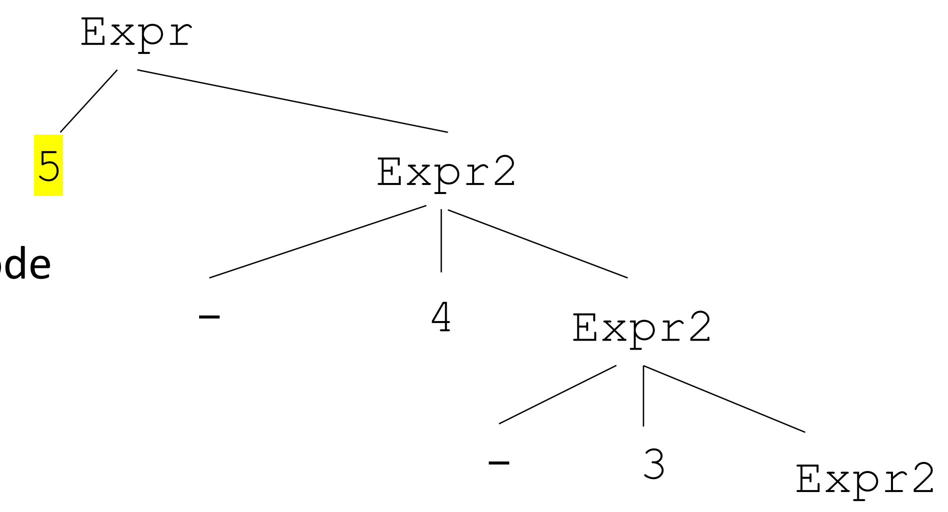
`5 - 4 - 3`



Creating an AST from top down grammar

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

5 - 4 - 3



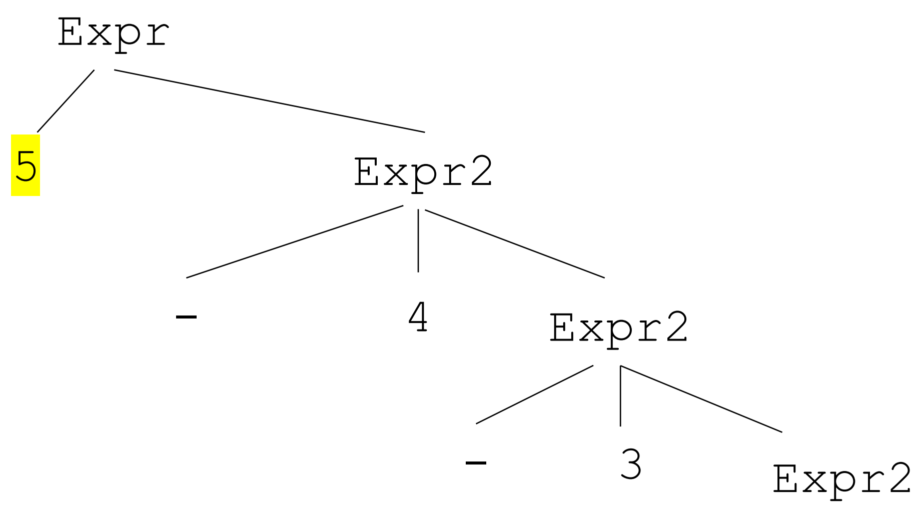
Get number node after we see a number

AST<5>

Creating an AST from top down grammar

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

5 - 4 - 3



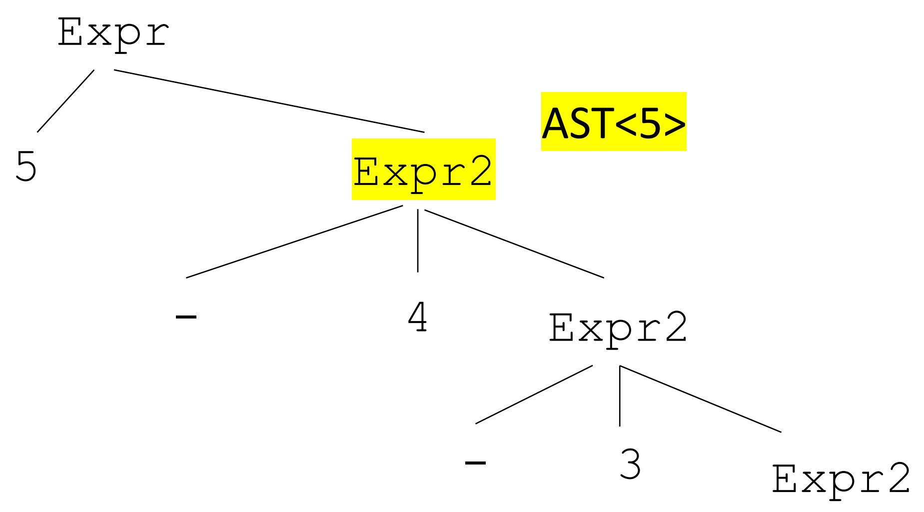
Pass the node down

AST<5>

Creating an AST from top down grammar

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

5 - 4 - 3



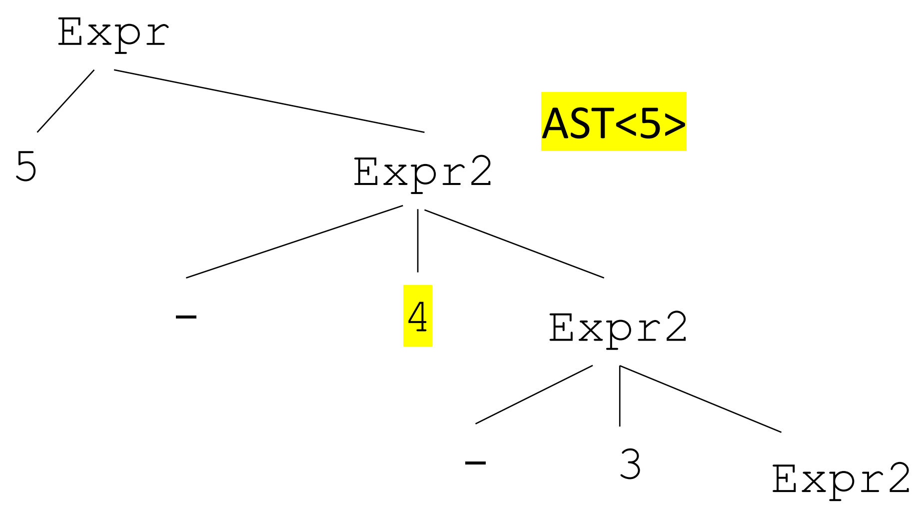
Pass the node down

AST<5>

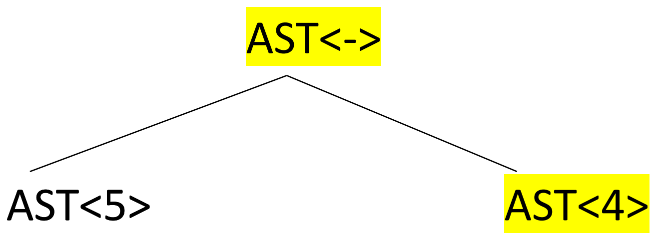
Creating an AST from top down grammar

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

5 - 4 - 3



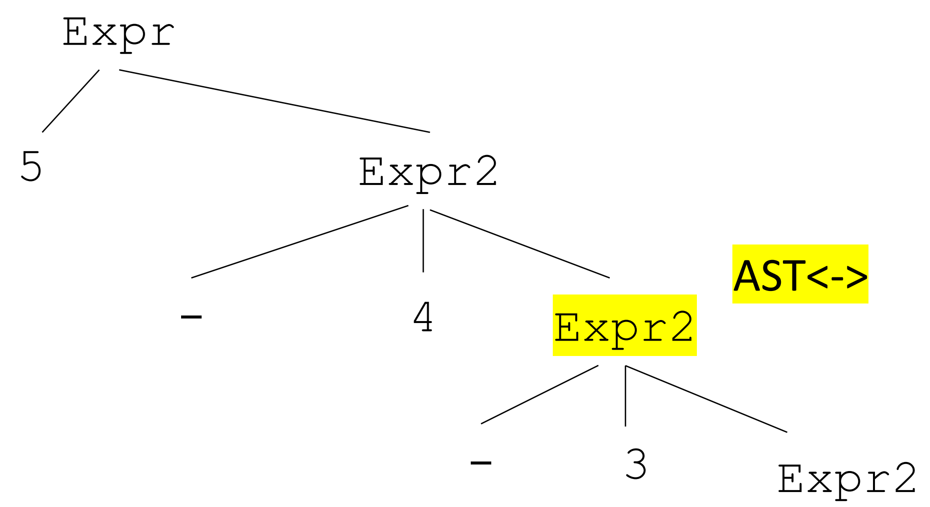
In Expr2, after 4 is parsed, create a number node and a minus node



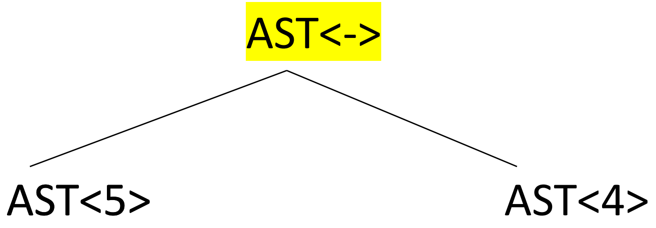
Creating an AST from top down grammar

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

5 - 4 - 3



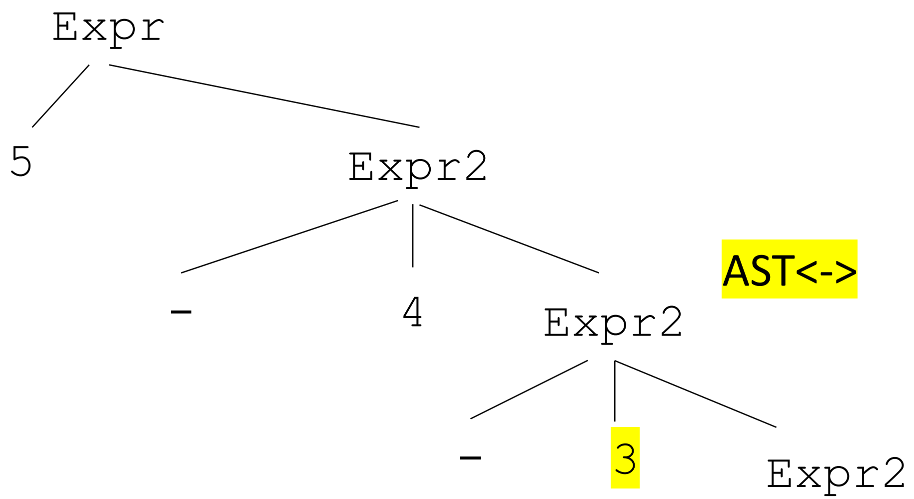
pass the new node down

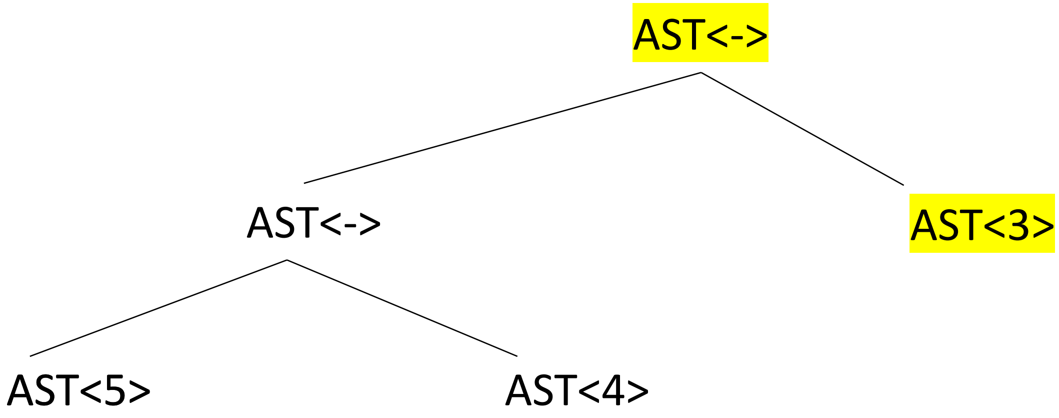


Creating an AST from top down grammar

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

5 - 4 - 3



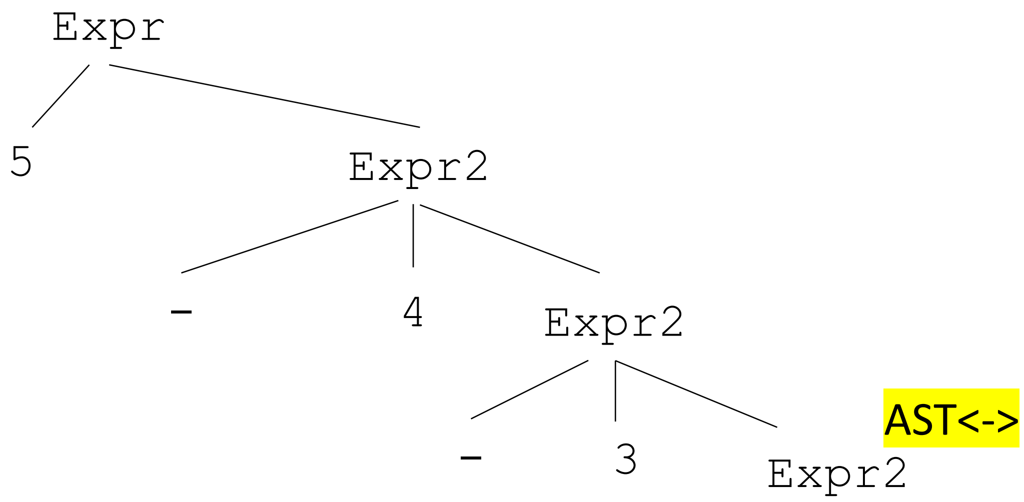


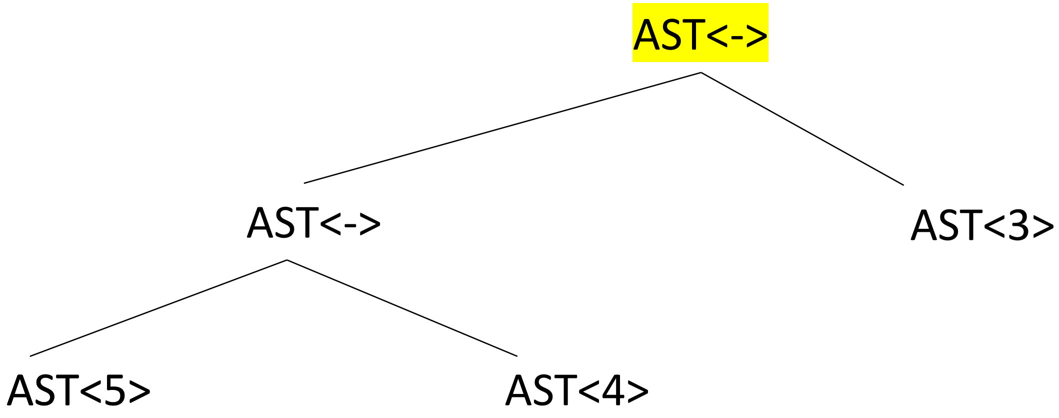
In Expr2, after 3 is parsed, create a number node and a minus node

Creating an AST from top down grammar

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

`5 - 4 - 3`



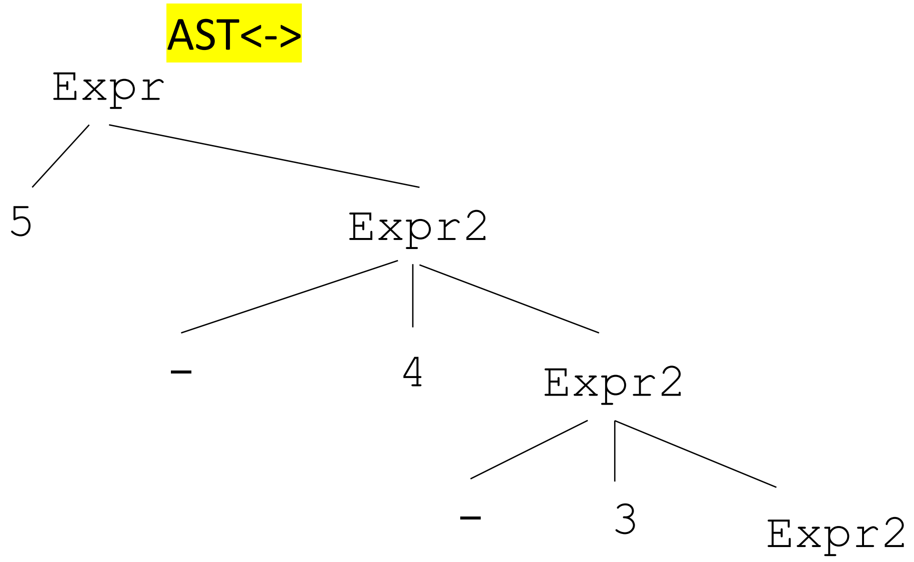


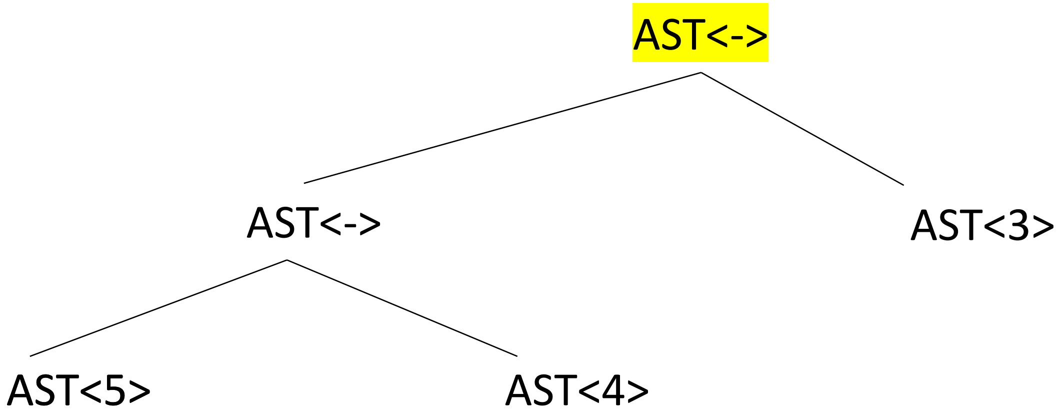
pass down the new node

Creating an AST from top down grammar

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

5 - 4 - 3





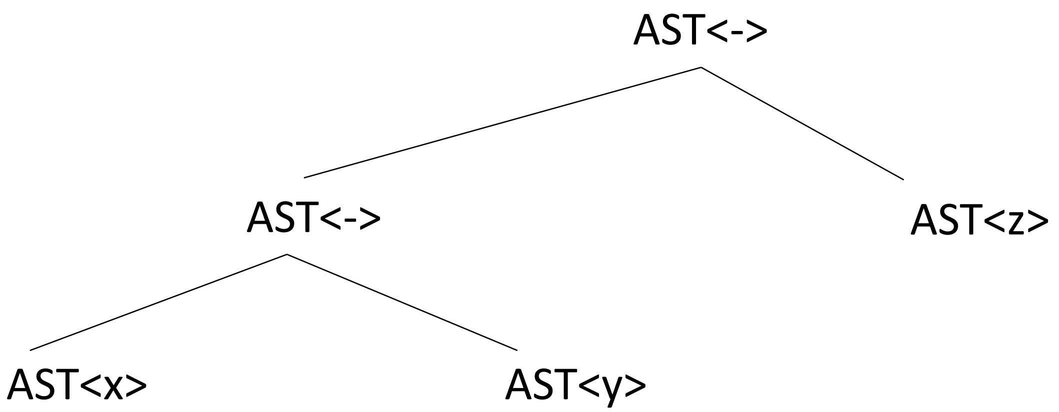
return the node when there is nothing left to parse

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`

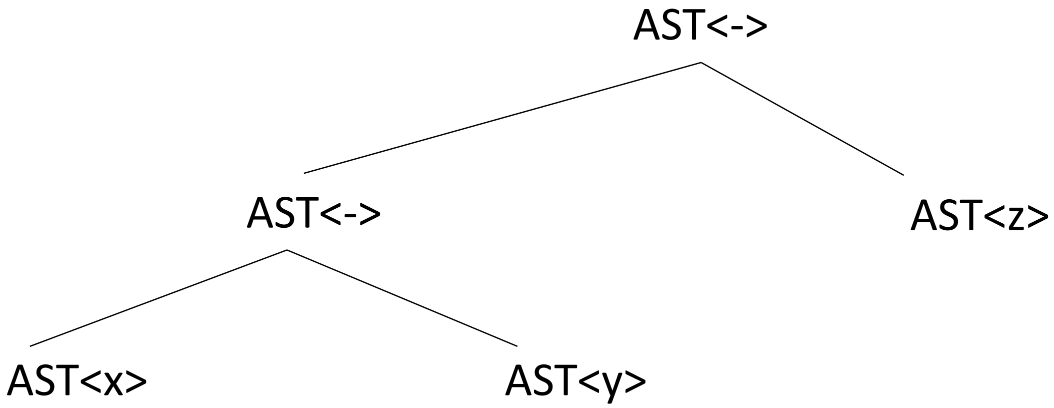


Evaluate an AST by doing a post order traversal

```
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
```



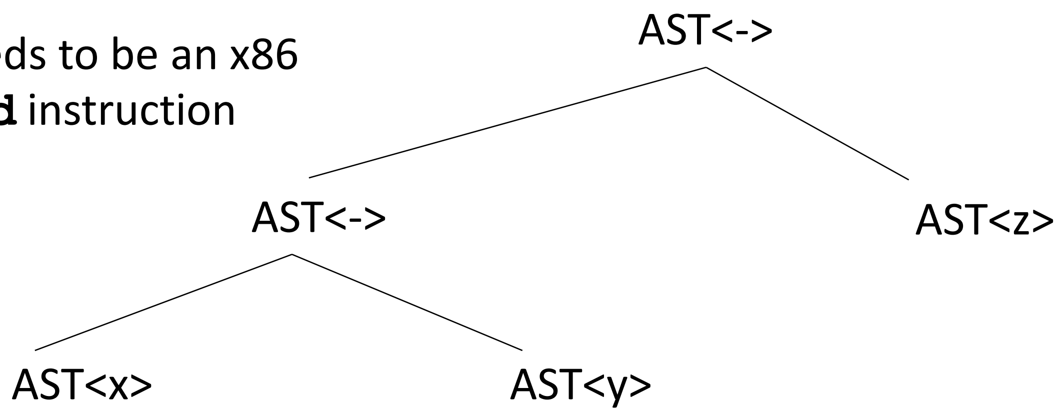
How does this change things?

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?

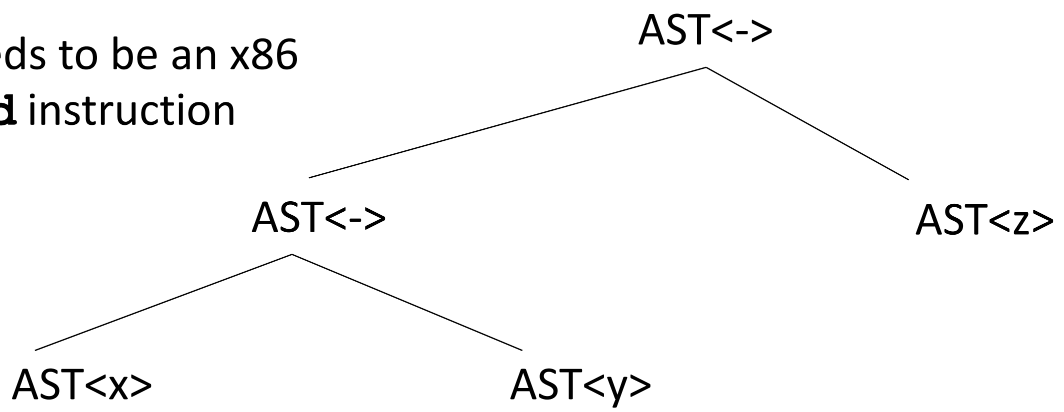
Evaluate an AST by doing a post order traversal

```
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?

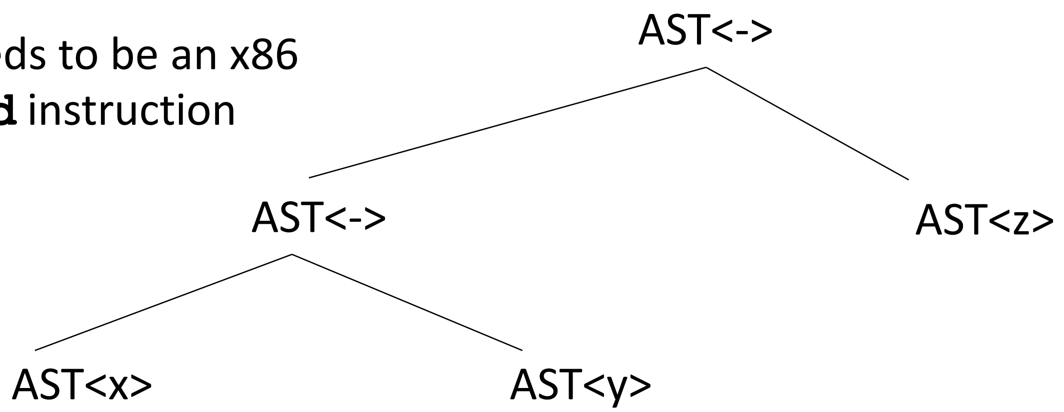
Evaluate an AST by doing a post order traversal

```
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

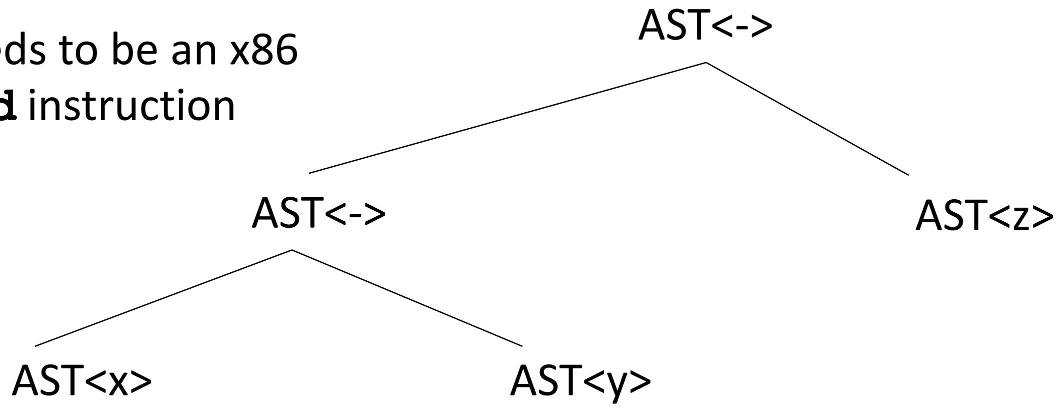
Evaluate an AST by doing a post order traversal

```
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?

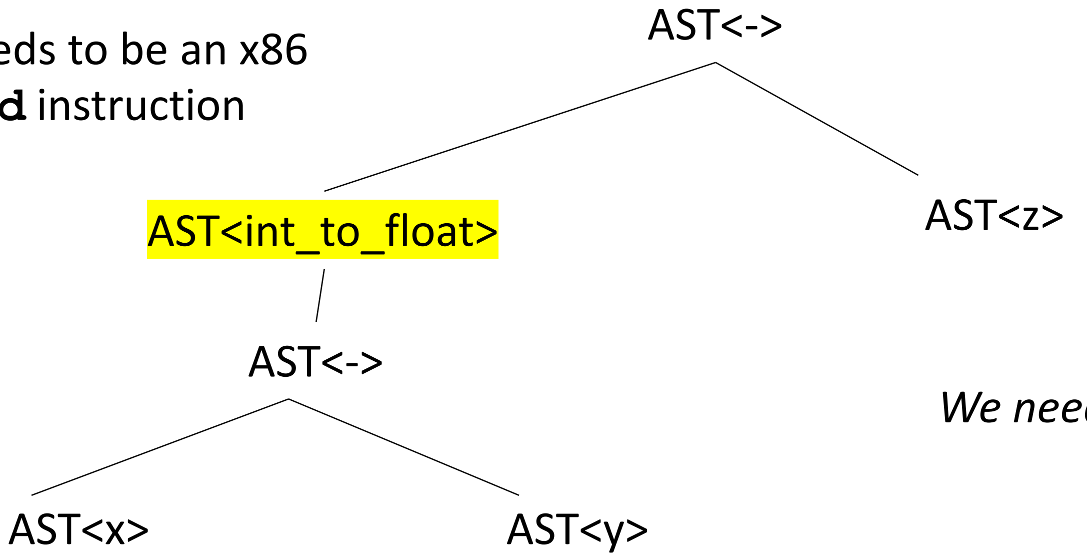
Evaluate an AST by doing a post order traversal

```
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

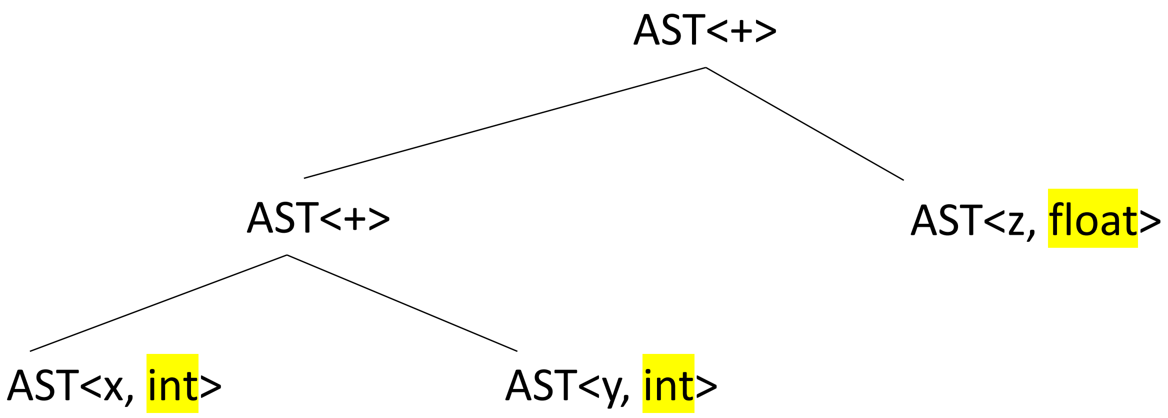
- Check a program to ensure that it adheres to the type system

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

Type checking on an AST

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

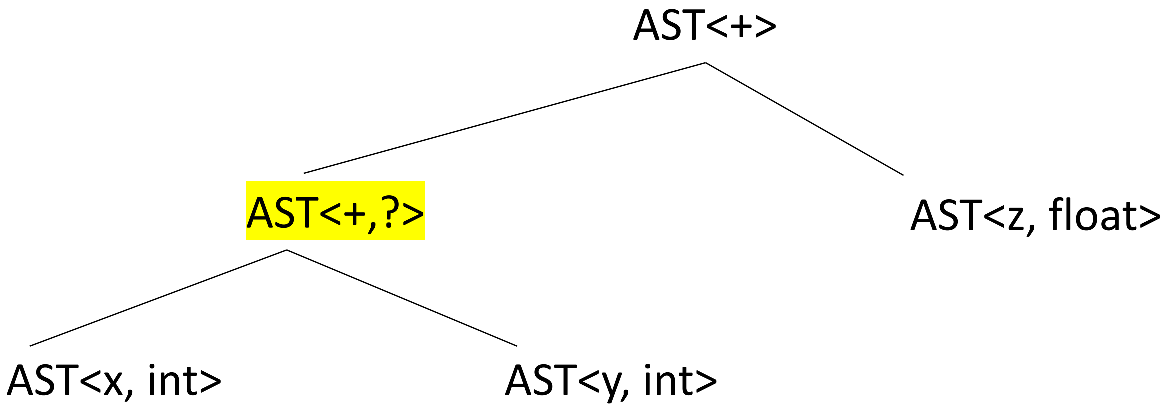
each node additionally gets a type
we can get this from the symbol table for the leaves or based on the input (e.g. 5 vs 5.0)



Type checking on an AST

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

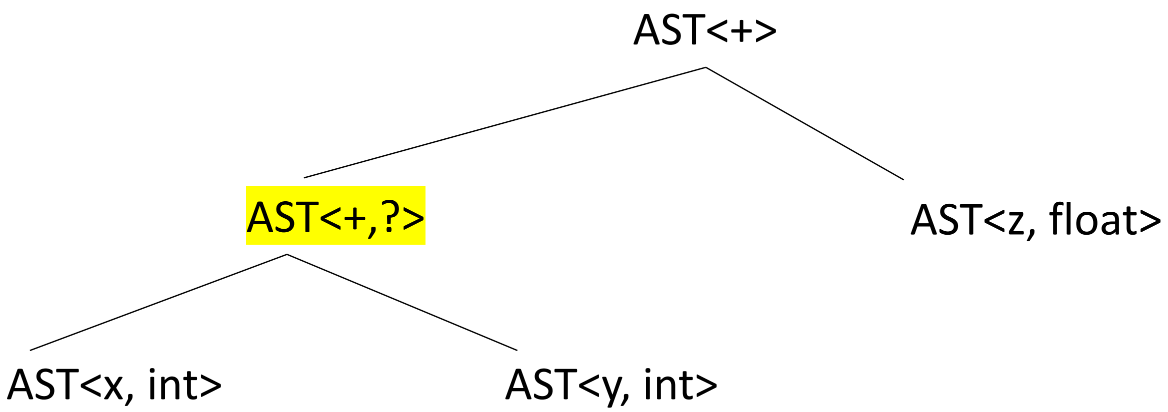
How do we get the type for this one?



Type checking on an AST

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

How do we get the type for this one?



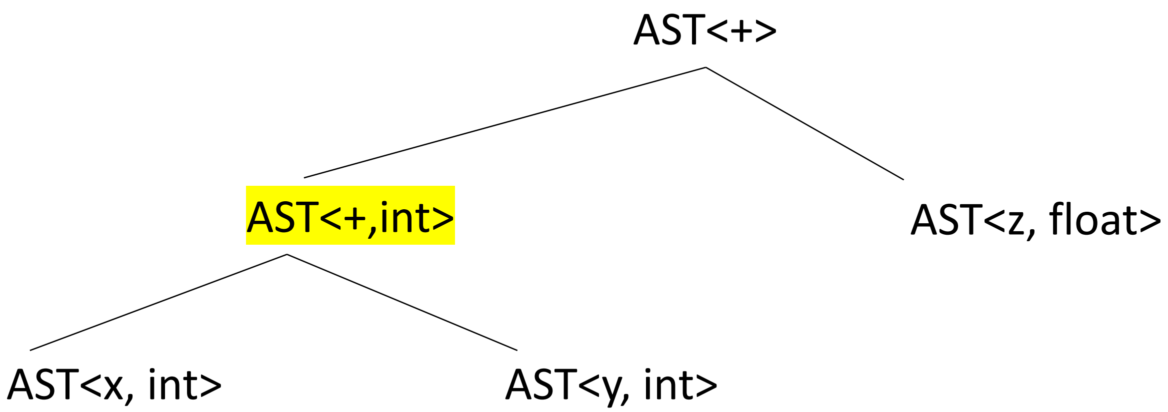
inference rules for addition:

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

Type checking on an AST

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

How do we get the type for this one?



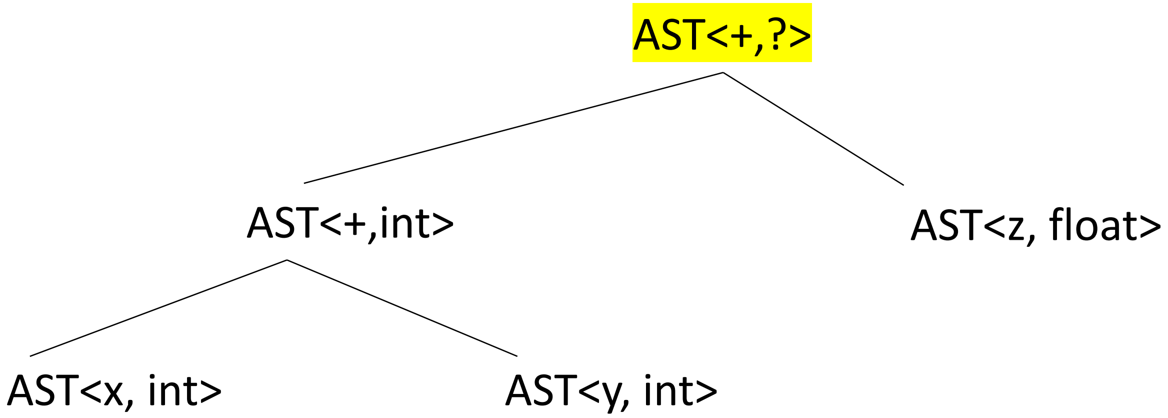
inference rules for addition:

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

Type checking on an AST

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

How do we get the type for this one?



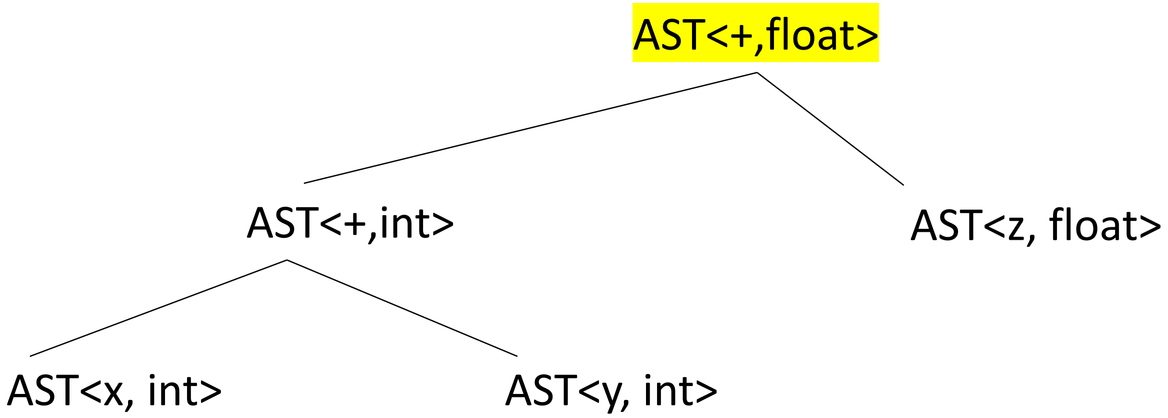
inference rules for addition:

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

Type checking on an AST

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

How do we get the type for this one?



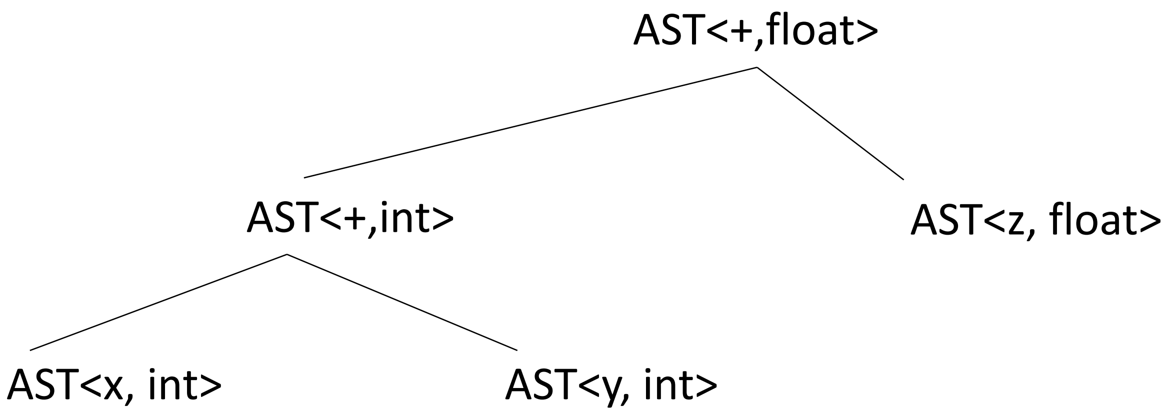
inference rules for addition:

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

Type checking on an AST

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

How do we get the type for this one?



inference rules for addition:

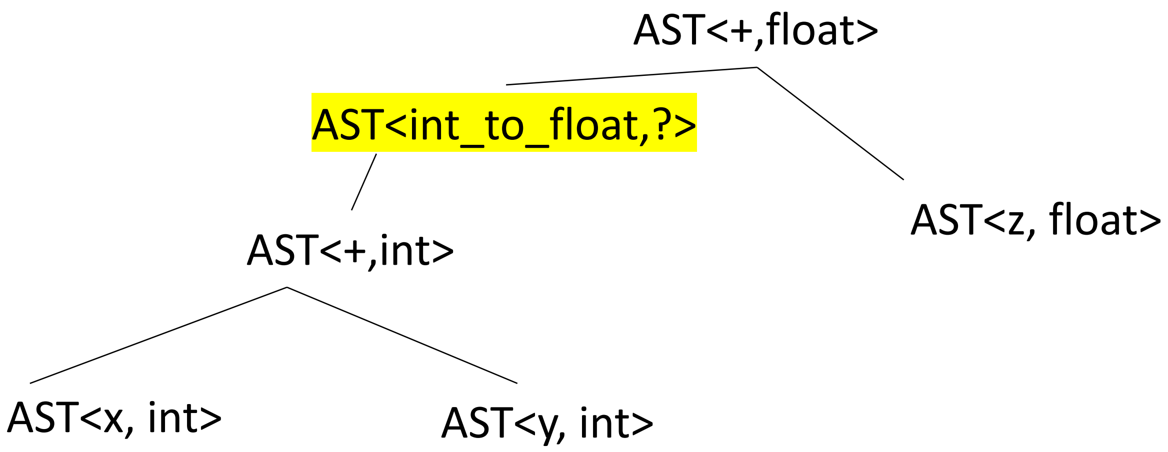
first	second	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

what else?

Type checking on an AST

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

How do we get the type for this one?



inference rules for addition:

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

what else? need to convert the int to a float

Intermediate Representations
RULES FOR OUR IR

Class-IR: The Players

Inputs/outputs (IO): 32-bit typed inputs
e.g.: `int x, int y, float z    // e.g. params to a function`

Program Variables (Variables): 32-bit untyped virtual register
given as `vrX` where `X` is an integer:
e.g. `vr0, vr1, vr2, vr3` …

Constants (float or ints): e.g. 3.5, 3e5, 6, 1024

we will assume input/output names are disjoint from virtual register names

Class-IR

binary operators:

```
dst = operation(op0, op1);
```

```
operations can be one of these:
[add, sub, mult, div, eq, lt]
```

each operation is followed by an “i” or “f”, which specifies how the bits in the registers are interpreted, eg **multi** for integers, **multf** for floating point.

Class-IR

binary operators:

```
dst = operation(op0, op1);
```

```
operations can be one of:
[add, sub, mult, div, eq, lt]
```

all of dst, op0, and op1 must be untyped virtual registers.

Class-IR: Examples

binary operators:

```
dst = operation(op0, op1);

Examples:

vr0 = addi(vr1, vr2);
vr3 = subf(vr4, vr5);

x  = multf(vr0, vr1); not allowed!
vr0 = addi(vr1, 1);  not allowed!
```

We'll talk about how to do this using other instructions

Class-IR: Control Flow

Control flow

`branch(label);`

- branches unconditionally to the label

`bne(op0, op1, label)`

- if op0 is not equal to op1 then branch to label
- operands must be virtual registers!

`beq(op0, op1, label)`

- Same as bne except it is for equal

Class-IR

Assignment

```
vr0 = vr1
```

```
one virtual register can be assigned to another
```

Class-IR

Assignment

`vr0 = vr1`

one virtual register can be assigned to another

Examples:

```
vr0 = 1; not allowed
vr1 = x; not allowed
```

Class-IR

```
unary get untyped register
dst = operation(op0);

operations are: [int2vr, float2vr]

Example:
```

Given IO: int x

```
vr1 = int2vr(x);
vr2 = float2vr(2.0);
```

Class-IR

`unary get typed data for IO`

```
dst = operation(op0);
```

```
operations are: [vr2int, vr2float]
```

```
Example:
```

Given IO: int x and float y

```
x = vr2int(vr1);
y = vr2float(vr3);
```

Class-IR

unary conversion operators for VRs:

`dst = operation(op0);`

`operations can be one of:`

`[vr_int2float, vr_float2int]`

`converts` `the bits in a` **`virtual register`** `from` **`one type to another.`** ***`op0`*** *`and`* ***`dst`*** *`must be a`* ***`virtual register!`***

Class-IR: Examples

unary conversion operators:

```
dst = operation(op0);

Examples:

vr0 = vr_int2float(vr1);
vr2 = vr_float2int(1.0); not allowed!
```

Example

`adding the values 1 - 9 to an input/output variable: int x`

Example

adding the values 1 - 9 to an input/output variable: int x

```
  vr0 = int2vr(1);
  vr1 = int2vr(1);
  vr2 = int2vr(10);
loop_start:
  vr3 = lti(vr0, vr2);
  bne(vr3, vr1, end_label);
  vr4 = int2vr(x);
  vr5 = addi(vr4,vr0);
  x = vr2int(vr5);
  vr0 = addi(vr0, vr1);
  branch(loop_start);
end_label:
```

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x = 6;
  y = x;
}
```

new scope. Add x with a new name

HT1

```
x: (INT, VAR, "x_1")
```

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

IR Program structure

- A sequence of 3 address instructions

- Programs can be split into **Basic Blocks:**
  - A **sequence of 3 address instructions** such that: there is **a single entry, single exit**

- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

How might they appear in a high-level language?

How many basic blocks?

```
…
if (x) {
    …
}
else {
   …
}
…
```

Single Basic Block

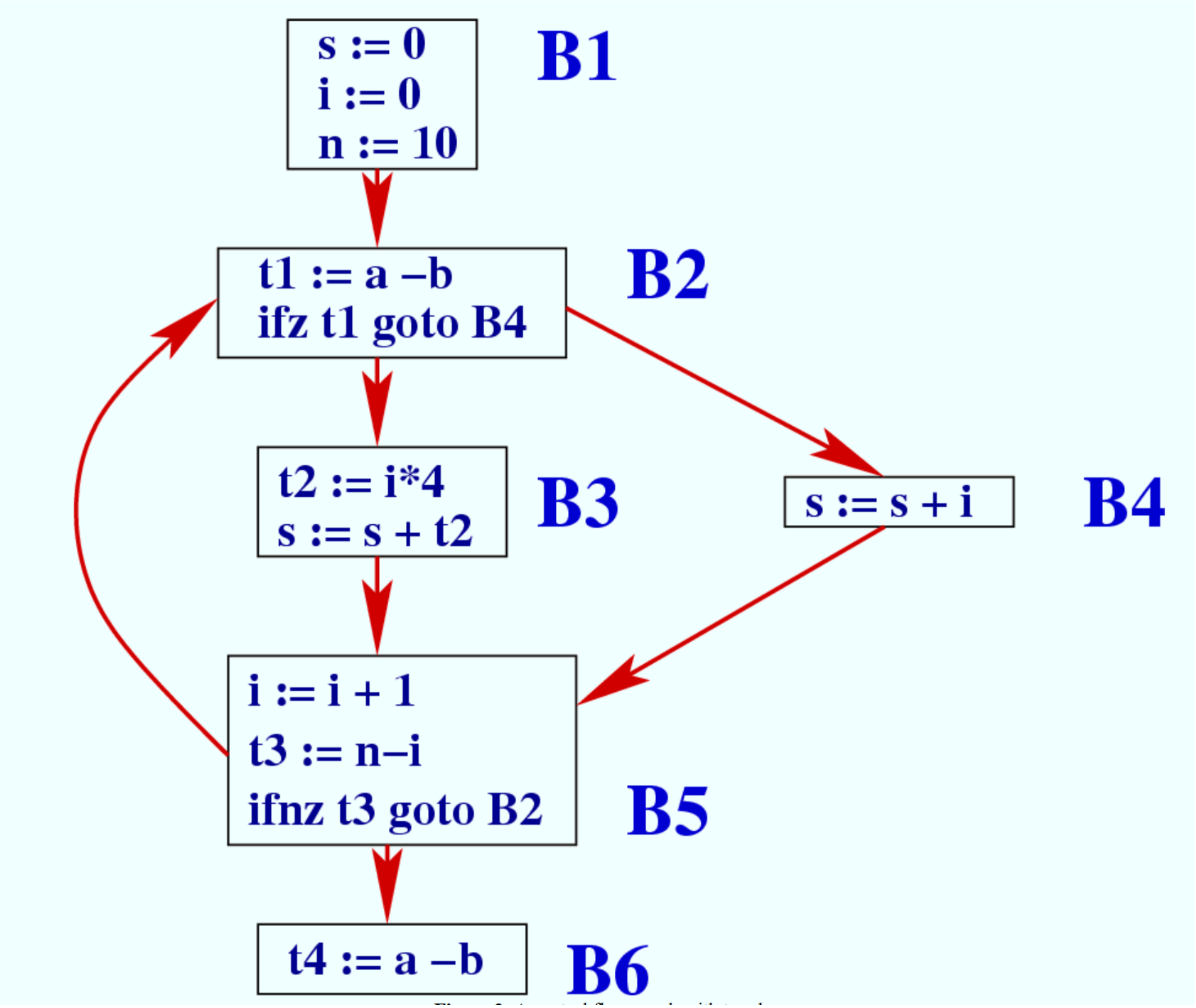
```
Label_x:
op1;
op2;
op3;
br label_z;
```

Two Basic Blocks

```
Label_x:
op1;
op2;
op3;

Label_y:
op4;
op5;
```

Example Control Flow Graph



How might they appear in a high-level language?

How many basic blocks?

```
…
if (x) {
    …
}
else {
   …
}
…
```

Single Basic Block

```
Label_x:
op1;
op2;
op3;
br label_z;
```

Two Basic Blocks

```
Label_x:
op1;
op2;
op3;

Label_y:
op4;
op5;
```

https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/CodeOptimization.html/node6.html

Local Value Numbering Optimization

Local value numbering

- A local optimization over 3 address code

- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)

- Can be extended to a regional optimization using flow analysis

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

We are going to use notation similar to Class IR applicable to any Intermediate reapresentation.

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

valid? →

```
a = b + c;
b = a - d;
c = a;
d = a - d;
```

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

valid? →

```
a = b + c;
b = a - d;
c = a;
d = a - d;
```

No! Because b is redefined

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

valid? →

```
a = b + c;
b = a - d;
c = b + c;
d = b;
```

Local value numbering

- A local optimization over 3 address code
- Attempts to replace arithmetic operations (expensive) with copy instructions (cheap)
- Can be extended to a regional optimization using flow analysis

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

valid? →

```
a = b + c;
b = a - d;
c = b + c;
d = b;
```

yes!

Local value numbering (LVN)

LVN Algorithm:

- Provide a number to each variable. Update the number each time the variable is updated.

- Keep a global counter; increment with new variables or assignments

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

Global_counter = 0

Local value numbering

Algorithm:

- Provide a number to each variable. Update the number each time the variable is updated.

- Keep a global counter; increment with new variables or assignments

```
a =  b0 + c1;
b =  a  - d;
c =  b  + c;
d =  a  - d;
```

Global_counter = 2

b and c have not been seen, give them a number

Local value numbering

Algorithm:

- Provide a number to each variable. Update the number each time the variable is updated.

- Keep a global counter; increment with new variables or assignments

```
a2 = b0 + c1;
b =  a2 - d3;
c =   b + c;
d =   a - d;
```

Global_counter = 4

a2 have been updated use the last defined number. i.e. a on RHS, becomes a2 and so on.

Local value numbering

Algorithm:

- Provide a number to each variable. Update the number each time the variable is updated (assigned to).

- Keep a global counter; increment with new variables or assignments

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

Global_counter = 7

So … for LHS variable always increment with updated number, for right hand side use, always appended last number version

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. **Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs**.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

If we remember to declare every one of these variable names, the IR is still valid !!!

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. **Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs**.
- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. **Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.**

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
      "b0 + c1" : "a2",
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
      "b0 + c1" : "a2",
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. **Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.**

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
       "b0 + c1" : "a2",
       "a2 - d3" : "b4",
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
        "b0 + c1" : "a2",
        "a2 - d3" : "b4",
}
```

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
}
```

mismatch due to re-numbering!!!

i.e. it is no longer just c = b + c

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
      “b0 + c1” : “a2”,
      “a2 - d3” : ”b4”,
      “b4 + c1” : “c5”,
}
```

Add new entry in Hash table with new Numbering.

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
        "b0 + c1" : "a2",
        "a2 - d3" : "b4",
        "b4 + c1" : "c5",
}
```

Do a look up and …

Local value numbering

Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = b4;
```

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
      "b4 + c1" : "c5",
}
```

...match!

What else can we do?

What else can we do?

Consider this snippet:

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

Commutative operations

What is the definition of commutative?

Commutative operations

What is the definition of commutative?

```
x OP y == y OP x
```

What operators are commutative? Which ones are not?

Adding commutativity to local value numbering

- For commutative operators (e.g. + *), the analysis should consider a deterministic order of operands.

- You can use variable numbers or lexigraphical order

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

cannot re-order because - is not commutative

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
      “c1 - b0” : “a2”,
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
        “c1 - b0” : “a2”,
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

re-ordered because a2 < d3 lexigraphically

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
      "c1 - b0" : "a2",
      "a2 * d3" : "f4",
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
       "c1 - b0" : "a2",
       "a2 * d3" : "f4",
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
        "c1 - b0" : "a2",
        "a2 * d3" : "f4",
        "b0 - c1" : "c5",
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
       "c1 - b0" : "a2",
       "a2 * d3" : "f4",
       "b0 - c1" : "c5",
}
```

Local value numbering: commutative operations

Algorithm optimization:

- for commutative operations, re-order operands into a deterministic order

And achieves more optimization opportunities …

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = f4;
```

```
H = {
      "c1 - b0" : "a2",
      "a2 * d3" : "f4",
      "b0 - c1" : "c5",
}
```

Generalized Loop Unrolling

Loop unrolling conditions

What about in the general case? For unroll factor F?

```
for (int i = x; i < y; i++) {
  // body
}
```

find out how many unrolled loops we can execute:
?
This gives us the first bound

second loop is initialized with the first bound

second loop's bound is same as the original loop

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = x; i < y/F *F; i++) {
  // body
  i++
  ...
}
```

Note that y/F * F creates a remainder
i.e. the benefit of integer arithmetic

```
for (int i = y/F *F; i < y; i++) {
  // body
}
```

Loop unrolling conditions

- general unroll

For unroll factor F

General unroll constraints:

- Loop update increments by 1
- Find the concrete number of loop iterations, LI

General unroll code generation:

- Create simple unrolled loop with new bound: (LI/F)*F
- Create cleanup (basic) loop with initialization: (LI/F)*F
- perform codegen

None of these numbers have to be concrete!