CSE110A: Compilers

June 9, 2023

Topics:

- *Homework review*
- *Class review*

Announcements

- Homework 5 due on Sunday
  - Given that our final is so early, I will give an extension until Wednesday
  - No office hours next week though
  - Piazza support will be sparse
- Homework 3 retesting is done today
  - If you fixed your exceptions
  - Also test 9 was off; some people failed when they shouldn’t have. We will update it
- Rest of grades coming ASAP. Plan is to be done by next thursday

Announcements

- Final: Monday June 12: 8 AM to 11 AM
  - 3 pages of notes, front and back
  - comprehensive
  - like the midterm, but 4 questions instead of 3

- Do not miss the final!
  - Any accommodations must go through DRC

Quiz

Quiz

Is the following loop a DOALL loop?

```
for (int i = 0; i < 3; i+=1) {
  a[i] = a[i+1] + a[i+2];
}
```

Quiz

Is the following loop a DOALL loop?

```
for (int i = 0; i < 3; i+=1) {
  a[i] = b[i+1] + c[i+2];
}
```

Quiz

We talked about several optimizations for DOALL loops. Try to think of another optimization that might be possible and write a few sentences about it.

Quiz

We talked about image processing being a good domain for DOALL loops. Can you think of any other domains? Briefly describe the domain and how it has DOALL loops.

Quiz

This is the last lecture of module 4. Please write any feedback you have about the module. Thank you and see you for the last day of class on Friday!

Thanks!

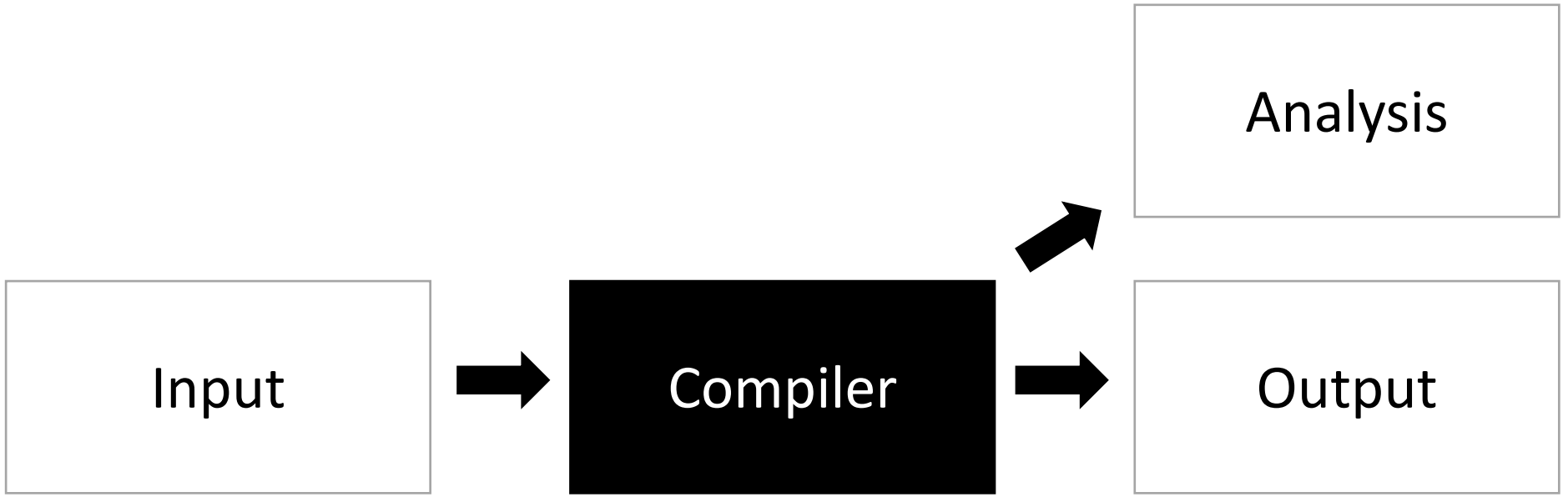
Homework review

- Command line

Class Review

What is a compiler?

What are some examples here?



Warnings
Errors
Performance logs

Strings belonging to language L

A series of statements in programming language L

A program written in C

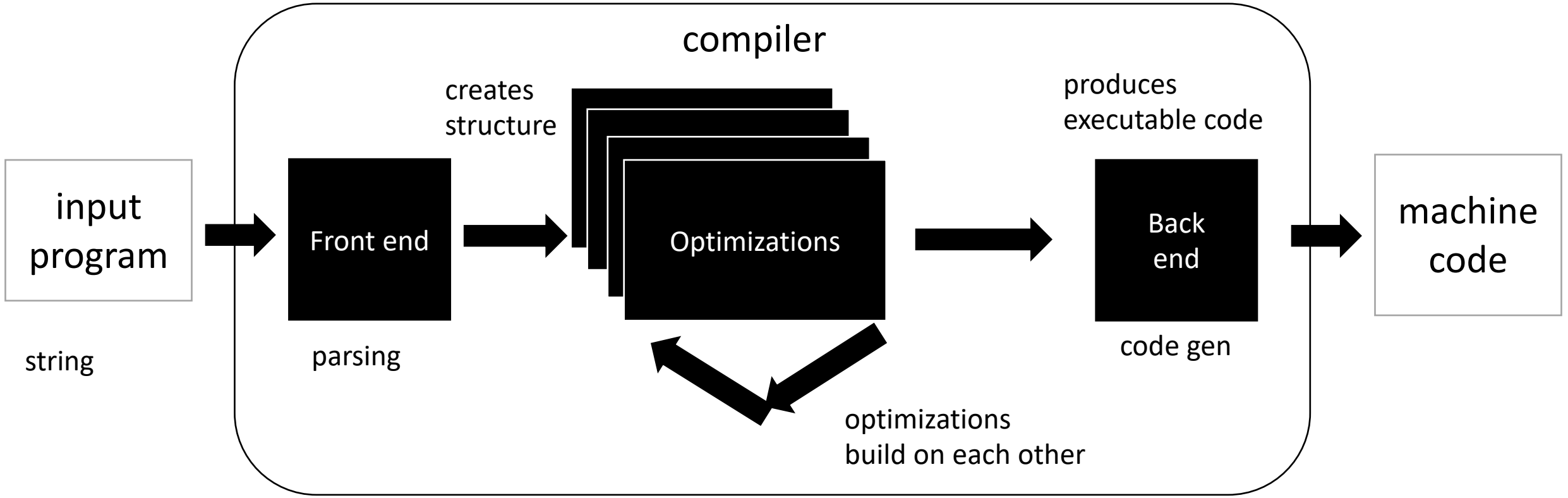
gcc/clang

Strings belonging to language L’

An executable binary file in an ISA language

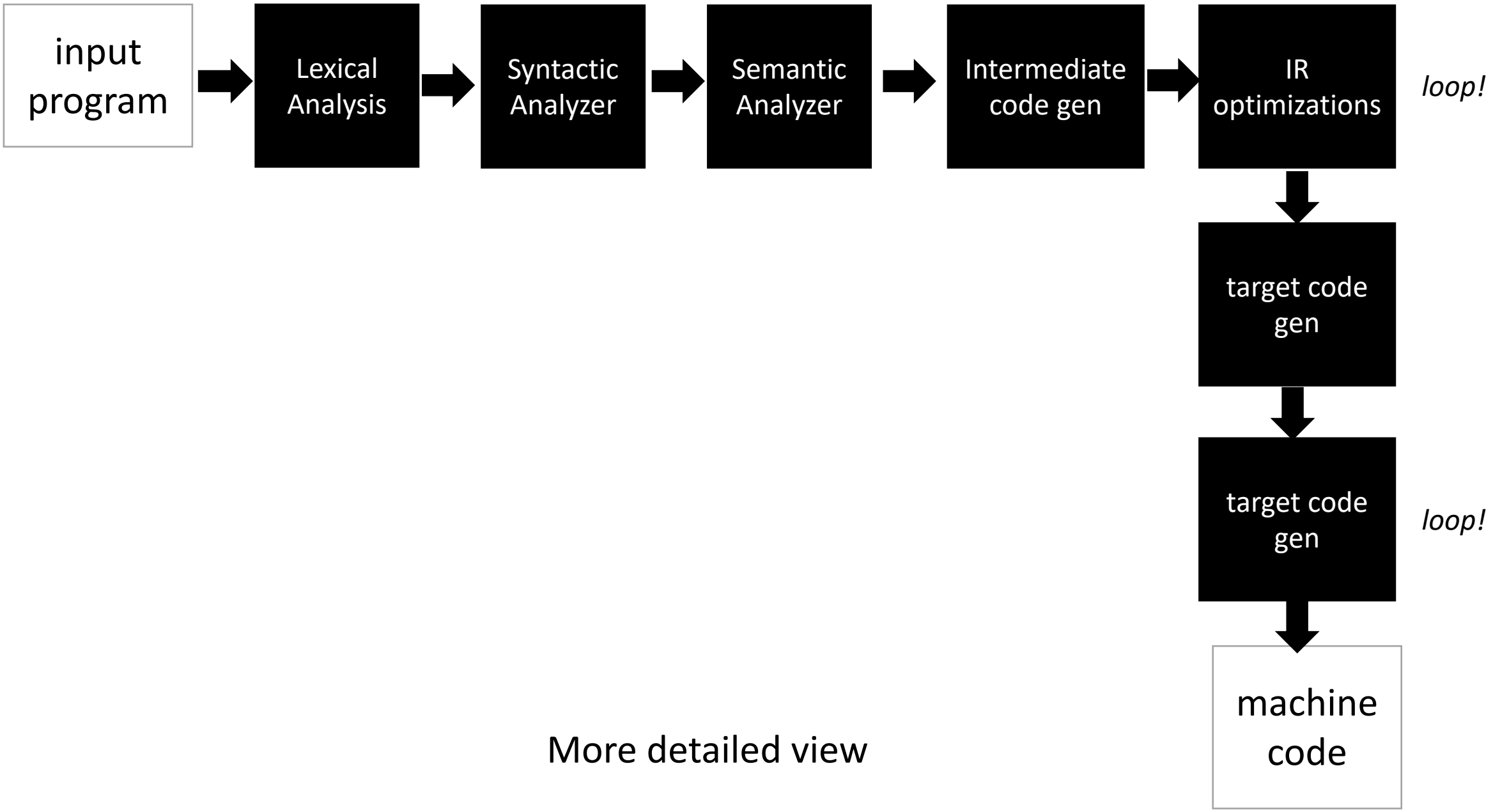
An x86 Binary executable

Compiler Architecture



Medium detailed view

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



More detailed view

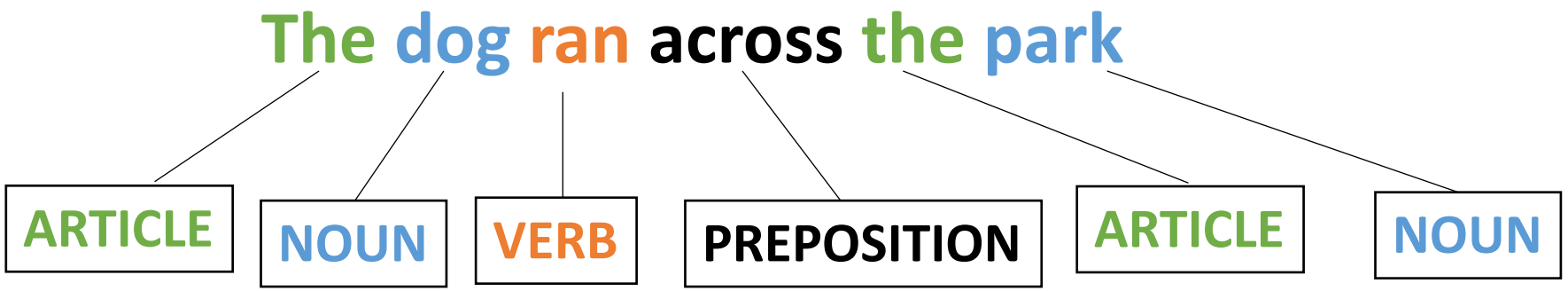
Parsing is the first step in a compiler

- How do we parse a sentence in English?

The dog ran across the park

Parsing is the first step in a compiler

- How do we parse a sentence in English?



Programs for Lexical Analysis

Scanner (sometimes called lexer)

Defined by a list of tokens and definitions:

Tokens		Tokens Definitions
• ARTICLE	=	{The, A, My, Your}
• NOUN	=	{Dog, Car, Computer}
• VERB	=	{Ran, Crashed, Accelerated}
• ADJECTIVE	=	{Purple, Spotted, Old}

Scanner API

What do we want?

“My Old Computer Crashed”

Scanner

Scanner API

What do we want?

“My Old Computer Crashed”

Scanner

```
[(ARTICLE, “My”), (ADJECTIVE, “Old”), (NOUN, “Computer”), (VERB, “Crashed”)]
```

***Lexeme**: (TOKEN, value)*

Longest possible match

Consider the token:

- `CLASS_TOKEN = {“cse”, “110”, “cse110”}`

What would the lexemes be for: “cse110”

options:

- `(CLASS_TOKEN, ”cse”) (CLASS_TOKEN, ”110”)`
- `(CLASS_TOKEN, ”cse110”)`

This one!

Longest possible match

- Important for operators, e.g. in C
- `++, +=`

how would we scan “`x++;`”

```
[(ID, “x”), (ADD, “+”), (ADD, “+”), (SEMI, “;”)]
```

```
[(ID, “x”), (INCREMENT, “++”), (SEMI, “;”)]
```

Let’s write tokens as regular expressions

- For our simple programming language

```
ID     = [a-z]+
NUM    = [0-9]+
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = [" "]
```

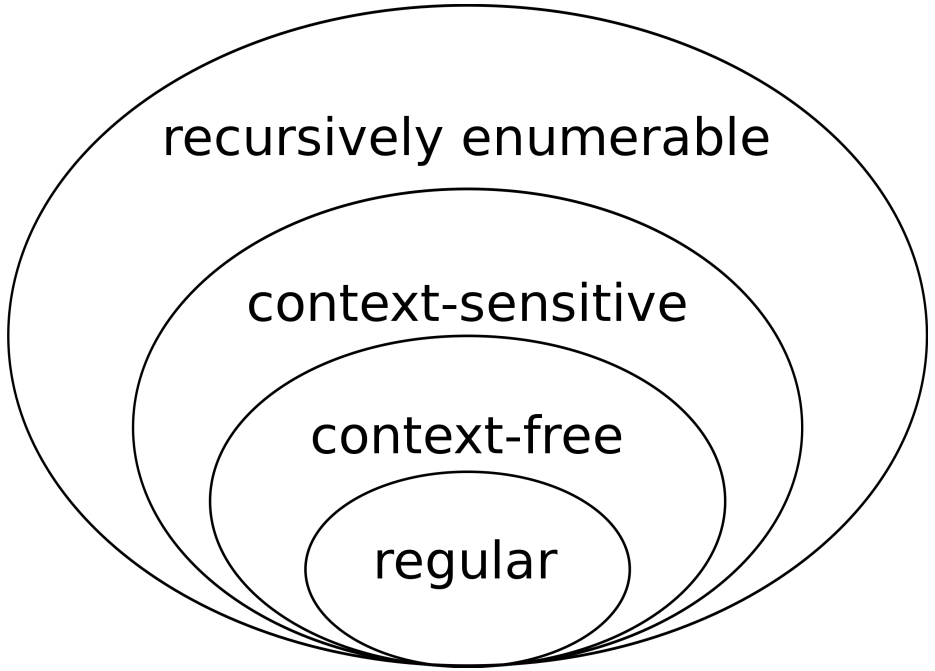
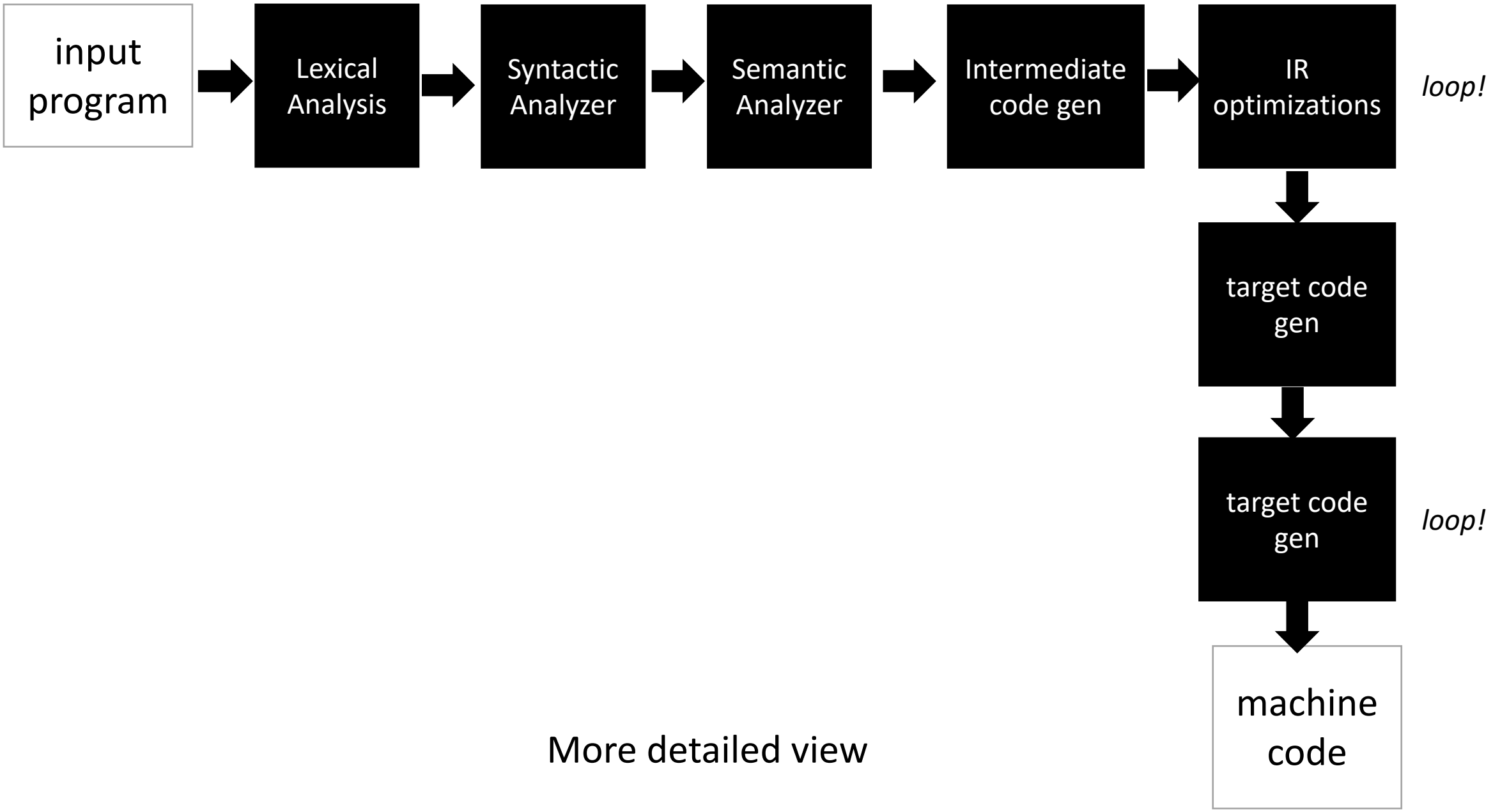


image source: wikipedia

Scanner implementations

- Naïve scanner:
  - Pros/cons?
- Exact match scanner
  - Pros/cons?
- Start of string scanner
  - Pros/cons?
- Named group scanner
  - Pros/cons?



More detailed view

Parsing

- Use CFGs to express our grammar
  - Why?
- CFGs consist of production rules and terminals
- production rules can be recursive

```
Examples:

add_expr ::= NUM PLUS NUM

mult_expr ::= NUM TIMES NUM

joint_expr ::= add_expr TIMES add_expr

simple_expr ::= simple_expr PLUS NUM
              |   simple_expr TIMES NUM
              |   NUM
```

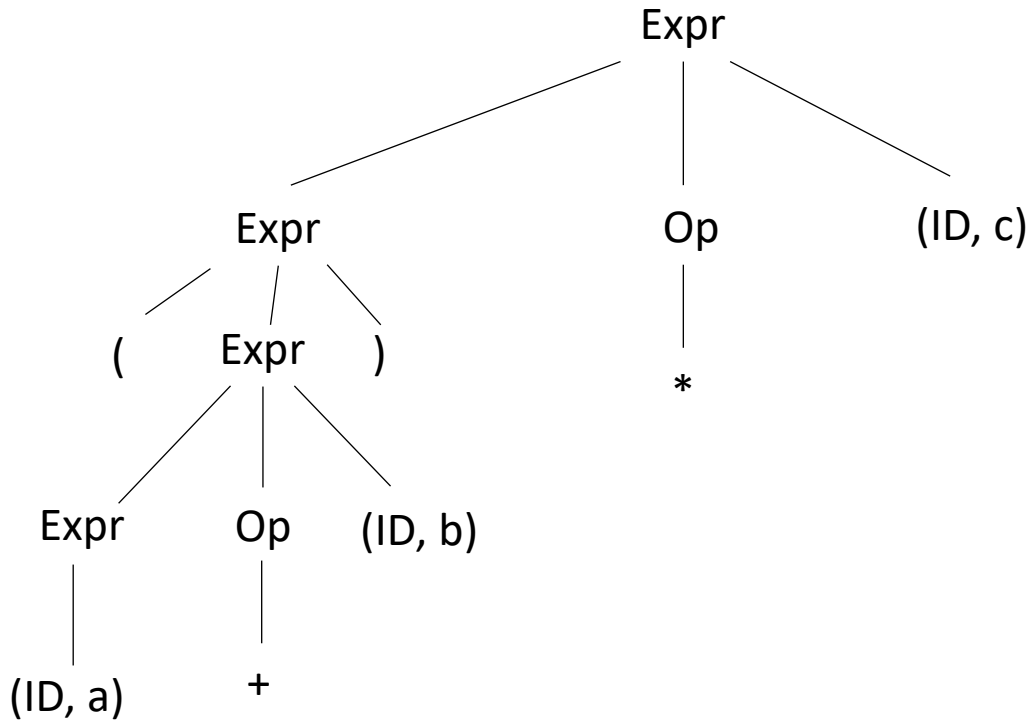
A more complicated derivation

```
1: Expr ::= '(' Expr ')'
2:        |    Expr Op ID
3:        |    ID
4: Op   ::= '+'
5: Op   |   '*'
```

*Are there other ways to derive (a+b)*c?*

RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID

We can visualize this as a tree:

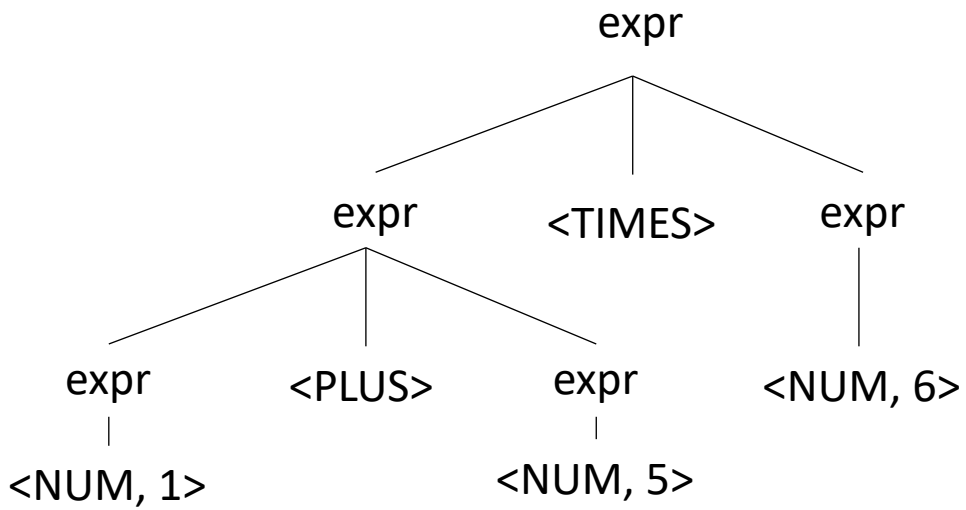


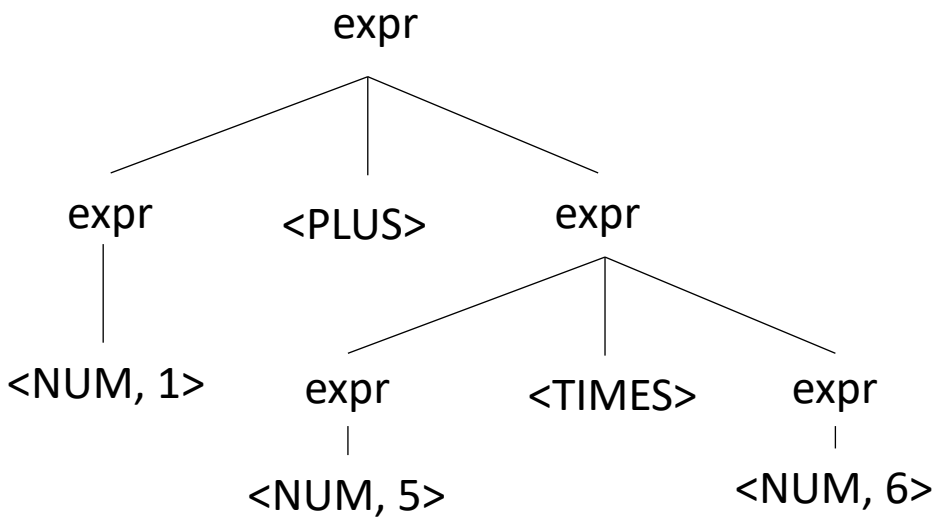
Ambiguous grammars

- `input: 1 + 5 * 6`

```
expr ::= NUM
       | expr PLUS expr
       | expr TIMES expr
       | LPAREN expr RPAREN
```

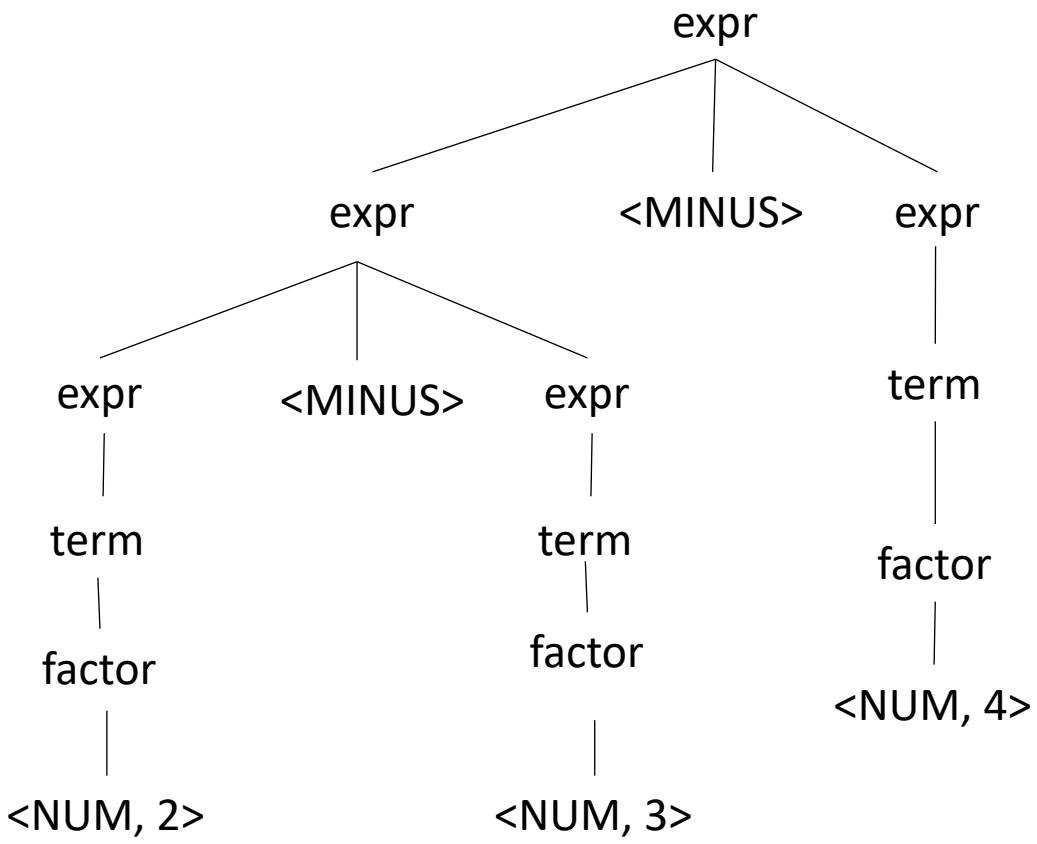
Two possible parse trees for the same input



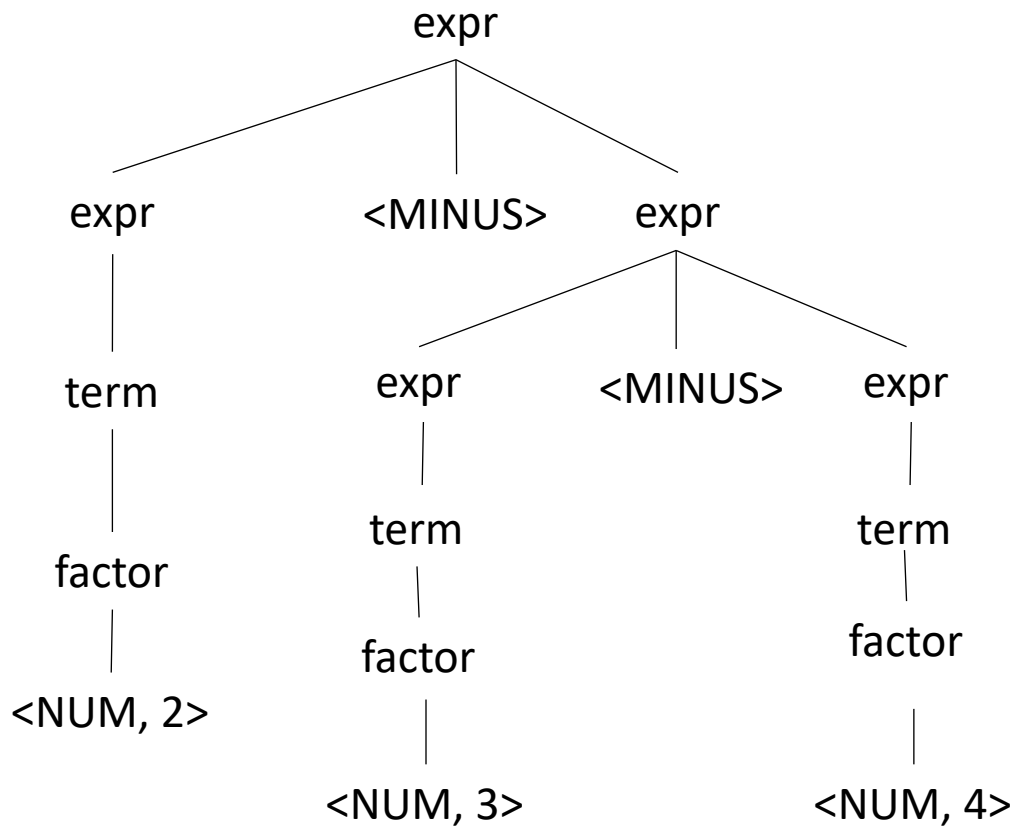


Does not correctly encode precedence!

More ambiguous grammars



`input: 2-3-4`



Which one is right?

How to avoid ambiguous grammars

Let's do operators [+,,-,/,^] and ( )*

Operator	Name	Productions
+,-	expr	: expr PLUS term | expr MINUS term | term
*,/	term	: term TIMES pow | term DIV pow | pow
^	pow	: factor CARROT pow | factor
()	factor	: LPAR expr RPAR | NUM

```
Tokens:
    NUM    = "[0-9]+"
    PLUS   = '\+'
    TIMES  = '\*'
    LP     = '\('
    RP     = \)'
    MINUS  = '-'
    DIV    = '/'
    CARROT =' \^'
```

Implementing parsers

```
root = start symbol;
focus = root;
push(None);
to_match = s.token();

while (true):
  if (focus is a nonterminal)
    pick next rule (A ::= B1,B2,B3...BN);
    push(BN... B3, B2);
    focus = B1

  else if (focus == to_match)
    to_match = s.token()
    focus = pop()

  else if (to_match == None and focus == None)
    Accept
```

What could a demonic choice do?

Variable	Value
focus	
to_match	
s.istring	
stack	

```
1: Expr   ::= Expr '+' ID
2:        |   ID
```

Can we derive the string a

Expanded Rule	Sentential Form
start	Expr

Eliminating direct left recursion

A = Op Unit
B = Unit

```
1: Expr  ::= Expr Op Unit
2:       |   Unit
3: Unit  ::= ‘(‘ Expr ‘)’
4:       |    ID
5: Op    ::= ‘+’
6:       |   ‘*’
```

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:       |   “”
```

Lets do this one as an example:

```
Fee ::= Fee A
    |   B
```

```
Fee  ::= B Fee2

Fee2 ::= A Fee2
       |   “”
```

The First+ Set

The First+ set is the combination of First and Follow sets

```
                                First sets:    Follow sets:     First+ sets:
1: Expr  ::= Unit Expr2         1: {‘(‘, ID}   1: NA            1: {‘(‘, ID}
2: Expr2 ::= Op Unit Expr2      2: {‘+’, ‘*’}  2: NA            2: {‘+’, ‘*’}
3:       | “”                   3: {“”}        3: {None, ’)’}   3: {None, ’)’}
4: Unit  ::= ‘(‘ Expr ‘)’       4: {‘(‘}       4: NA            4: {‘(‘}
5:       |    ID                5: {ID}        5: NA            5: {ID}
6: Op    ::= ‘+’                6: {‘+’}       6: NA            6: {‘+’}
7:       |   ‘*’                7: {‘*’}       7: NA            7: {‘*’}
```

Do we need backtracking?

The First+ set is the combination of First and Follow sets

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | ""
4: Unit  ::= '(' Expr ')'
5:         |    ID
6: Op    ::= '+'
7:         |   '*'
```

First+ sets:

```
1: {'(', ID}
2: {'+', '*'}
3: {None, ')'}
4: {'('}
5: {ID}
6: {'+'}
7: {'*'}
```

These grammars are called LL(1)

- L - scanning the input left to right
- L - left derivation
- 1 - how many look ahead symbols

They are also called predictive grammars

Many programming languages are LL(1)

For each non-terminal: if every production has a disjoint First+ set then we do not need any backtracking!

Recursive descent parser

Recursive descent parser

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

How do we parse an Expr?
We parse a Unit followed by an Expr2

We can just write exactly that!

```
def parse_Expr(self):
        self.parse_Unit();
        self.parse_Expr2();
        return
```

Recursive descent parser

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

How do we parse an Expr2?

Recursive descent parser

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

How do we parse an Expr2?

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Recursive descent parser

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

How do we parse an Expr2?

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

```
def parse_Expr2(self):

    token_id = get_token_id(self.to_match)

    # Expr2 ::= Op Unit Expr2
    if token_id in ["PLUS", "MULT"]:
        self.parse_Op()
        self.parse_Unit()
        self.parse_Expr2()
        return

    # Expr2 ::= ""
    if token_id in [None, ”RPAR"]:
        return

    raise ParserException(... # observed token
                          ["PLUS", "MULT", ”RPAR"])   # expected token
```

Symbol Table

Consider this simple programming language:

ID = [a-z]+

INCREMENT = “\+\+”

TYPE = “int”

LBRAC = “{“

RBRAC = “}”

SEMI = “;”

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

statements are either a declaration or an increment

How to implement a symbol table?

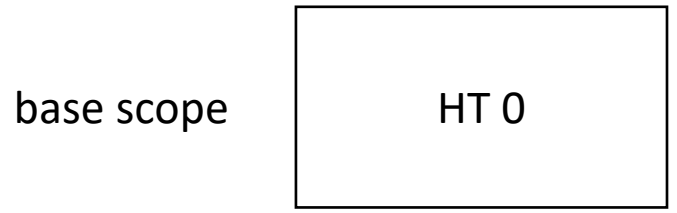
- Thoughts? What data structures are good at mapping strings?

- Symbol table
- four methods:
  - **`lookup(id) :`** `lookup an id in the symbol table. Returns None if the id is not in the symbol table.`
  - **`insert(id,info) :`** `insert a new id into the symbol table along with a set of information about the id.`
  - **`push_scope() :`** `push a new scope to the symbol table`
  - **`pop_scope() :`** `pop a scope from the symbol table`

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:



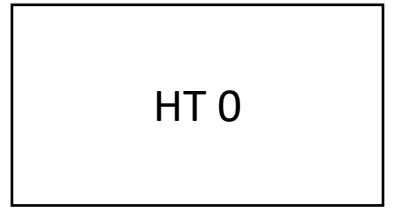
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`push_scope()`

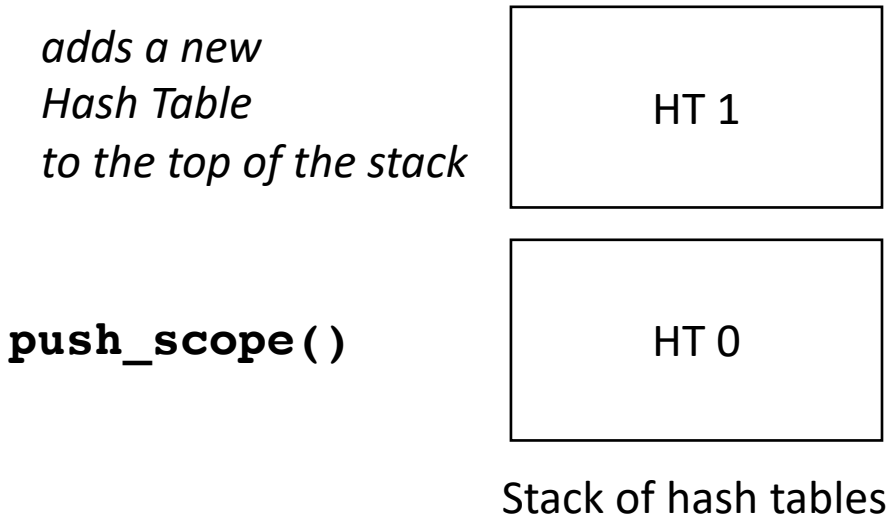


Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

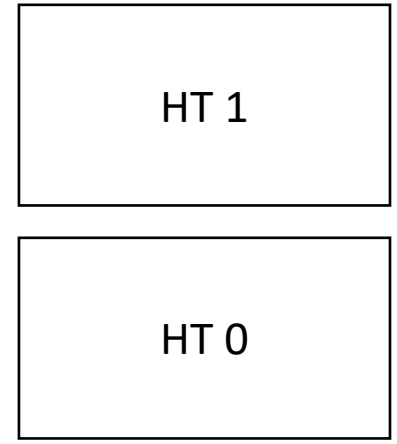
- A good way is a stack of hash tables:



How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:



`insert(id,data)`

Stack of hash tables

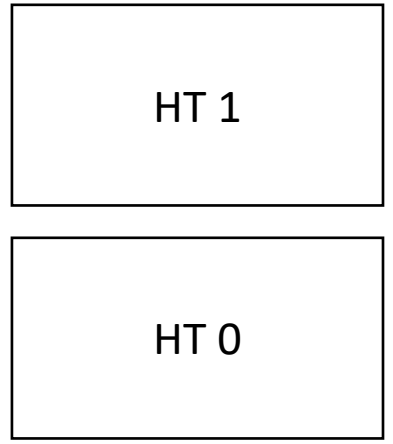
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`insert(id,data)`

insert `(id -> data)` at top hash table

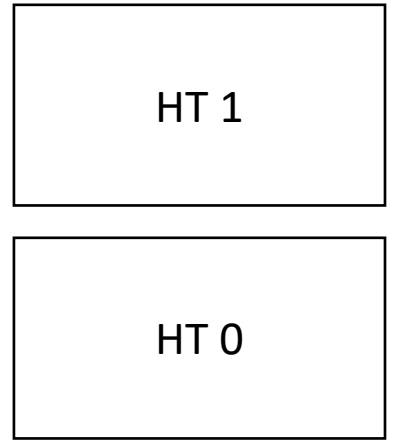


Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:



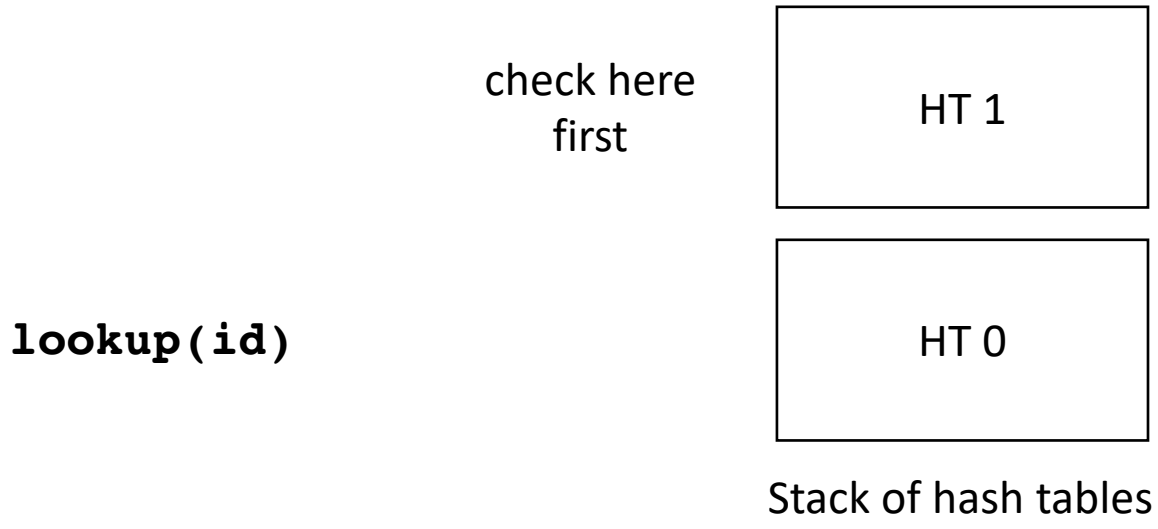
Stack of hash tables

`lookup(id)`

How to implement a symbol table?

- Many ways to implement:

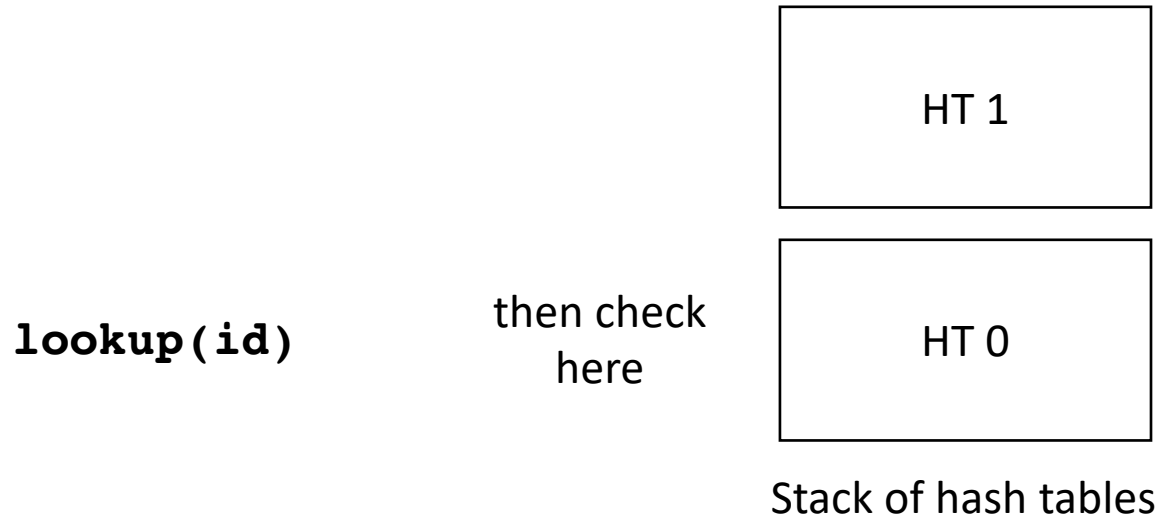
- A good way is a stack of hash tables:



How to implement a symbol table?

- Many ways to implement:

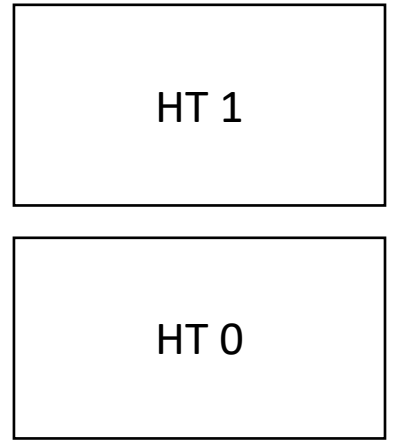
- A good way is a stack of hash tables:



How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:



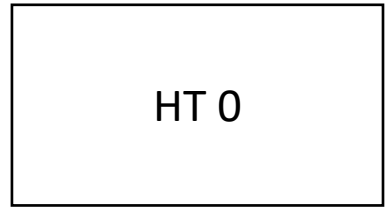
`pop_scope()`

Stack of hash tables

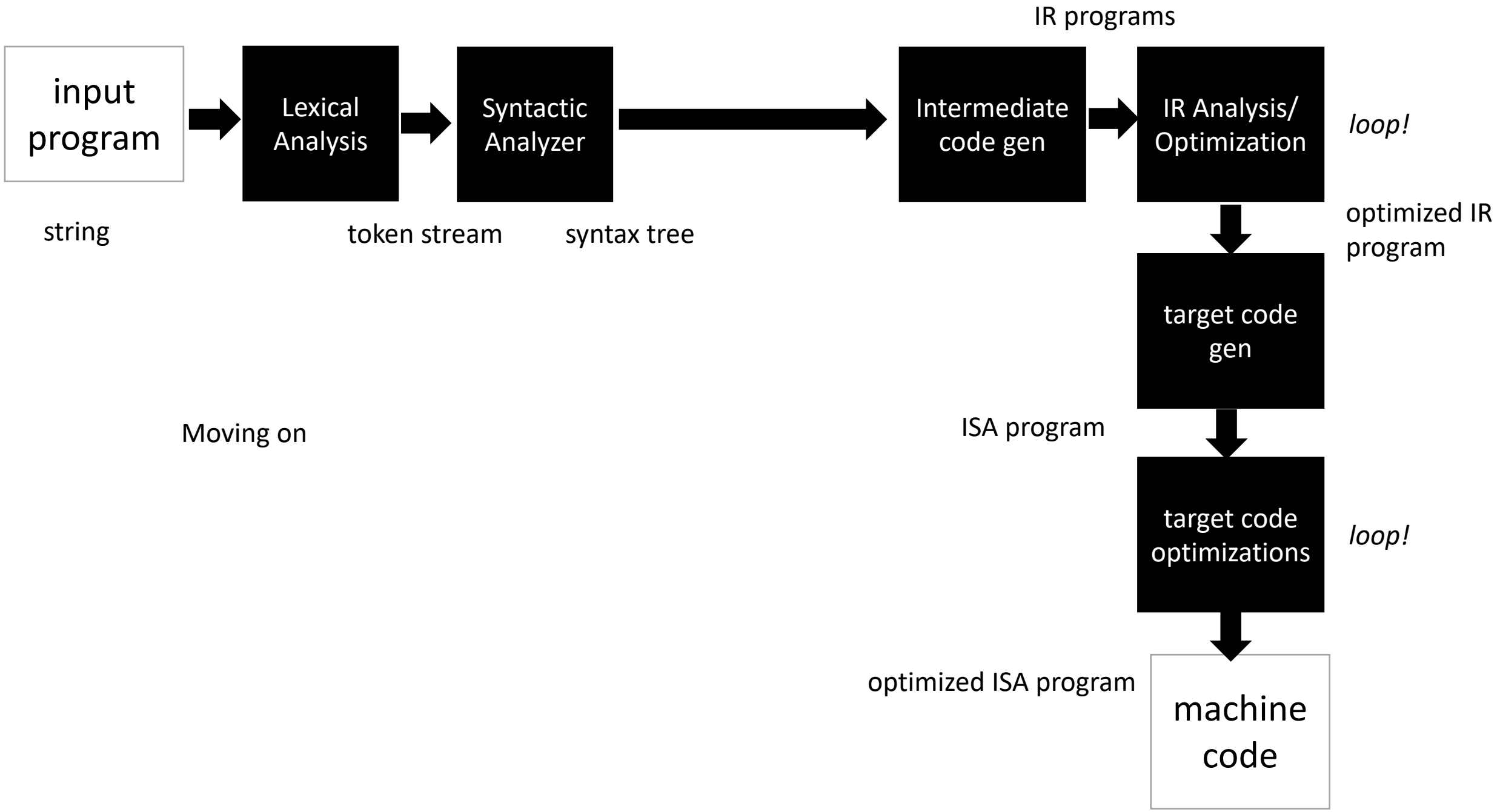
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:



Stack of hash tables

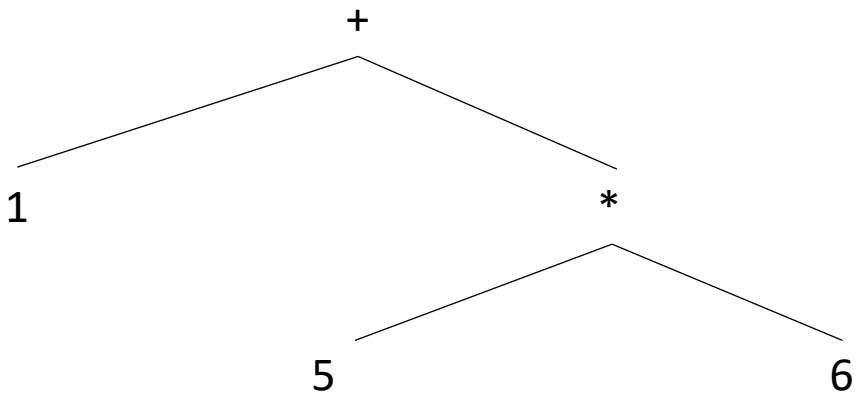

IR programs
input program
Lexical Analysis
Syntactic Analyzer
Intermediate code gen
IR Analysis/ Optimization
loop!
string
token stream
syntax tree
optimized IR program
target code gen
Moving on
ISA program
target code optimizations
loop!
optimized ISA program
machine code

First IR: Abstract Syntax Tree

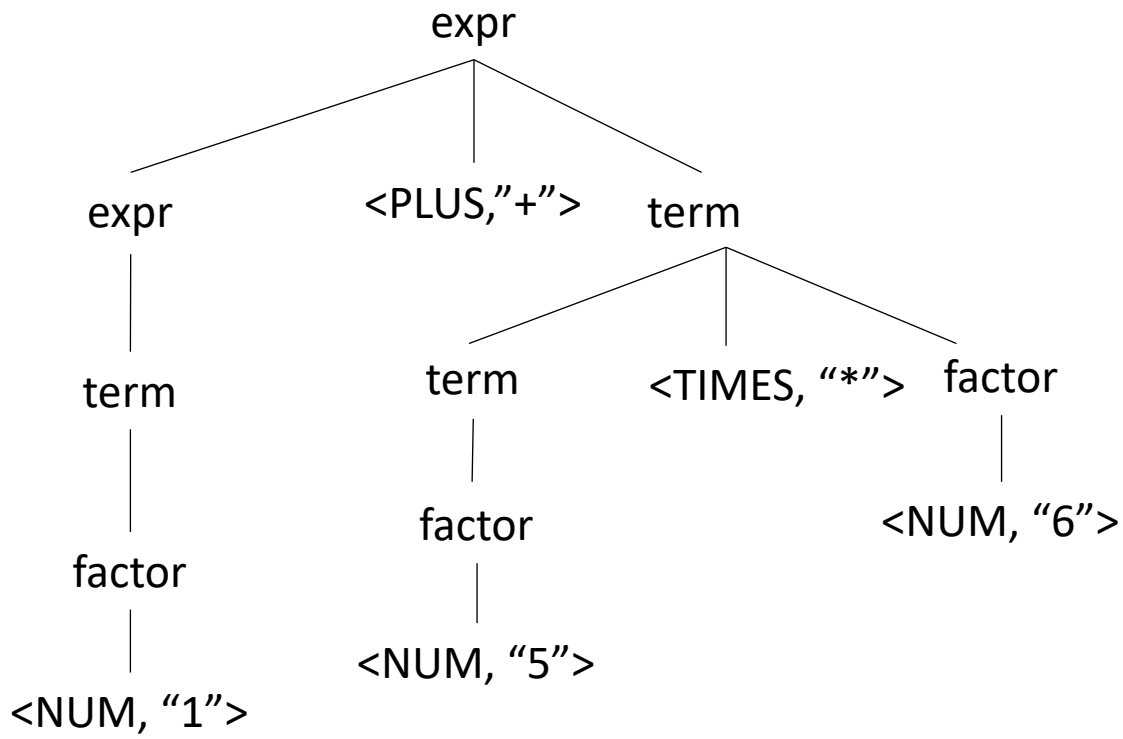
`input: 1+5*6`

What is an AST?

What are some differences?



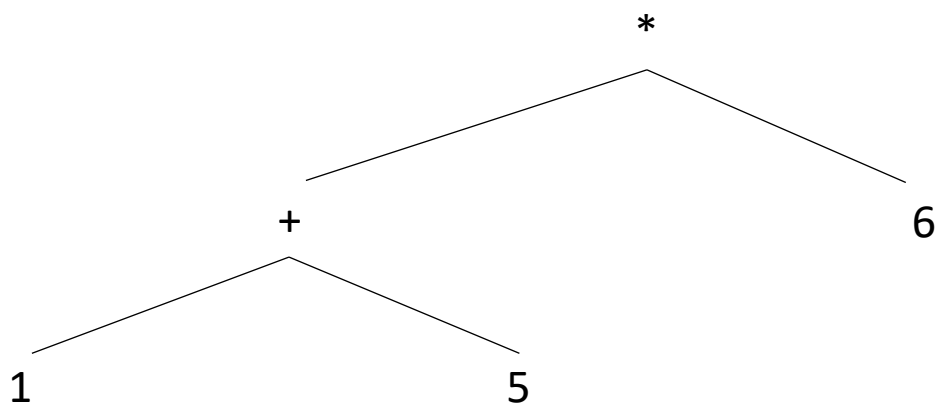
AST



Parse Tree

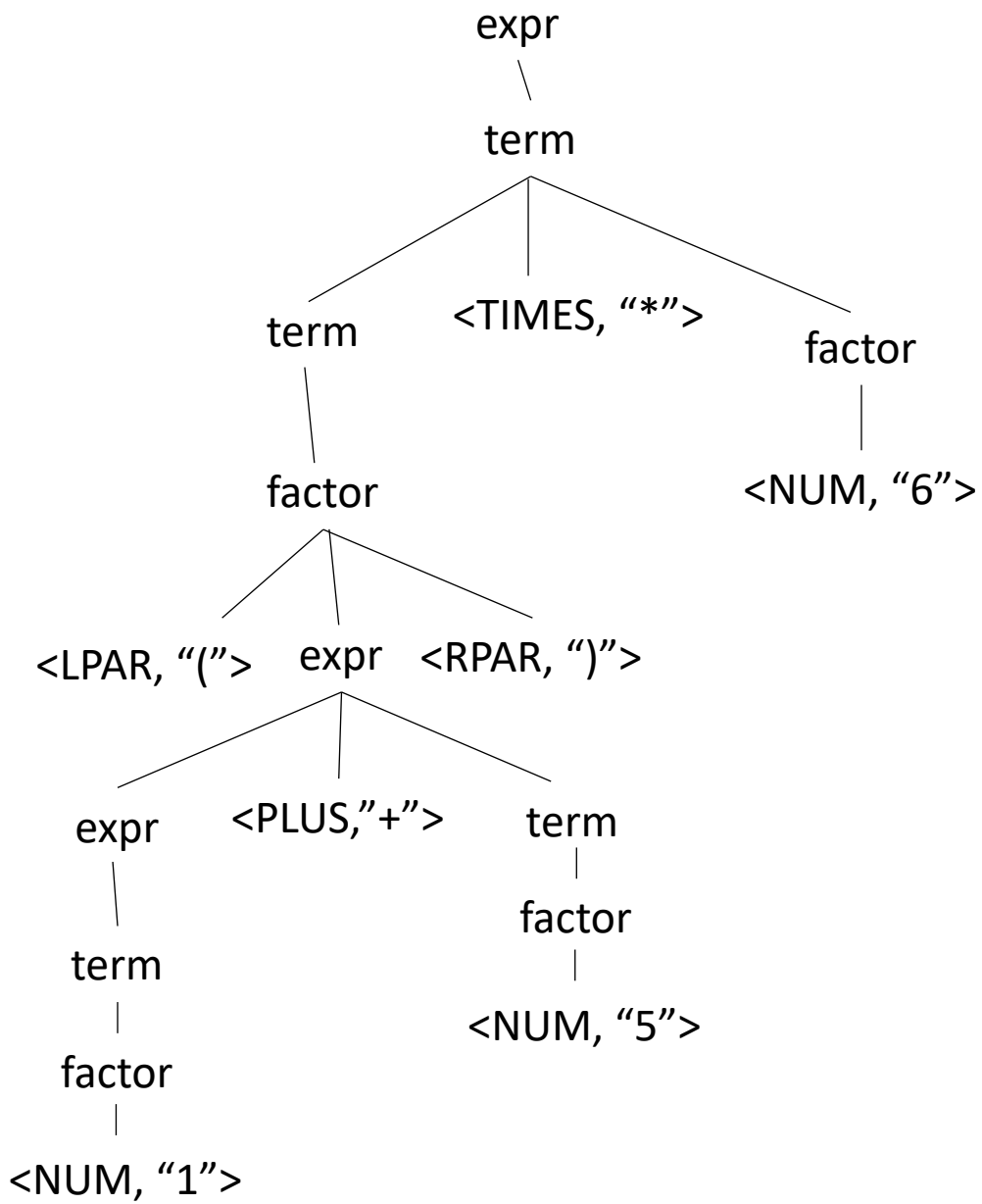
Example

what happens to ()s in an AST?

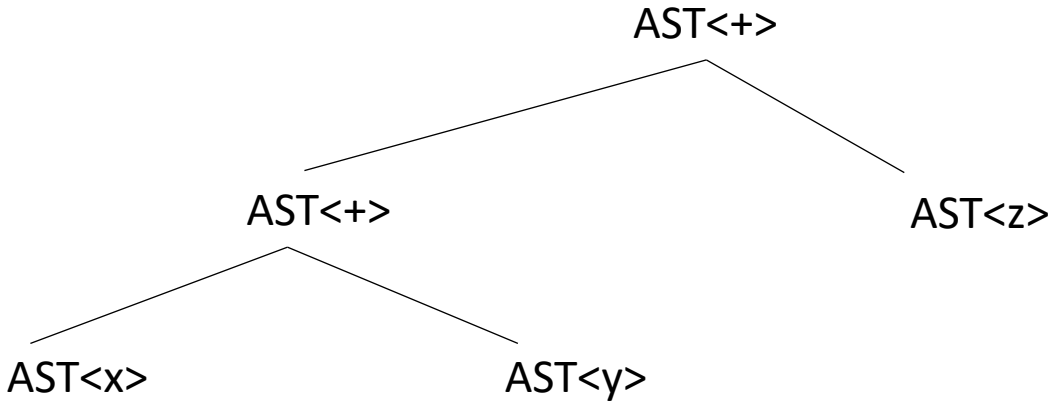


No need for (), they simply capture precedence. And now we have precedence in the AST tree structure

`input: (1+5)*6`



Evaluate an AST by doing a post order traversal

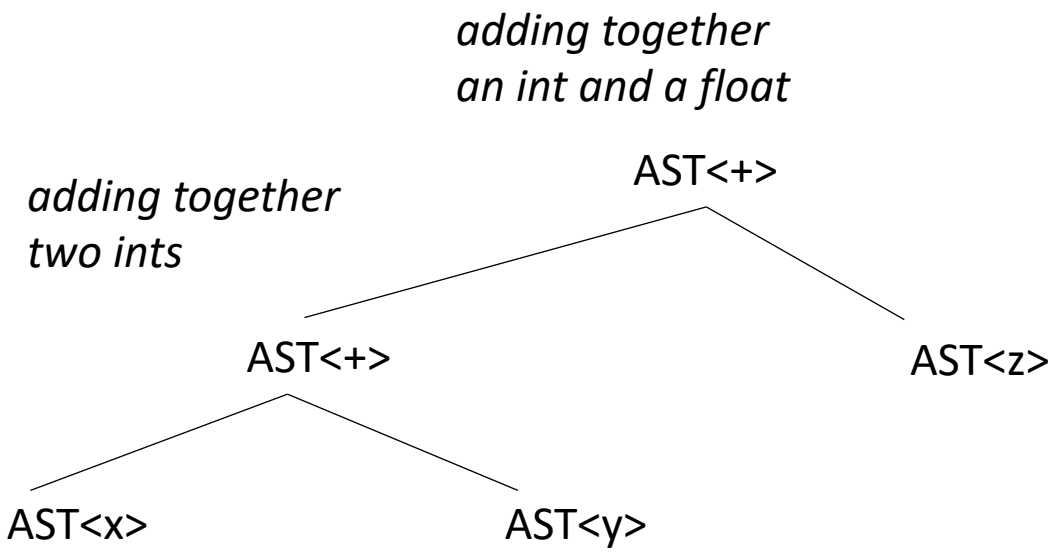


What if you cannot evaluate it?
What else might you do?

```
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



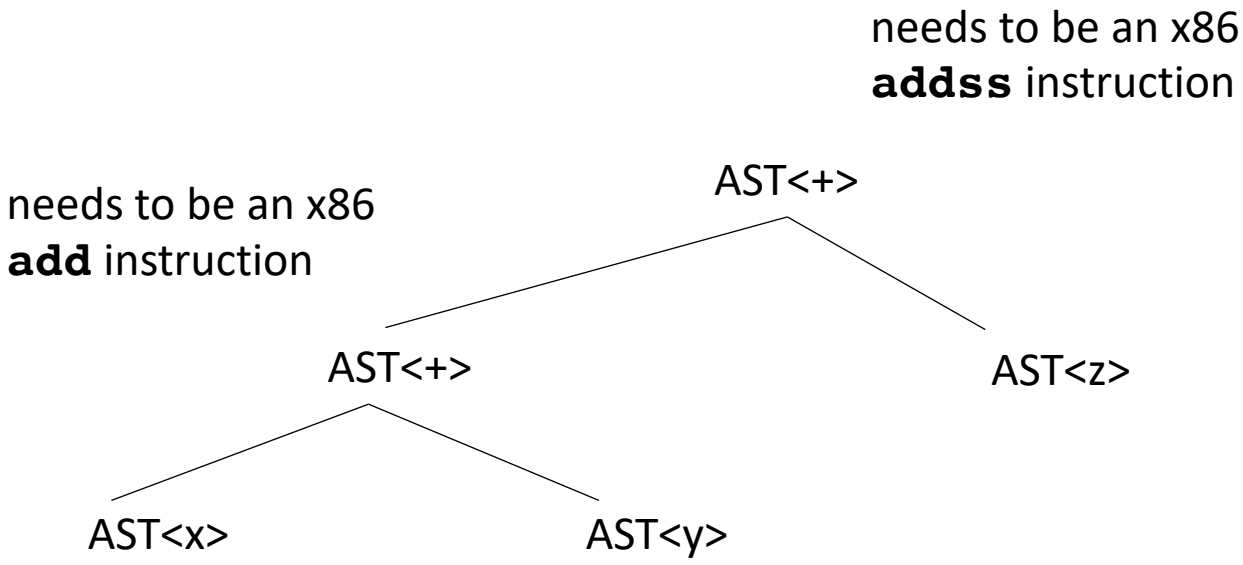
What if you cannot evaluate it?
What else might you do?

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

How does this change things?

in many languages this is fine, but we are working towards assembly language

Evaluate an AST by doing a post order traversal



`add r0 r1` - interprets the bits in the registers as integers and adds them together

`addss r0 r1` - interprets the bits in the registers as floats and adds them together

Evaluate an AST by doing a post order traversal

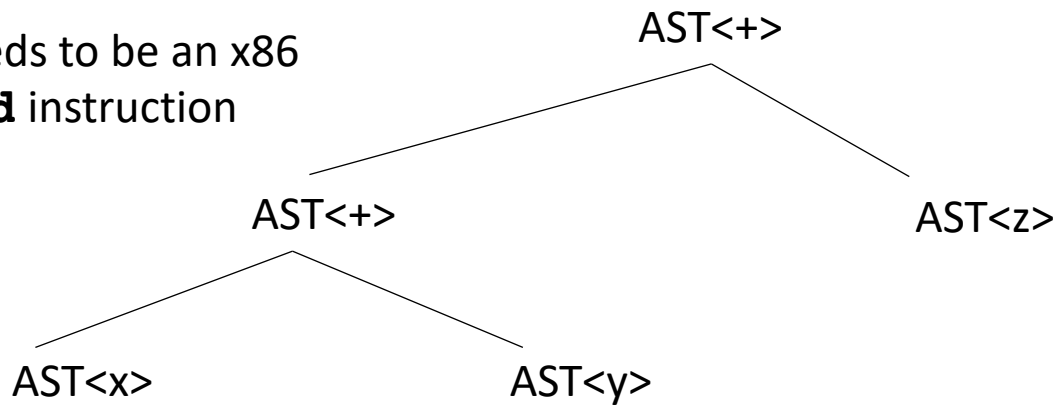
```
int x;
int y;
float z;
float w;
w = x + y + z
```

Lets do some experiments.

What should 5 + 5.0 be?

needs to be an x86 **addss** instruction

needs to be an x86 **add** instruction



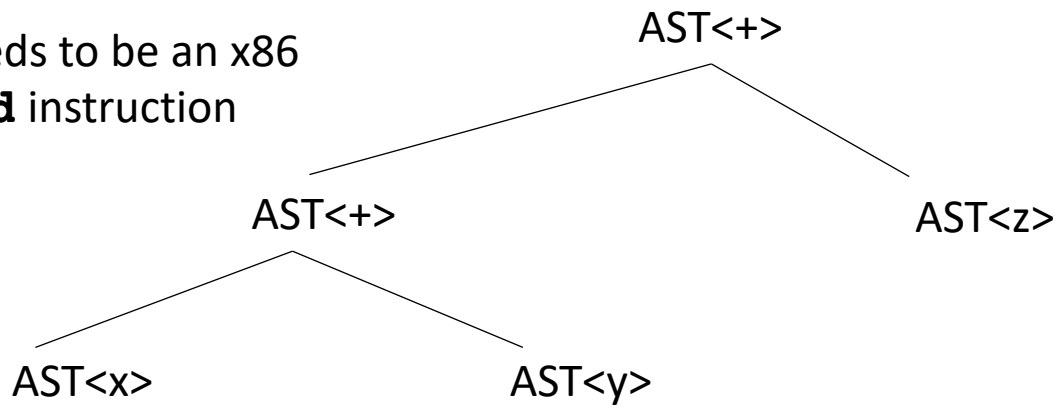
Is this all?

Evaluate an AST by doing a post order traversal

```
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

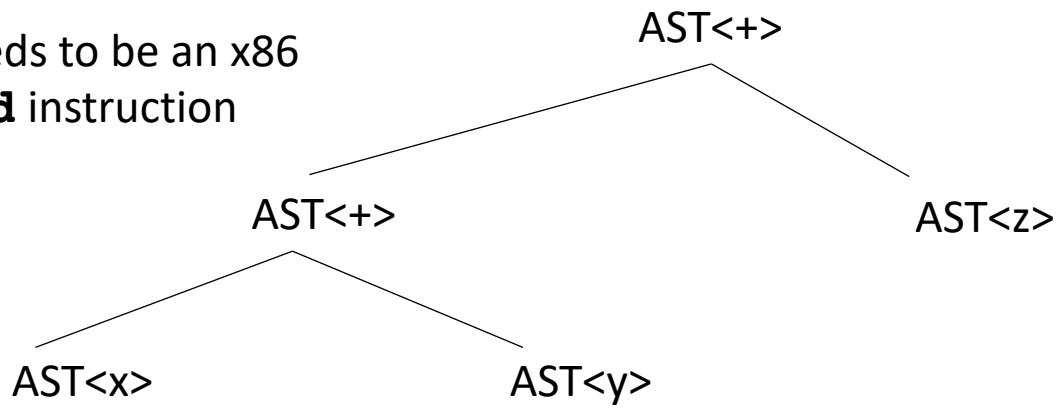
interprets both registers as floats

Evaluate an AST by doing a post order traversal

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

needs to be an x86 **addss** instruction

needs to be an x86 **add** instruction



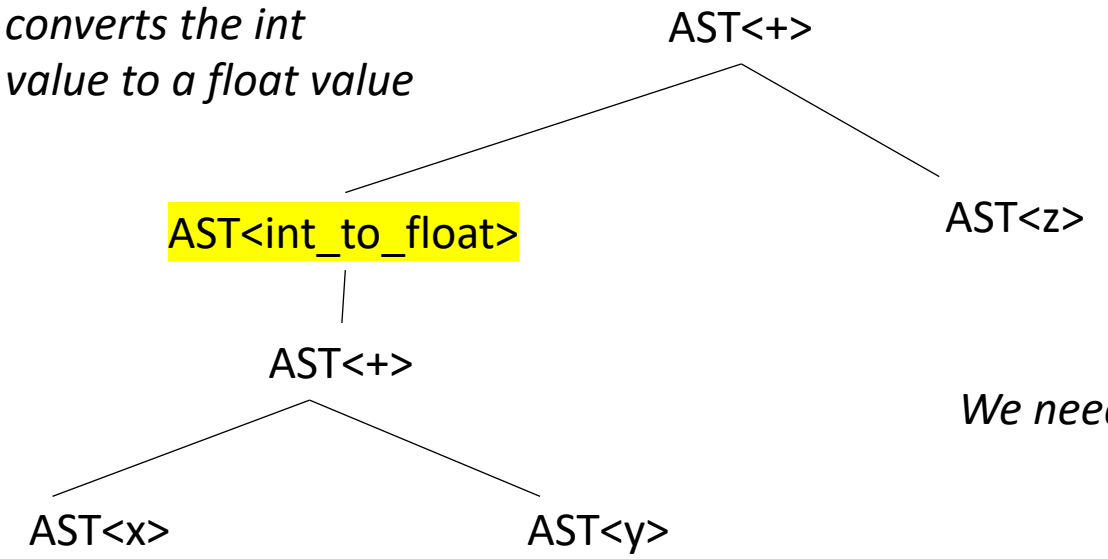
But the binary of 5 is 0b101
the float value of 0b101 is 7.00649232162e-45

We cannot just add them!

Is this all?

Evaluate an AST by doing a post order traversal

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

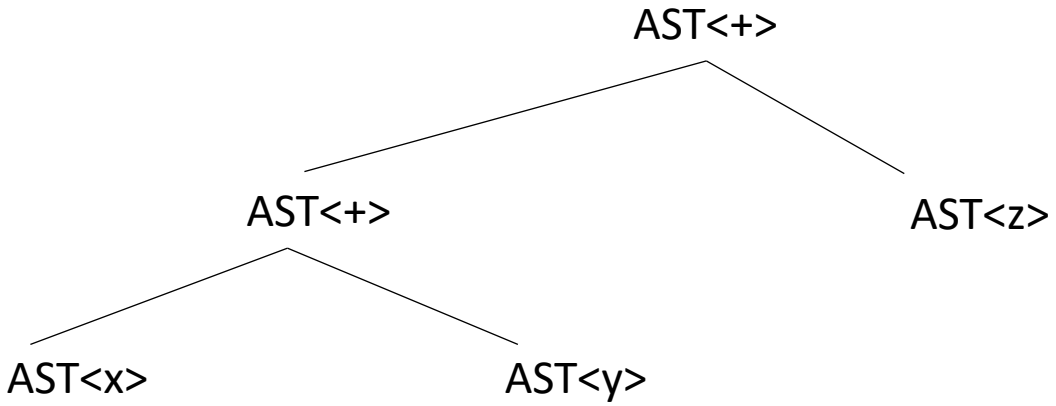


We need to make sure our operands are in the right format!

Type inference on an AST

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

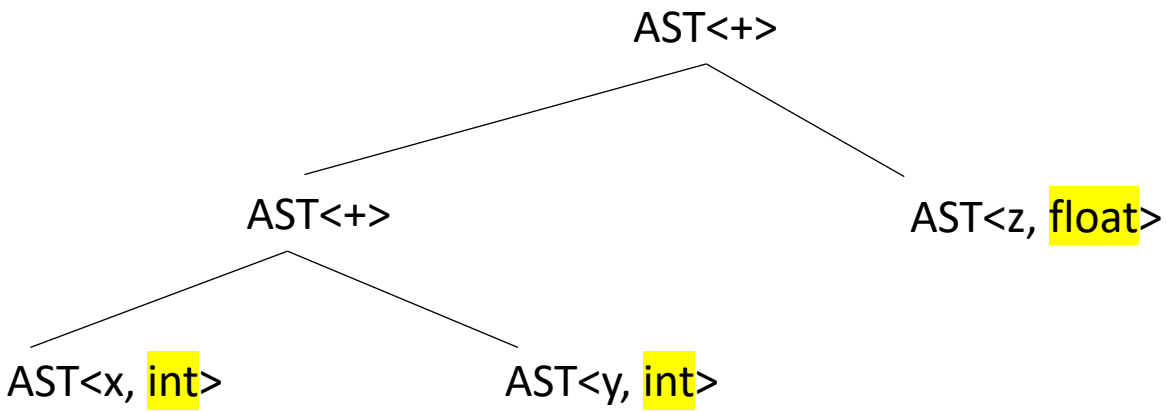
each node additionally gets a type



Type inference on an AST

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

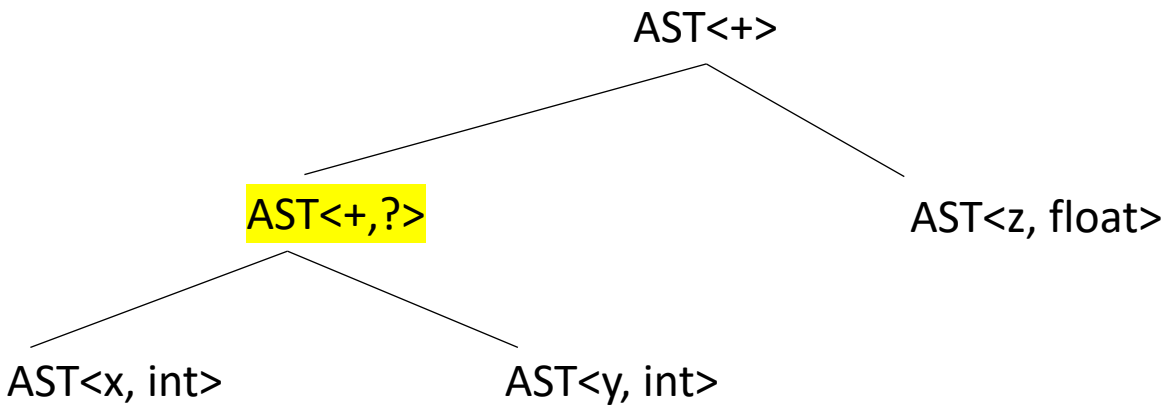
each node additionally gets a type
we can get this from the symbol table for the leaves



Type inference on an AST

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

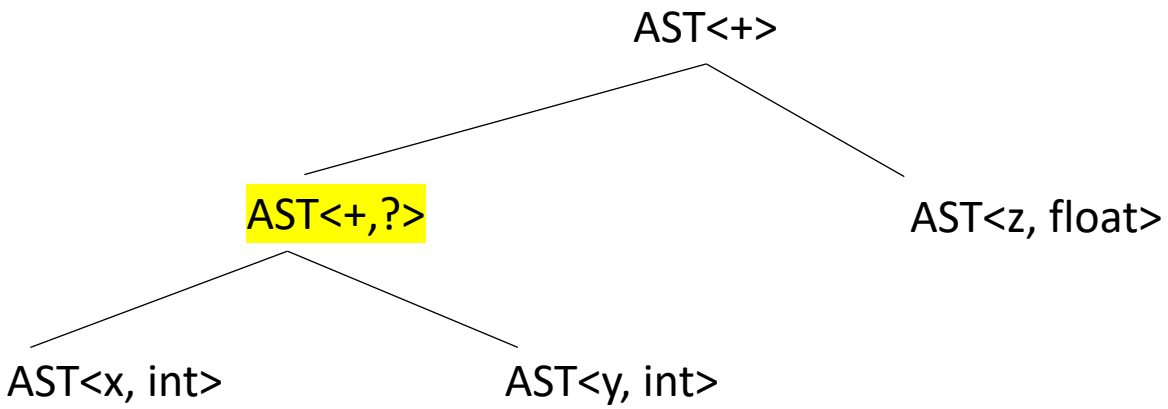
How do we get the type for this one?



Type inference on an AST

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

How do we get the type for this one?



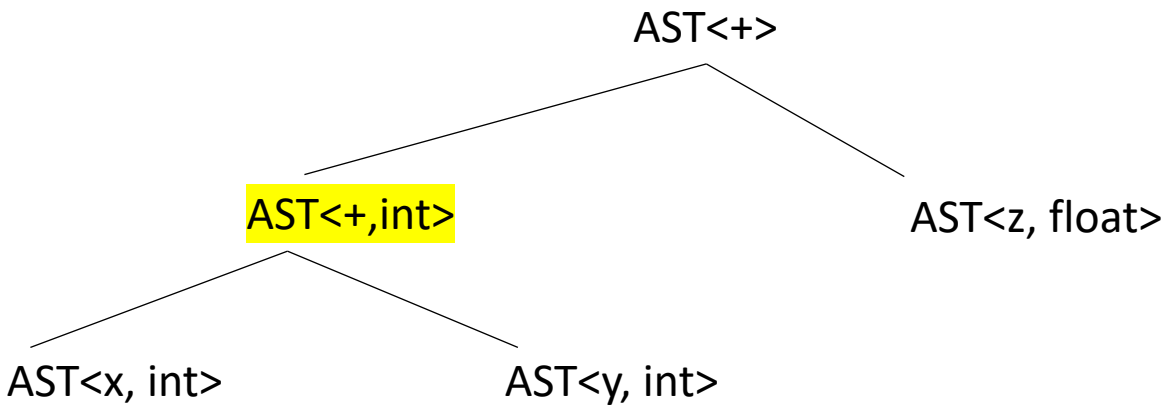
combination rules for subtraction:

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

Type inference on an AST

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

How do we get the type for this one?



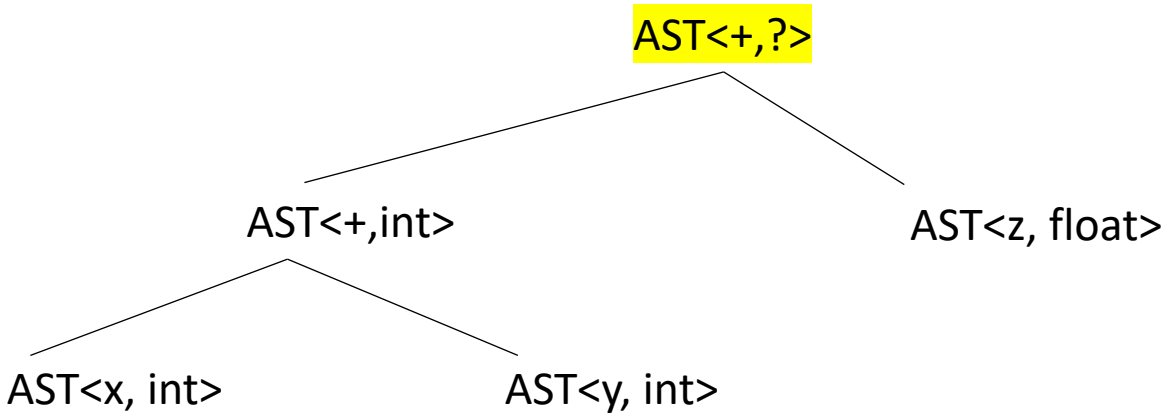
inference rules for subtraction:

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

Type inference on an AST

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

How do we get the type for this one?



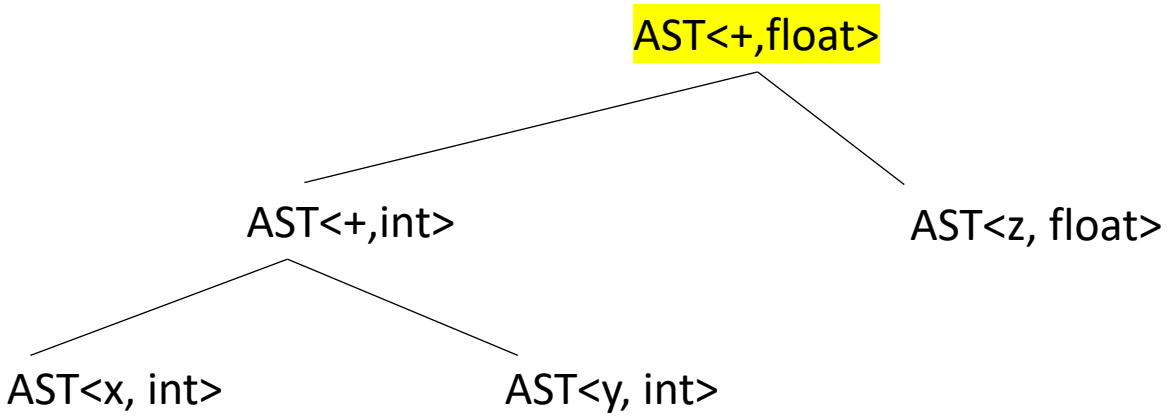
inference rules for subtraction:

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

Type inference on an AST

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

How do we get the type for this one?



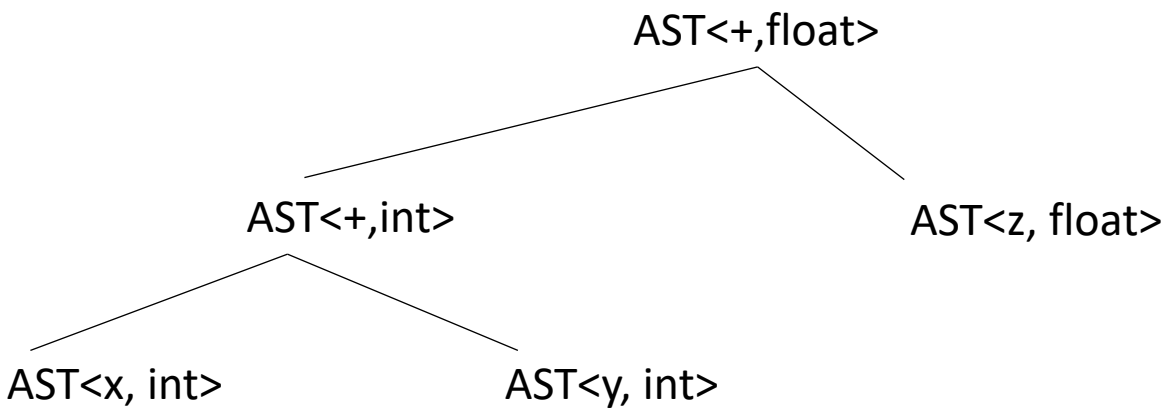
inference rules for subtraction:

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

Type inference on an AST

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

How do we get the type for this one?



inference rules for subtraction:

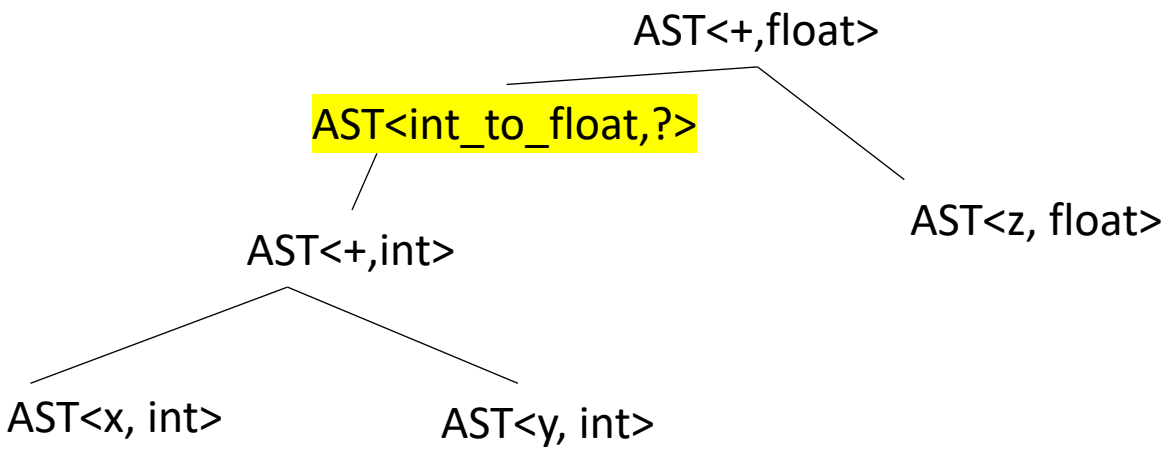
first	second	result
`int`	`int`	`int`
`int`	`float`	`float`
`float`	`int`	`float`
`float`	`float`	`float`

what else?

Type inference on an AST

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

How do we get the type for this one?



inference rules for subtraction:

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

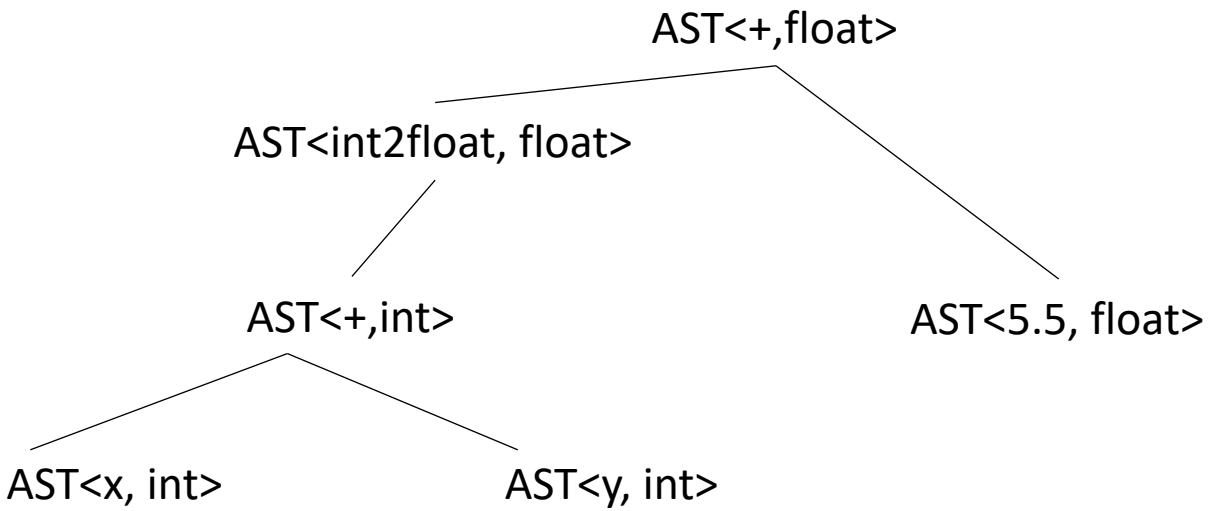
what else? need to convert the int to a float

Linearizing an AST

Converting AST into Class-IR

```
int x;
int y;
float w;
w = x + y + 5.5
```

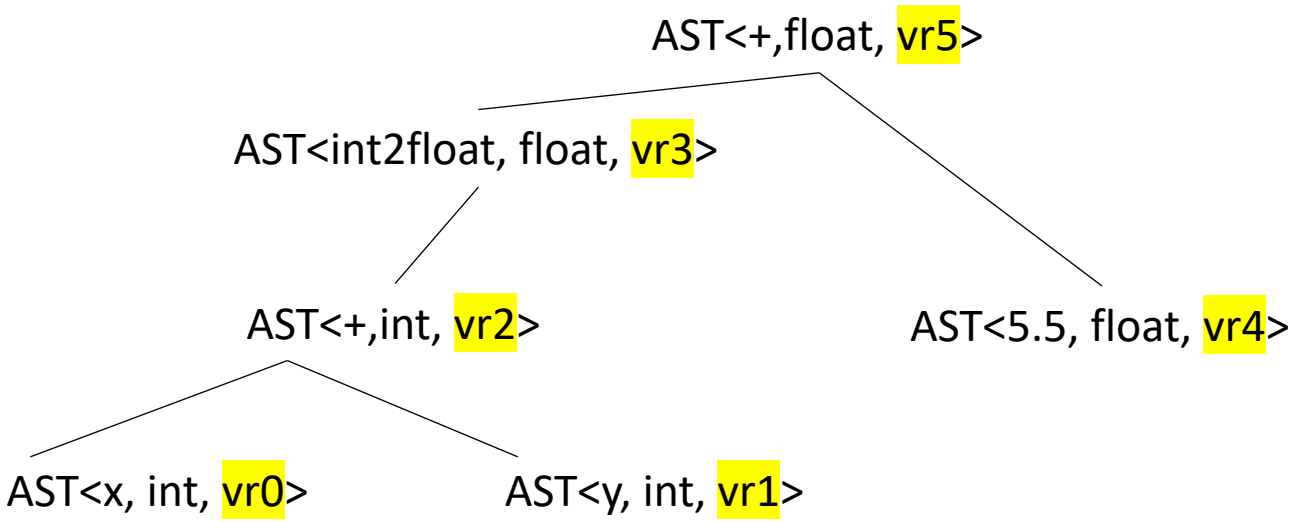
After type inference



Converting AST into Class-IR

```
int x;
int y;
float w;
w = x + y + 5.5
```

After type inference

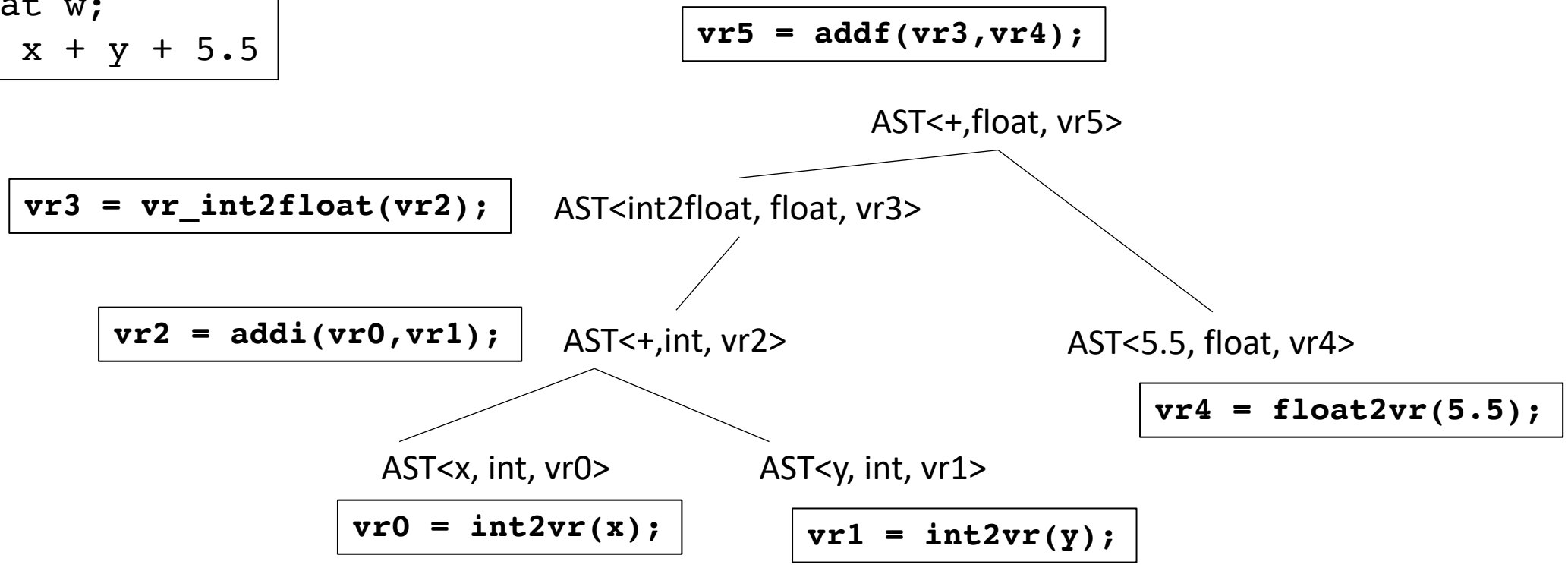


We will start by adding a new member to each AST node:

A virtual register

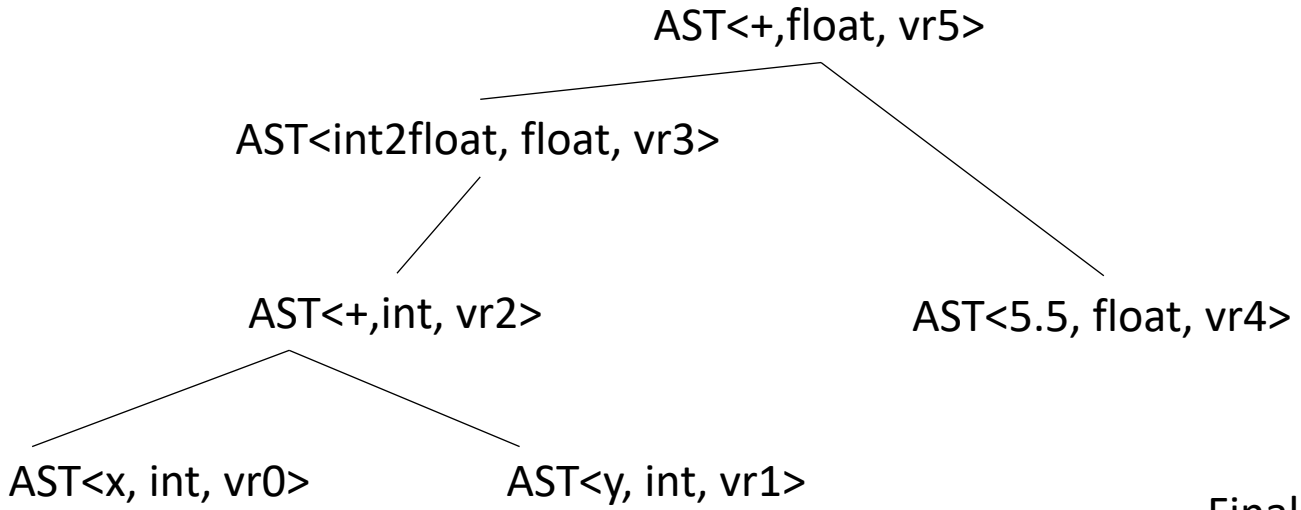
Each node needs a distinct virtual register

```
int x;
int y;
float w;
w = x + y + 5.5
```



What now?
We can create a 3 address program doing a post-order traversal

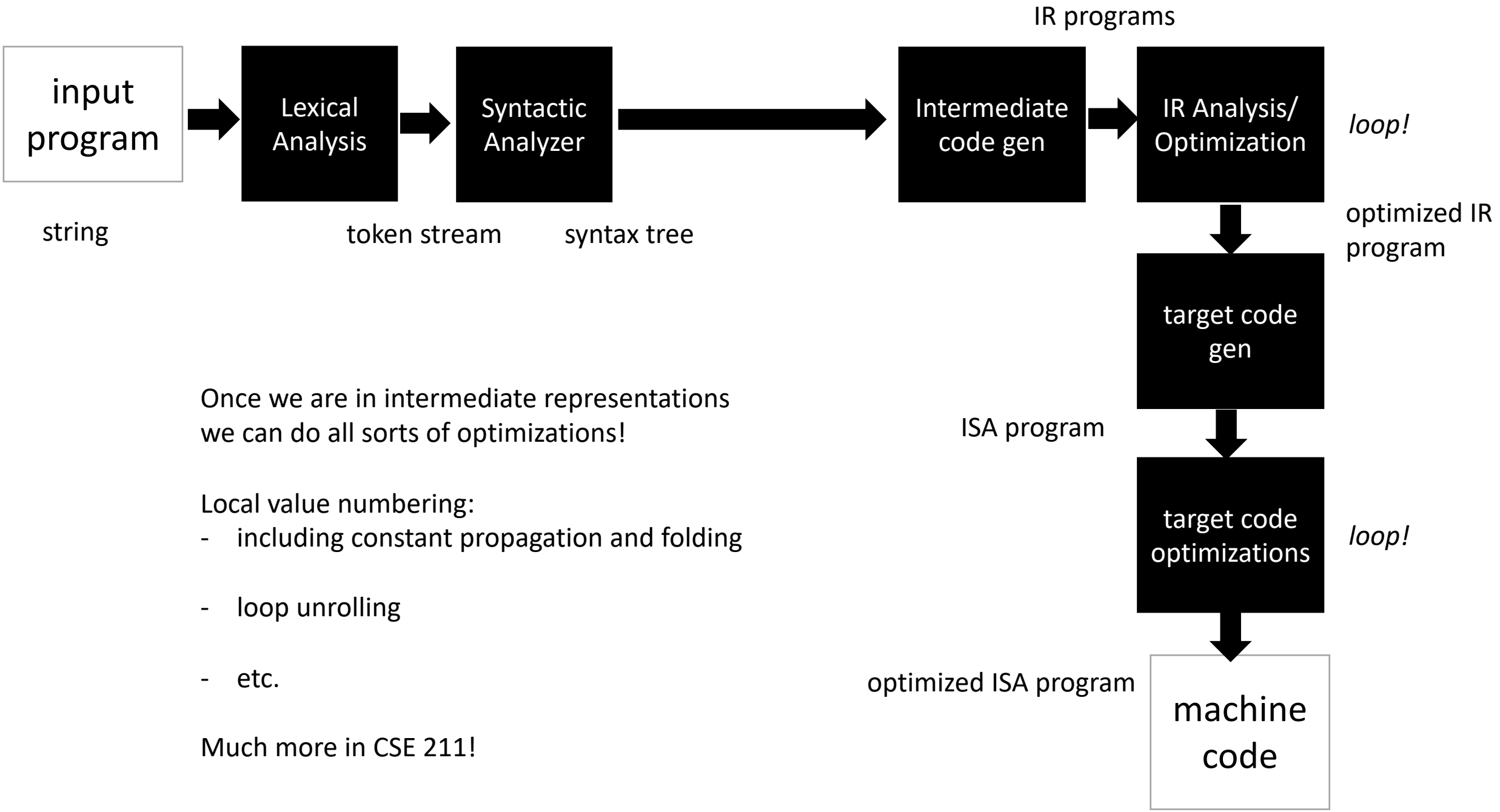
```
int x;
int y;
float w;
w = x + y + 5.5
```



We can create a 3 address program doing a post-order traversal

Final program

```
vr0 = int2vr(x);
vr1 = int2vr(y);
vr2 = addi(vr0,vr1);
vr3 = vr_int2float(vr2);
vr4 = float2vr(5.5);
vr5 = addf(vr3,vr4);
```

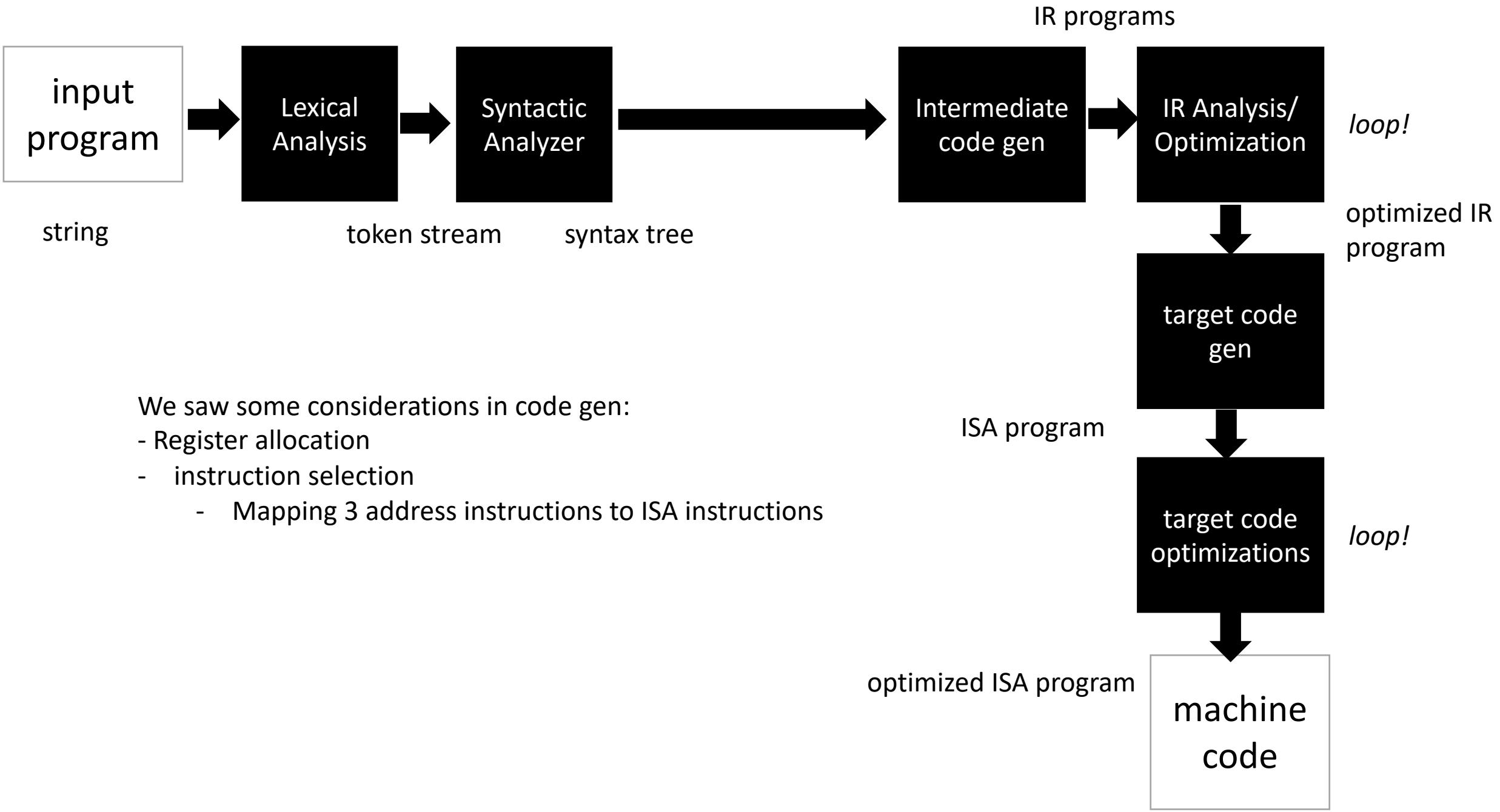


Once we are in intermediate representations we can do all sorts of optimizations!

Local value numbering:

- including constant propagation and folding
- loop unrolling
- etc.

Much more in CSE 211!



We saw some considerations in code gen:

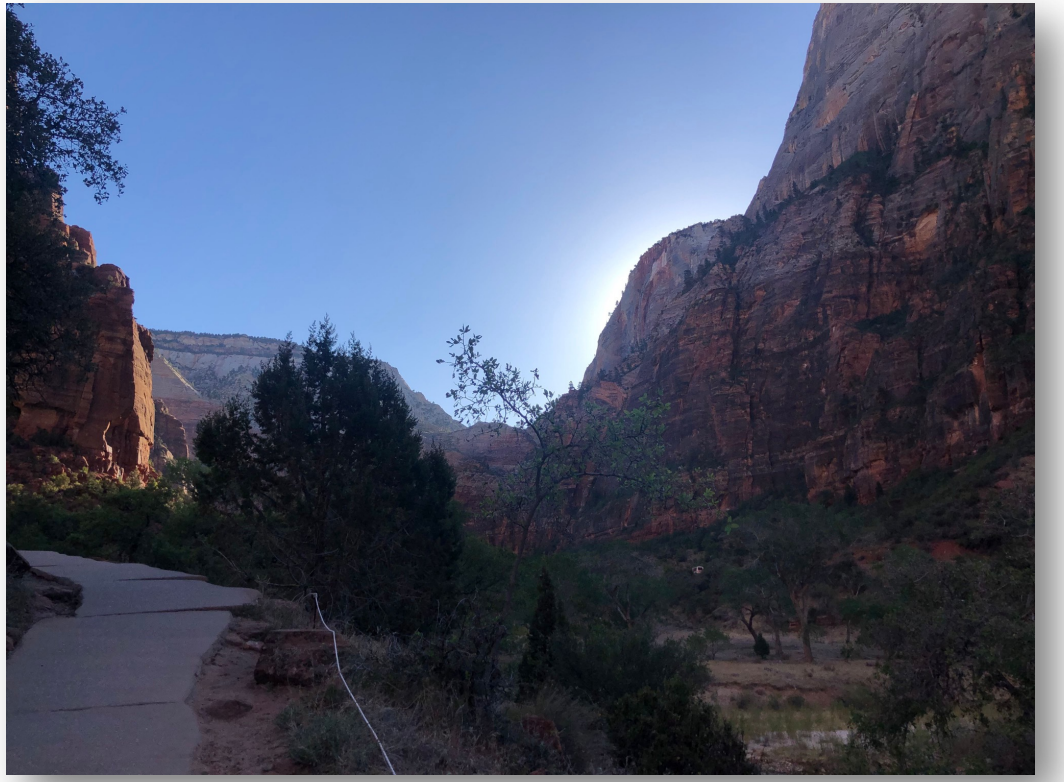
- Register allocation
- instruction selection
  - Mapping 3 address instructions to ISA instructions

Last day of class!

- I hope after the final you take some time to reflect

Taking a class is like going on a long hike

Scanners



Scanners

AST and type checking

The culmination of your homeworks is quite big! A parser and IR generator for a non trivial subset of C!

Take some time in the summer to enjoy the view!

Thank you!

- This is still a new version of the class and I know there were some issues with the assignments. Thanks for your patience and working with us!
- Even if you don’t work on compilers in your career, understanding them will help you write better code and understand programming languages in a deeper way
  - And I hope you found things interesting regardless!
- Hope to keep in touch!
- Let us know if there are any issues with grades, which should be coming out ASAP