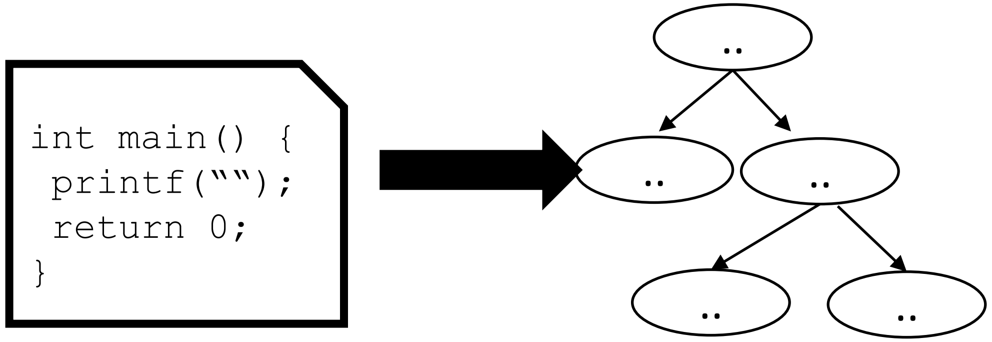
CSE110A: Compilers

Topics:

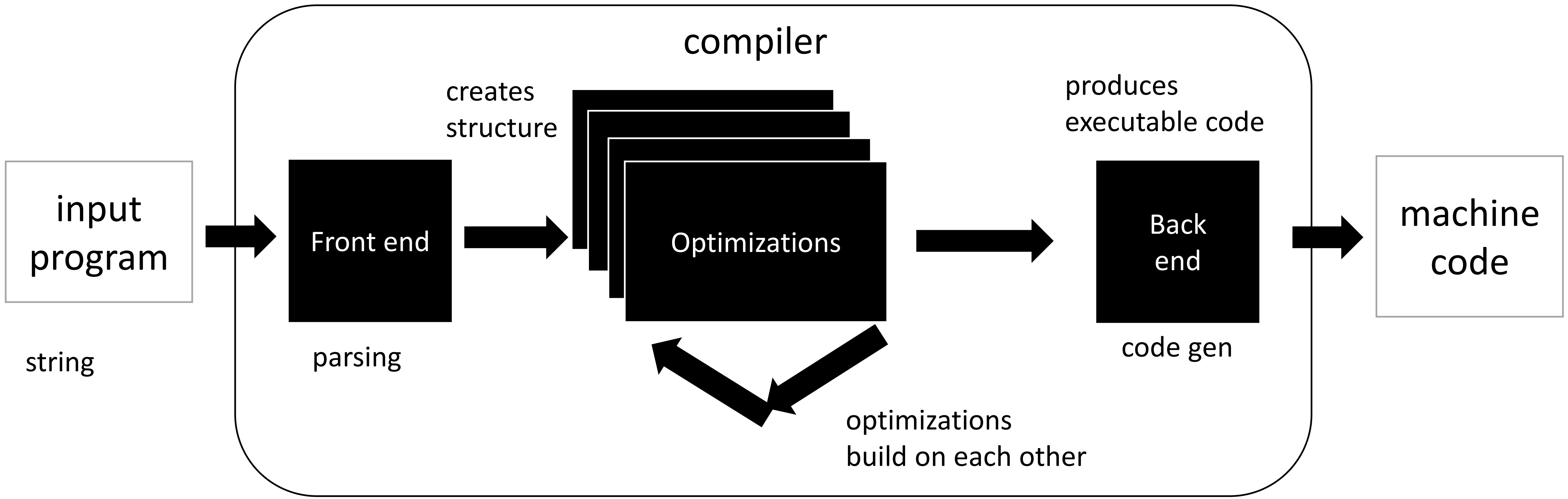
- *Starting Module 2: Parsing*
  - *Introduction*
  - *Production Rules*
  - *Derivations and Parse Trees*
  - *A Simple Expression Grammar*



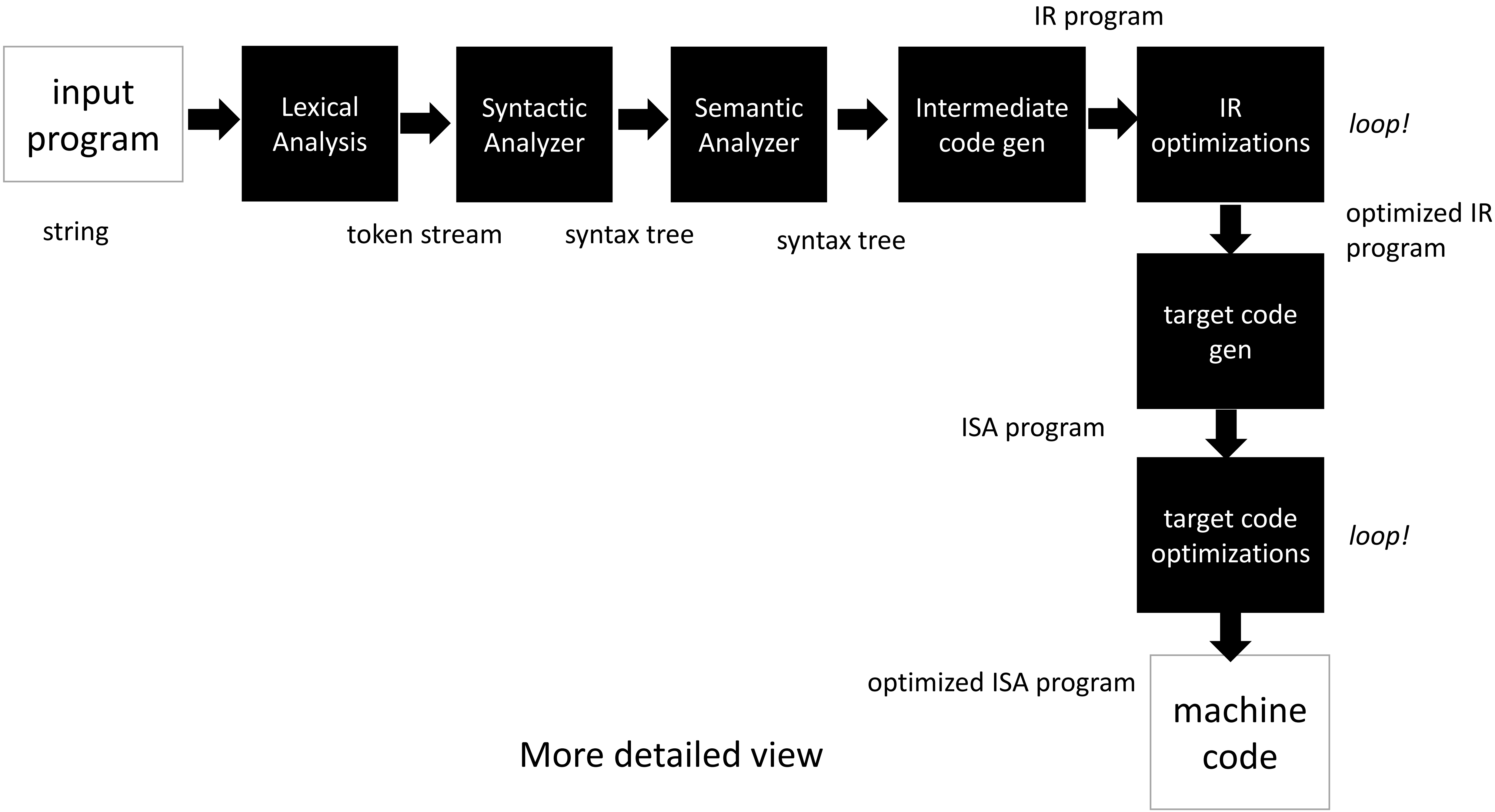
Module 2: CFG and Parsing

- Parsing:
  - Often times scanning is also included in parsing
  - Specifically this module is about “Syntactic Analysis”

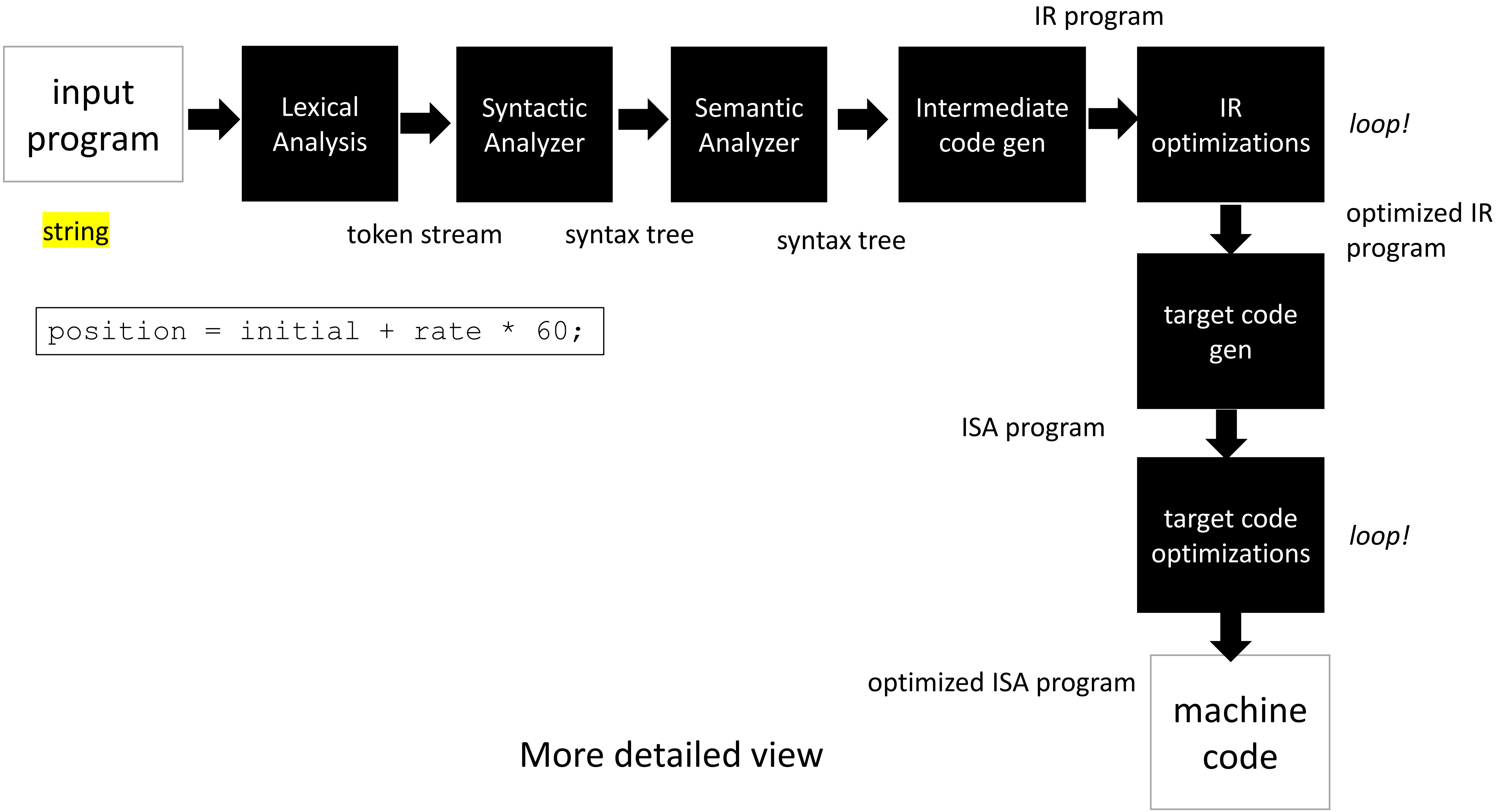
Compiler Architecture



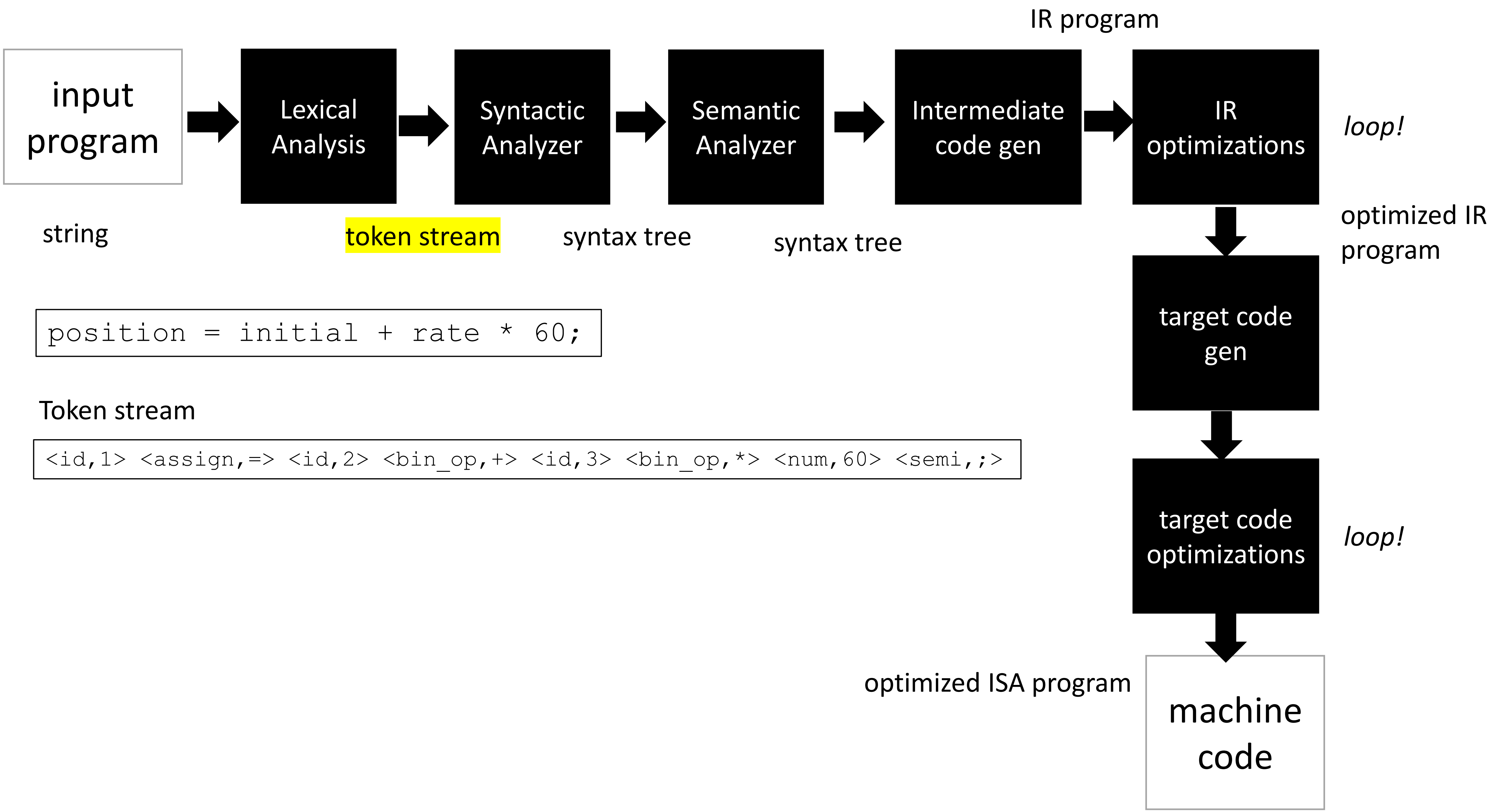
Still working in the front end



More detailed view



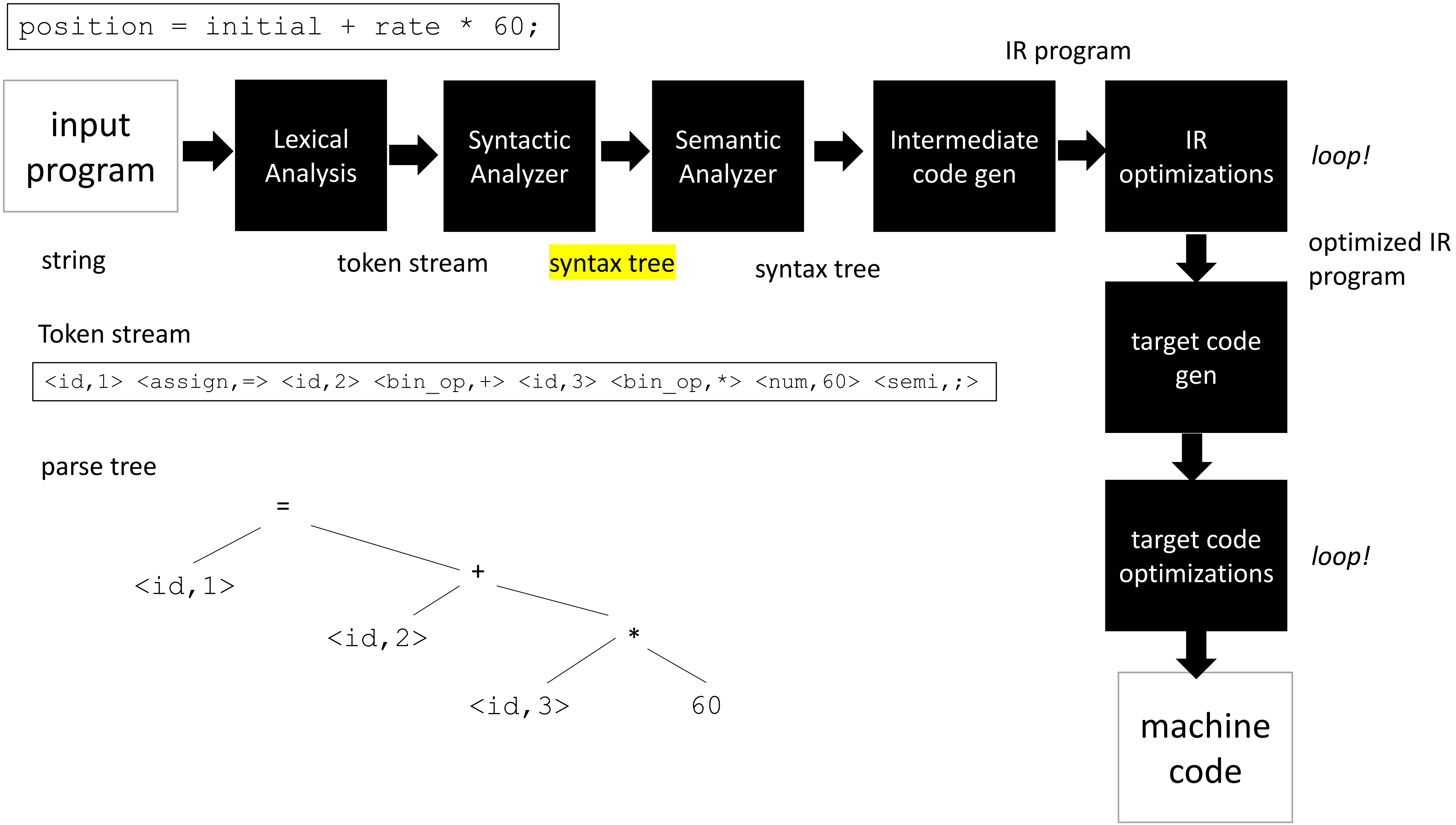
More detailed view




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

Token stream

```
<id,1> <assign,=> <id,2> <bin_op,+> <id,3> <bin_op,*> <num,60> <semi,;>
```


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

Syntactic Analysis

- Lexical Analysis turns a string into a stream of tokens
- Syntactic Analysis determines if the tokens fit into the syntactic structure of the language
- In our natural language example, it describes the structure of sentences

Syntactic Analysis

- Natural language example

The dog ran across the park

ARTICLE NOUN VERB PREPOSITION ARTICLE NOUN

What are valid sentences?

ARTICLE NOUN VERB PREPOSITION ARTICLE NOUN

ARTICLE ADJECTIVE NOUN VERB

Now we check
if stream of lexemes fits
a sentence

How do we express a valid sentence?

- List of tokens:
  - **`ARTICLE NOUN VERB PREPOSITION ARTICLE NOUN`**

- Pros? Cons?

How do we express a valid sentence?

- List of tokens:
  - `ARTICLE NOUN VERB PREPOSITION ARTICLE NOUN`
- Pros? Cons?
  - Simple, but probably too simple

How do we express a valid sentence?

- Several lists of tokens
  - `ARTICLE NOUN VERB PREPOSITION ARTICLE NOUN`
  - `ARTICLE NOUN VERB`
  - `ARTICLE ADJECTIVE NOUN VERB`
  - `ARTICLE ADJECTIVE ADJECTIVE NOUN VERB`

- Pros? Cons?

How do we express a valid sentence?

- Several lists of tokens
  - `ARTICLE NOUN VERB PREPOSITION ARTICLE NOUN`
  - `ARTICLE NOUN VERB`
  - `ARTICLE ADJECTIVE NOUN VERB`
  - `ARTICLE ADJECTIVE ADJECTIVE NOUN VERB`

- Pros? Cons?
  - Potentially infinite choices

How do we express a valid sentence?

- Regular expressions over tokens:
  - `ARTICLE ADJECTIVE* NOUN VERB`
- Pros? Cons?

How do we express a valid sentence?

- Regular expressions over tokens:
  - `ARTICLE ADJECTIVE* NOUN VERB`

- Pros? Cons?
  - Regular expressions worked really well for tokens
  - Provides decent expressivity
  - But what might go wrong?

Mathematical expressions

- tokens:
  - `NUM  = “[0-9]+”`
  - `PLUS = “\+”`
  - `MULT = “\*”`

- Can we describe expressions?

Mathematical expressions

```
NUM ((PLUS | MULT) NUM)*
```

```
5
```

```
5 + 6
```

```
5 + 6 * 3
```

Mathematical expressions

```
NUM ((PLUS | MULT) NUM)*
```

```
5
```

```
5 + 6
```

```
5 + 6 * 3
```

But what does this one mean? What if we want different precedence?

Mathematical expressions

```
NUM ((PLUS | MULT) NUM)*
```

```
5
```

```
5 + 6
```

```
5 + 6 * 3
```

But what does this one mean? What if we want different precedence?

```
(5 + 6) * 3
```

Can we do this one?

Mathematical expressions

- tokens:
  - `NUM  = "[0-9]+"`
  - `PLUS = "\+"`
  - `MULT = "\*"`
  - `OPAR = "\("`
  - `CPAR = "\)"`

Mathematical expressions

```
OPAR? NUM ((PLUS | MULT) OPAR? NUM CPAR?)*
```

Add parenthesis tokens

```
5
```

```
5 + 6
```

```
5 + 6 * 3
```

But what does this one mean? What if we want different precedence?

```
(5 + 6) * 3
```

Can we do this one?

Mathematical expressions

```
OPAR? NUM ((PLUS | MULT) OPAR? NUM CPAR?)*
```

Seems like it works! But what is the issue?

Mathematical expressions

```
OPAR? NUM ((PLUS | MULT) OPAR? NUM CPAR?)*
```

Seems like it works! But what is the issue?

```
(5 + 6 * 3
```

What about this one?

Mathematical expressions

```
OPAR? NUM ((PLUS | MULT) OPAR? NUM CPAR?)*
```

Seems like it works! But what is the issue?

```
(5 + 6 * 3
```

What about this one?

()s are a key part of syntax. They are import for the structure we want to create and we need to reliably detect strings that are not syntactically valid!

Context Free Grammars: A new class of languages

- Regular expressions CANNOT match
  - (),
  - {},
  - HTML start/end tags
  - etc.

- We will use ***context free grammars***

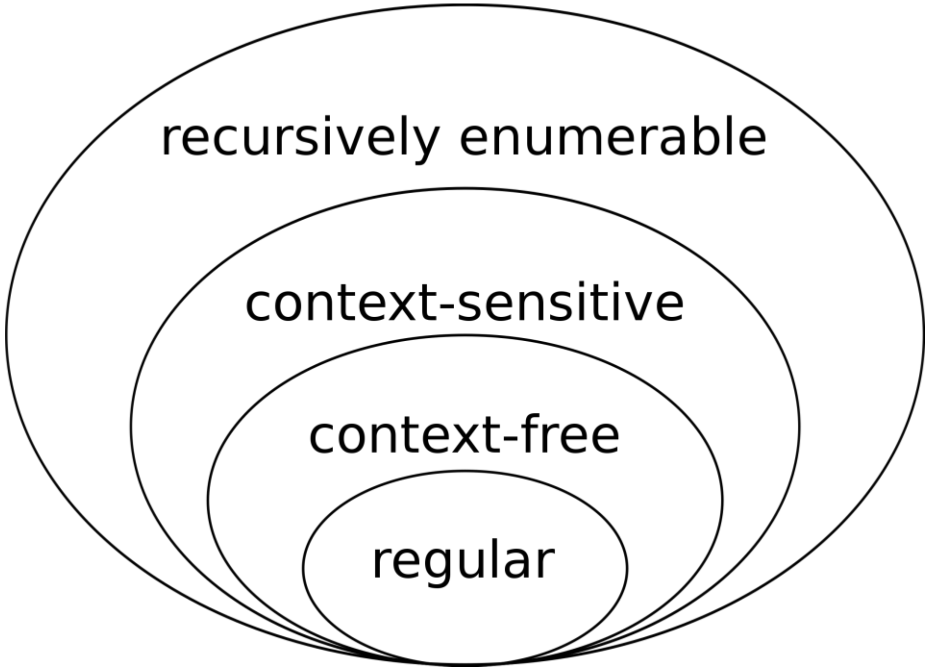
Recall: Language theory

Some theory:

- Given a language *L*, a string *s* is either part of that language or not
  - Integers are a language: “5”, “6”, “-7” is in the language. “abc” is not.

- Languages are grouped into families depending on how “hard” it is to determine if a string is part of that language.

Recall: Language theory



The simplest languages are regular. We used regular expressions for tokens.

- They are fast, even in the general case
- good level of abstraction for tokens

We will now use context-free languages for Syntactic Analysis

- Fast algorithms exist in many cases (not all)

Determining membership can be even inefficient or even undecidable at higher levels (context-sensitive and recursively enumerable)

image source: wikipedia

Context-free languages

We will define similar to regular languages

- In this *class a context-free language is a language that can be recognized by a context-free grammar*

Context-free languages

We will define similar to regular languages

- In this *class a context-free language is a language that can be recognized by a context-free grammar*

- ....

- What is a context-free grammar?

Context-free grammar

We will use *Backus–Naur form* (BNF) form

- non-terminals are language ids. You can have as many as you need.
- each non-terminal maps to one or more production rules.
- one non-terminal is designated as the *start* or *goal* symbol

```
non-terminal-1 ::= production-rule-1
                |   production-rule-2
                |   ...

non-terminal-2 ::= production-rule-1
                |   production-rule-2
                |   ...

....
```

Context-free Grammar (CFG)

We will use *Backus–Naur form* (BNF) form to define a Grammar (G) for a language L(G) by using:

- Production rules (P) contain a sequence of either non-terminals (NT) or terminals (T)
- In our class, terminals (T) will either be string constants or tokens
- Formally a CFG is defined by a 4-tuple as follows:

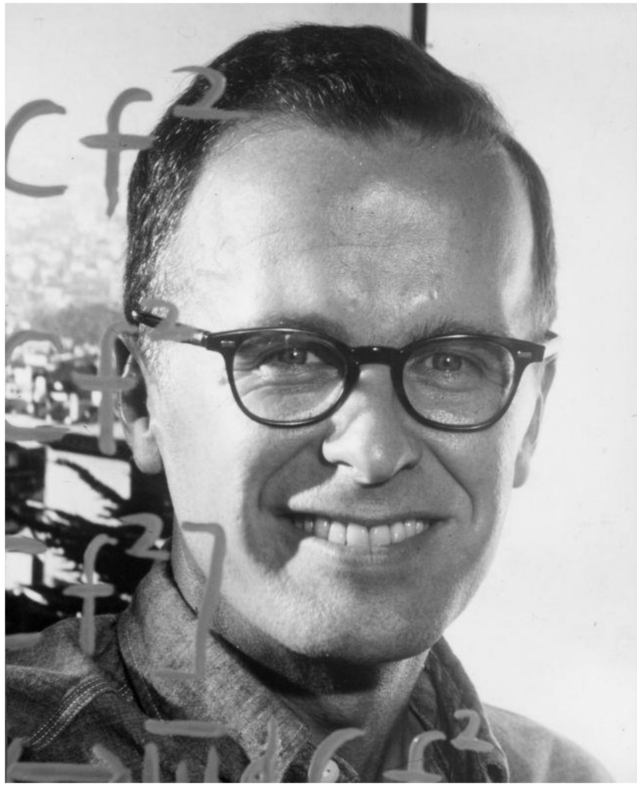
  (NT, T, S, P) where S typically is a Start Symbol preferably not found on the right side of a production. It defines all sentential forms that can be derived in the language.

```
Example:

   S ::= joint_expr
     | simple_expr

   add_expr ::= NUM ‘+’ NUM

   mult_expr ::= NUM ‘*’ NUM

  joint_expr ::= add_expr ‘*’ add_expr

  simple_expr ::= NUM ‘+’ NUM
              |   NUM ‘*’ NUM
```

John Backus (BNF)

```
Examples:
  add_expr ::= NUM '+' NUM
  mult_expr ::= NUM '*' NUM
  joint_expr ::= add_expr '*' add_expr
  simple_expr ::= NUM '+' NUM
              |   NUM '*' NUM
```

John Warner Backus (December 3, 1924 – March 17, 2007) was an American computer scientist. He led the team that invented and implemented FORTRAN, the first widely used high-level programming language, and was the inventor of the Backus–Naur form (BNF), a widely used notation to define syntaxes of formal languages. He later did research into the function-level programming paradigm, presenting his findings in his influential 1977 Turing Award lecture "Can Programming Be Liberated from the von Neumann Style?"[1]



“Much of my work has come from being lazy. I didn’t like writing programs, so I started work on a system to make them easier to write”

John Backus, Inventor of Fortran

Source: Wikipedia
https://www.ibm.com/history/john-backus

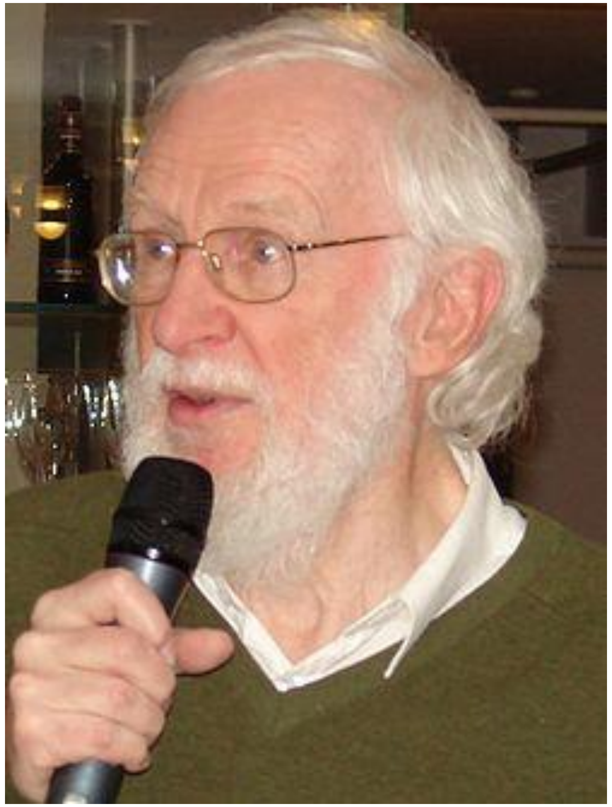
Peter Naur (BNF)

Peter Naur (25 October 1928 – 3 January 2016)[1] was a Danish computer science pioneer and 2005 Turing Award winner. He is best remembered as a contributor, with John Backus, to the Backus–Naur form (BNF) notation used in describing the syntax for most programming languages. He also contributed to creating the language ALGOL 60.

In his book *Computing: A Human Activity* (1992), which is a collection of his contributions to computer science, he rejected the formalist school of programming that views programming as a branch of mathematics. He did not like being associated with the Backus–Naur form (attributed to him by Donald Knuth) and said that he would prefer it to be called the *Backus normal form*.

Source: Wikipedia

```
Examples:
  add_expr ::= NUM ‘+’ NUM
  mult_expr ::= NUM ‘*’ NUM
  joint_expr ::= add_expr ‘*’ add_expr
  simple_expr ::= NUM ‘+’ NUM
              |   NUM ‘*’ NUM
```



Deriving strings

A CFG *G* is said to derive a string *s* if *s* is in the language of *G*

We can show a string *s* belongs to *G* by providing a derivation

```
SheepNoise ::= 'baa' SheepNoise
            |   'baa'
```

Start with a sentinel string: a string containing terminals and non-terminals:

"SheepNoise"

Then pick one of the non-terminals and expand it

Deriving strings

Give each production rule a numeric id

```
1: SheepNoise ::= 'baa' SheepNoise
2:              |   'baa'
```

“baa”

RULE	Sentential Form
start	SheepNoise

“baa baa”

RULE	Sentential Form
start	SheepNoise

Deriving strings

Give each production rule a numeric id

```
1: SheepNoise ::= ‘baa’ SheepNoise
2:              |   ‘baa’
```

“baa”

RULE	Sentential Form
start	SheepNoise
2	“baa”

“baa baa”

RULE	Sentential Form
start	SheepNoise
1	“baa” SheepNoise
2	“baa baa”

Mathematical expressions

- tokens:
  - `NUM  = “[0-9]+”`
  - `OPAR = “\(“`
  - `CPAR = “\)”`

Mathematical expressions

- tokens:
  - `NUM  = “[0-9]+”`
  - `OPAR = “\(“`
  - `CPAR = “\)”`

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

A more complicated example

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

Can we derive the string `(a+b)*c`

RULE	Sentential Form
start	Expr

A more complicated example

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

Can we derive the string `(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

A more complicated example

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

Can we derive the string `(a+b)*c`

We can visualize this as a tree:

Expr

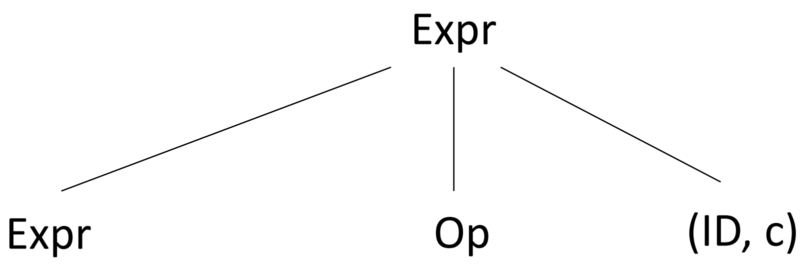
RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID

A more complicated example

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

Can we derive the string `(a+b)*c`

We can visualize this as a tree:



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

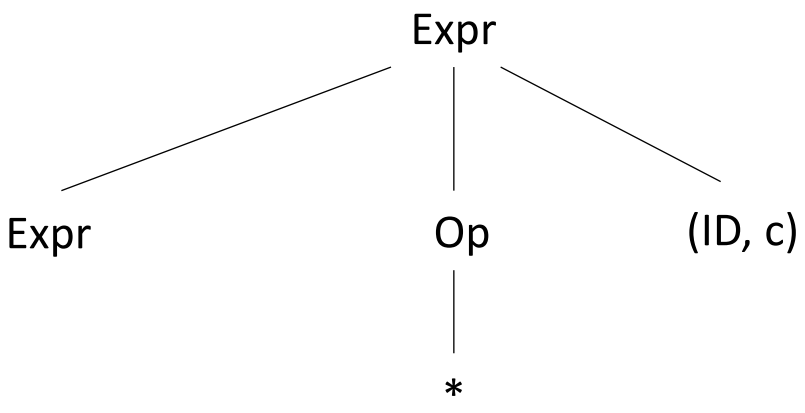
A more complicated example

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

Can we derive the string `(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:



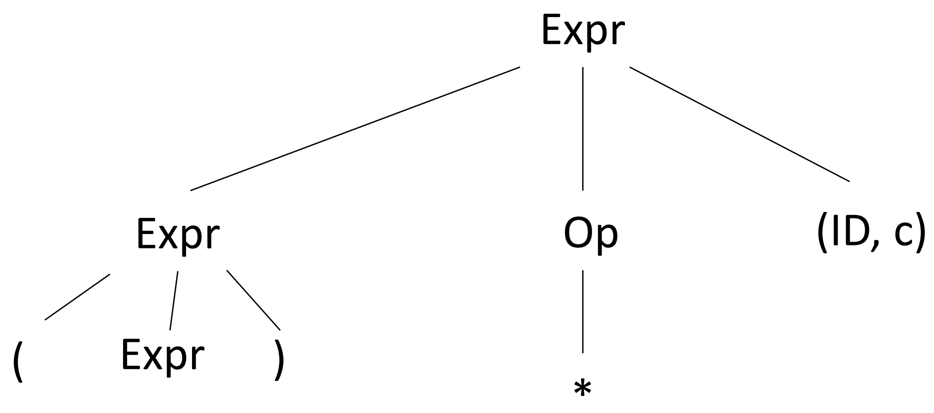
A more complicated example

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

Can we derive the string `(a+b)*c`

We can visualize this as a tree:

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



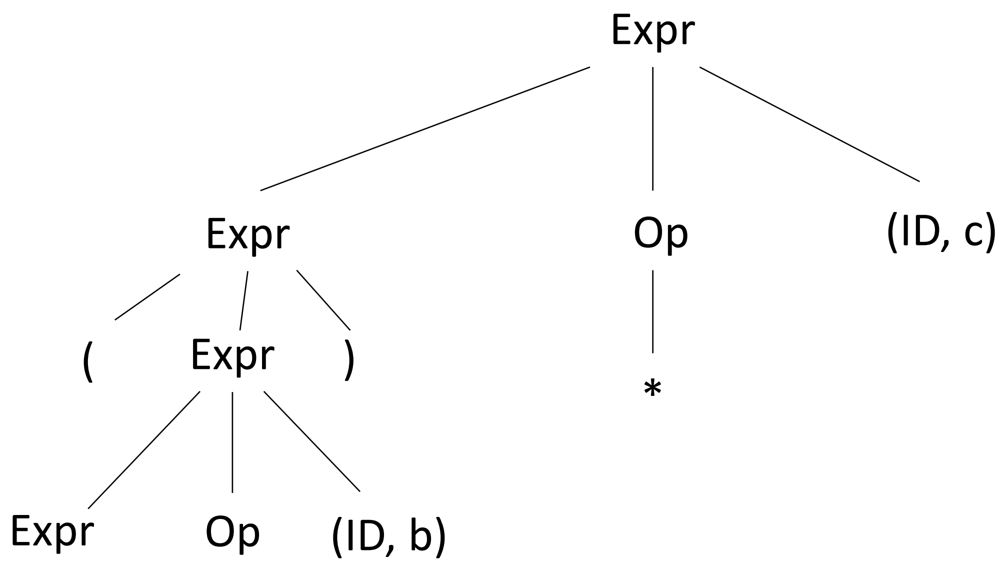
A more complicated example

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

Can we derive the string `(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:



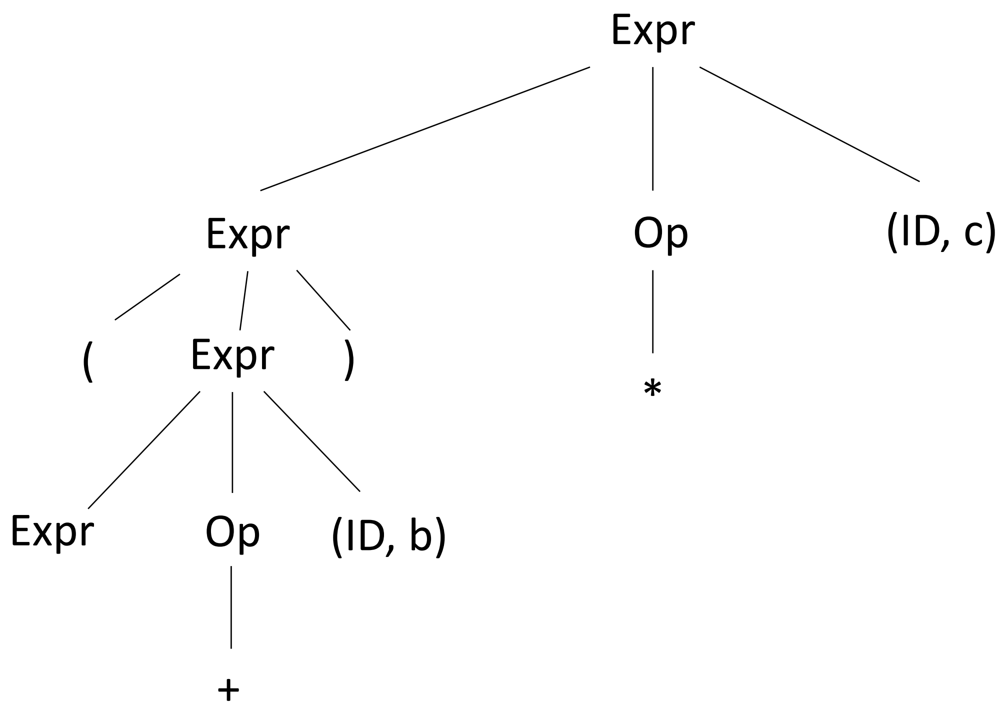
A more complicated example

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

Can we derive the string `(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:



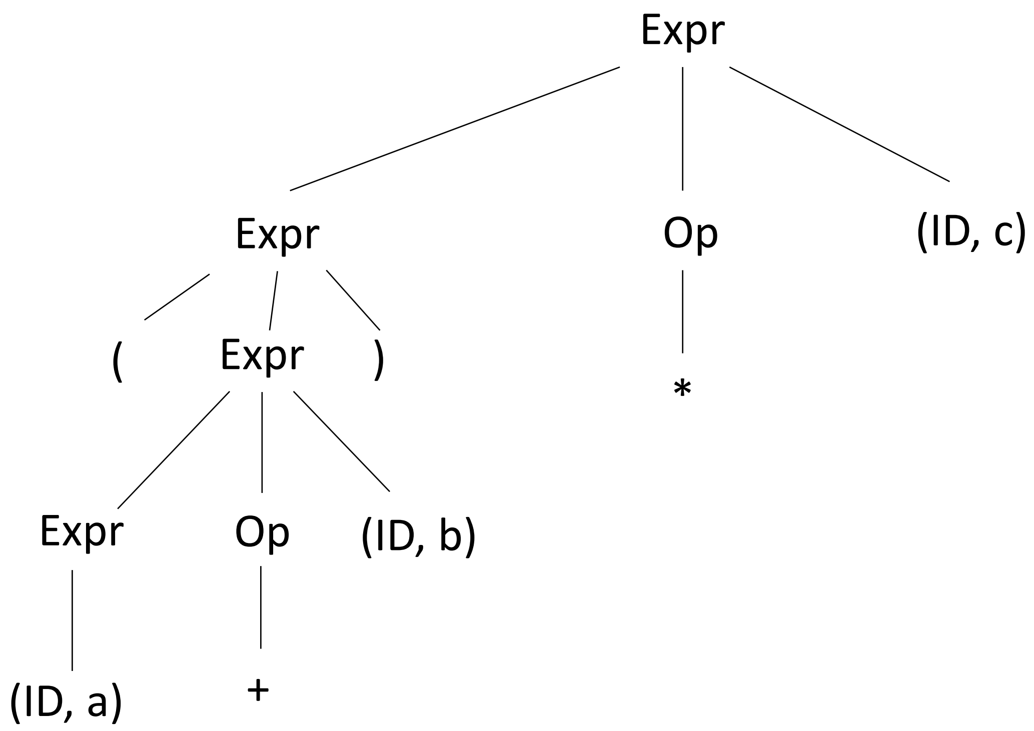
A more complicated example

```
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:



A more complicated example

```
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

RULE	Sentential Form
start	Expr

A more complicated example

```
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

right derivation

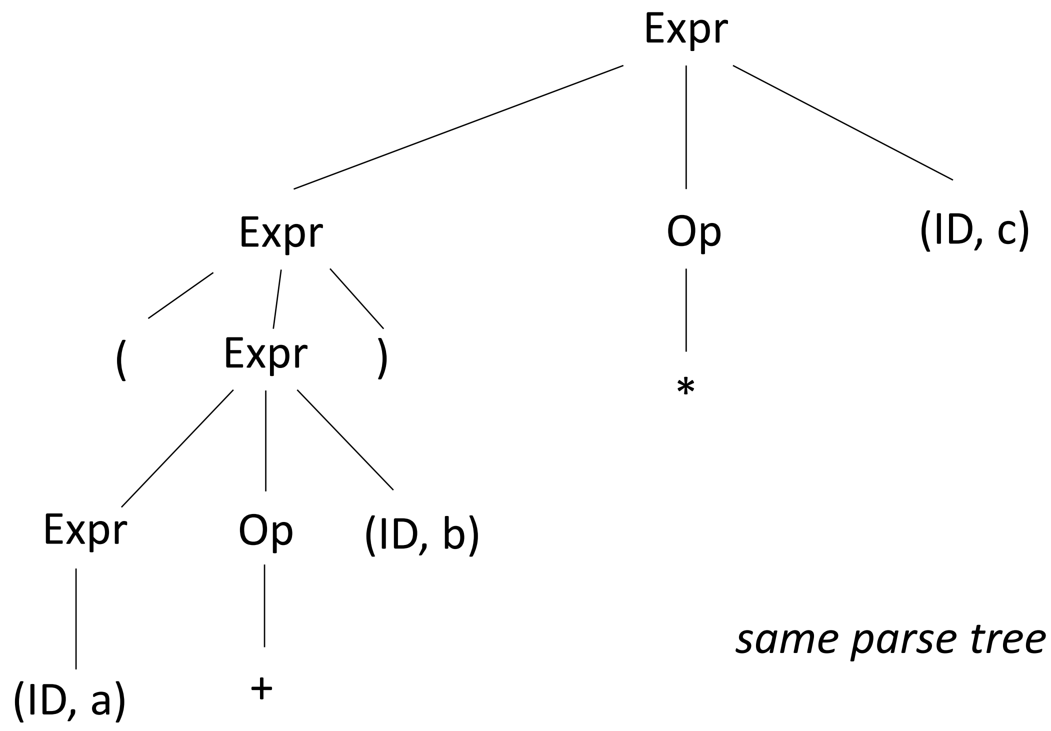
RULE	Sentential Form
start	Expr
2	Expr Op ID
1	(Expr) Op ID
2	(Expr Op ID) Op ID
3	(ID Op ID) Op ID
4	(ID + ID) Op ID
5	(ID + ID) + ID

left derivation

A more complicated example

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

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



same parse tree

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

left derivation

Ambiguous grammars

- What happens when different derivations have different parse trees?

```
1: Statement ::= ″if″ Expr “then” Statement “else” Statement
2:           |   ″if″ Expr “then” Statement
3:           |    Assignment
4:           |    ....
```

can we derive this string?

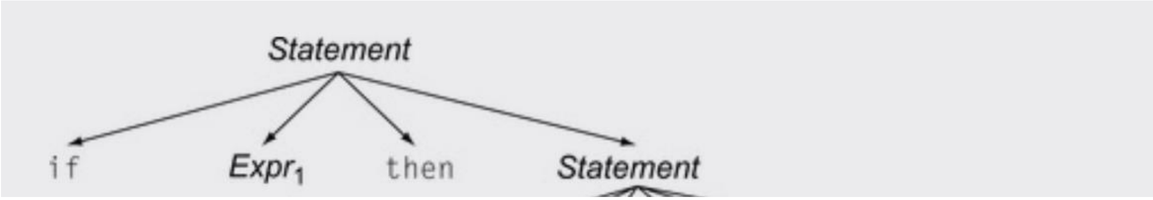
if *Expr1* **then** **if** *Expr2* **then** *Assignment1* **else** *Assignment2*

Next few figures taken from the EAC book (Chapter 3.1)

Ambiguous grammars

```
1: Statement ::= "if" Expr "then" Statement "else" Statement
2:           |    "if" Expr "then" Statement
3:           |     Assignment
4:           |     ....
```

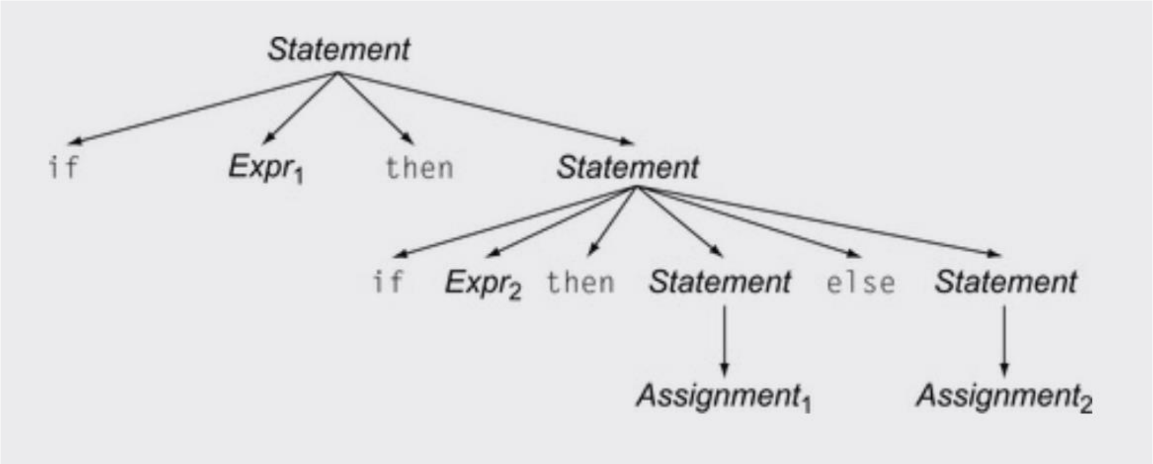
if *$Expr_1$* **then** **if** *$Expr_2$* **then** *$Assignment_1$* **else** *$Assignment_2$*



Ambiguous grammars

```
1: Statement ::= "if" Expr "then" Statement "else" Statement
2:             |   "if" Expr "then" Statement
3:             |    Assignment
4:             |    ....
```

if $Expr_1$ **then** **if** $Expr_2$ **then** $Assignment_1$ **else** $Assignment_2$

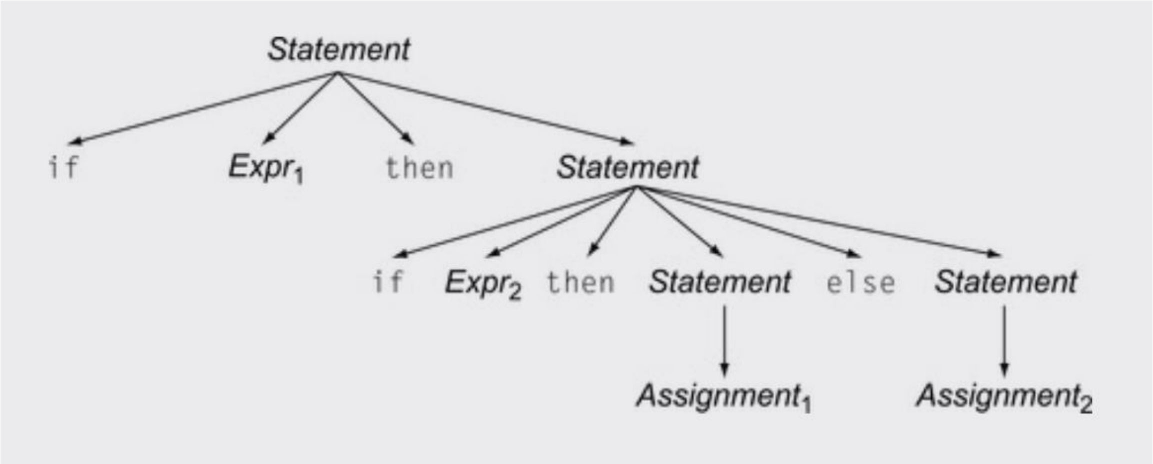


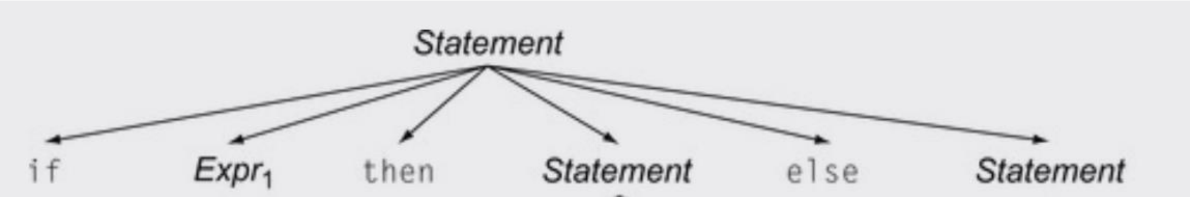
Valid derivation

Ambiguous grammars

```
1: Statement ::= "if" Expr "then" Statement "else" Statement
2:           |   "if" Expr "then" Statement
3:           |    Assignment
4:           |    ....
```

if *Expr₁* **then** **if** *Expr₂* **then** *Assignment₁* **else** *Assignment₂*



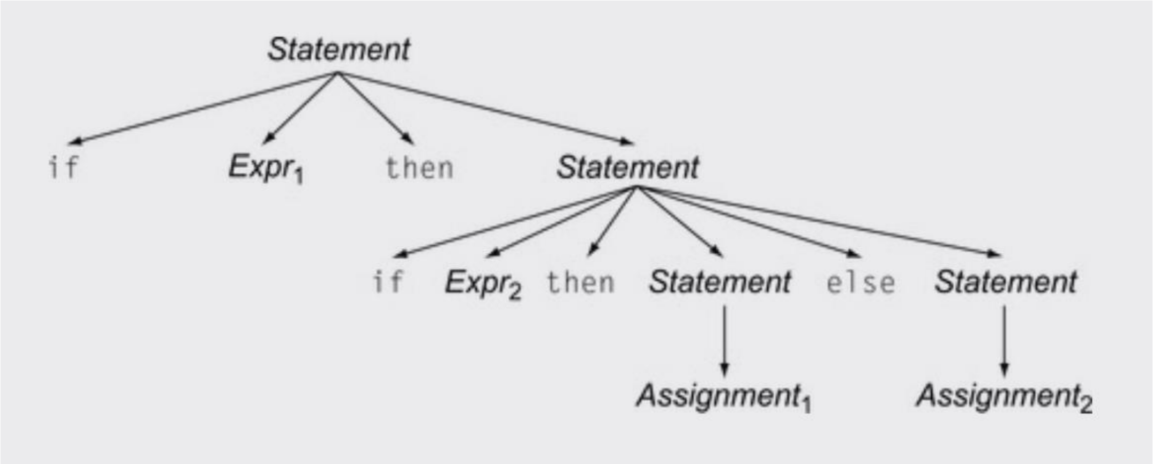


Valid derivation

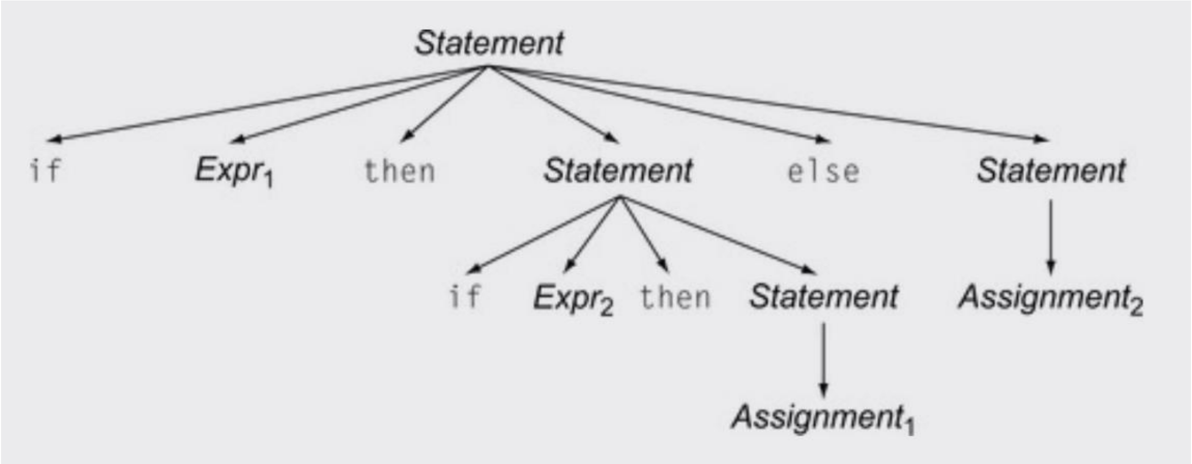
Ambiguous grammars

```
1: Statement ::= "if" Expr "then" Statement "else" Statement
2:           |   "if" Expr "then" Statement
3:           |    Assignment
4:           |    ....
```

if *$Expr_1$* **then** **if** *$Expr_2$* **then** *$Assignment_1$* **else** *$Assignment_2$*



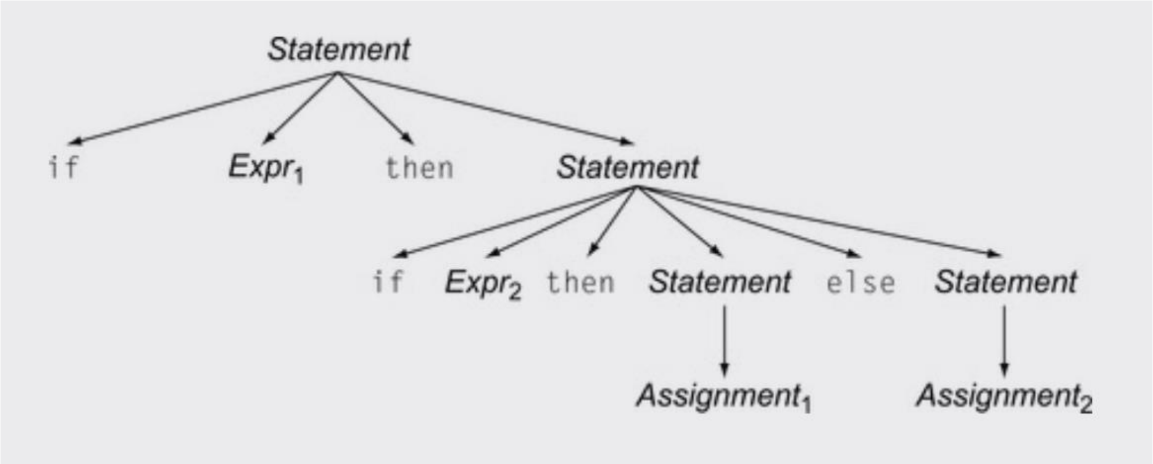
Valid derivation



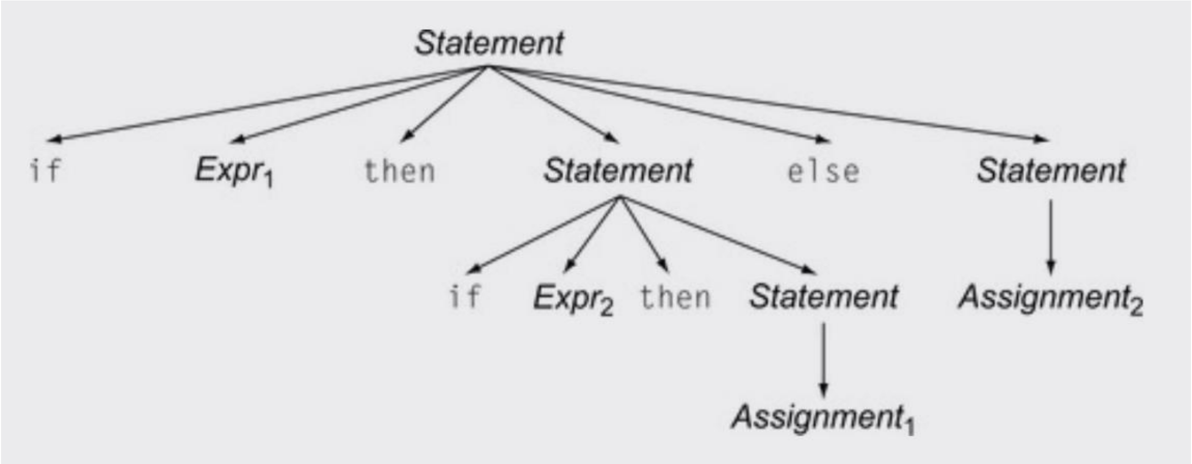
Also a valid derivation

Ambiguous grammars

Is this an issue? Don't we only care if a grammar can derive a string?



Valid derivation



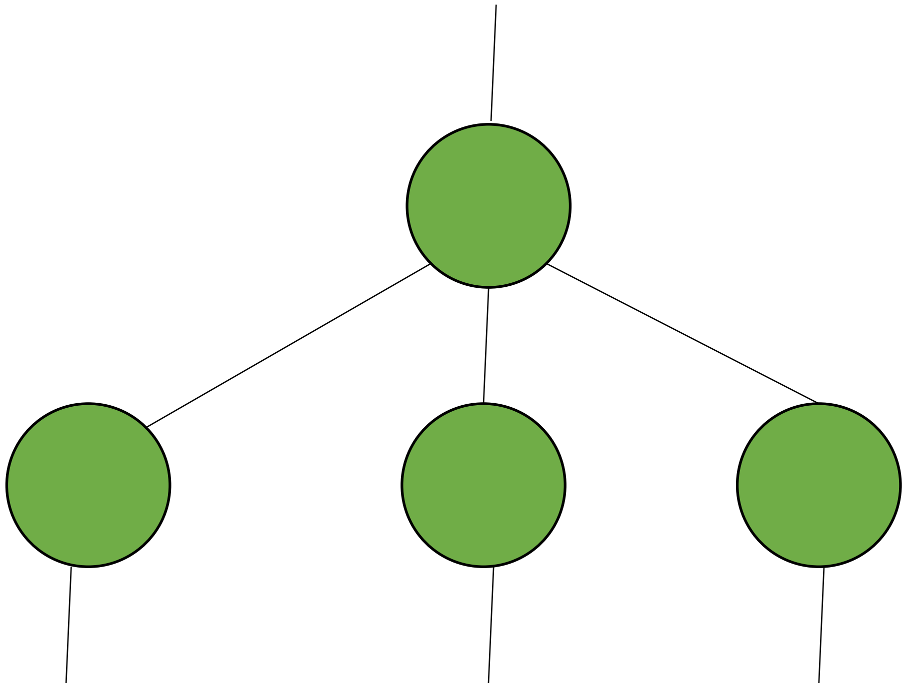
Also a valid derivation

Meaning into structure

- We want to start encoding meaning into the parse structure. We will want as much structure as possible as we continue through making a compiler

- The intended structure is wanting the evaluation of a program to correspond to a post order traversal of the parse tree (also called the natural traversal)

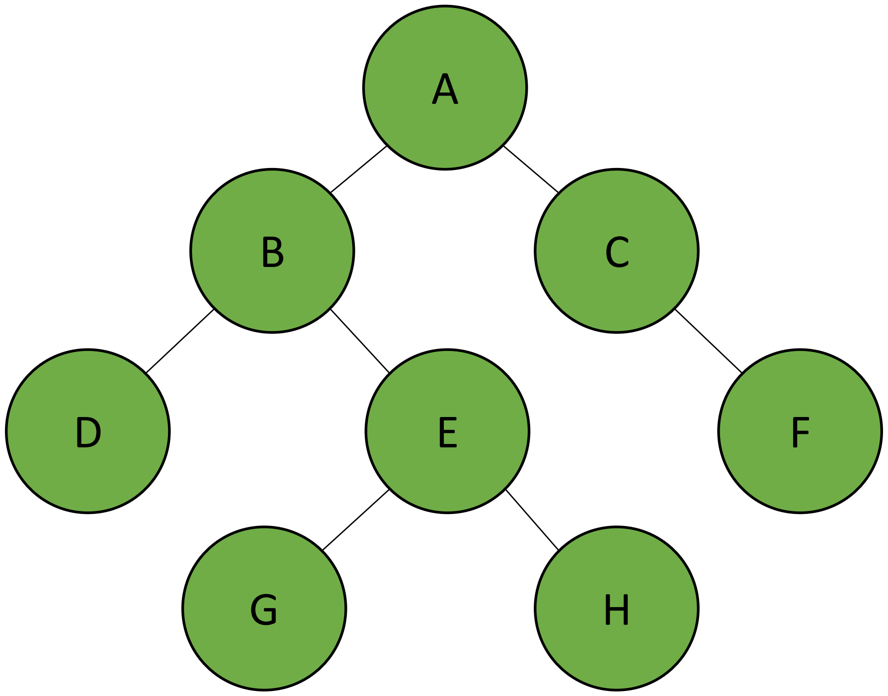
Post order traversal



Visiting nodes for different types of traversals:

pre order?
in order?
post order

Review: Possible Orders of Traversal



Traversal Order	Order Visited	Example Output
pre-order	Top-Root->Left-Child->Right-Child	A B D E G H C F
in-order	Left/Bottom->Its-Root->Right/Bottom	D B G E H A C F
post-order	Left/Bottom->Right/Bottom->Its-Root	D G H E B F C A

Traversals never visit the same node twice.

Pre-order Traversal (Root-Left-Right. Top to Bottom)

Prioritizes visits from top to bottom, and left to right

Useful for serializing and copying trees.

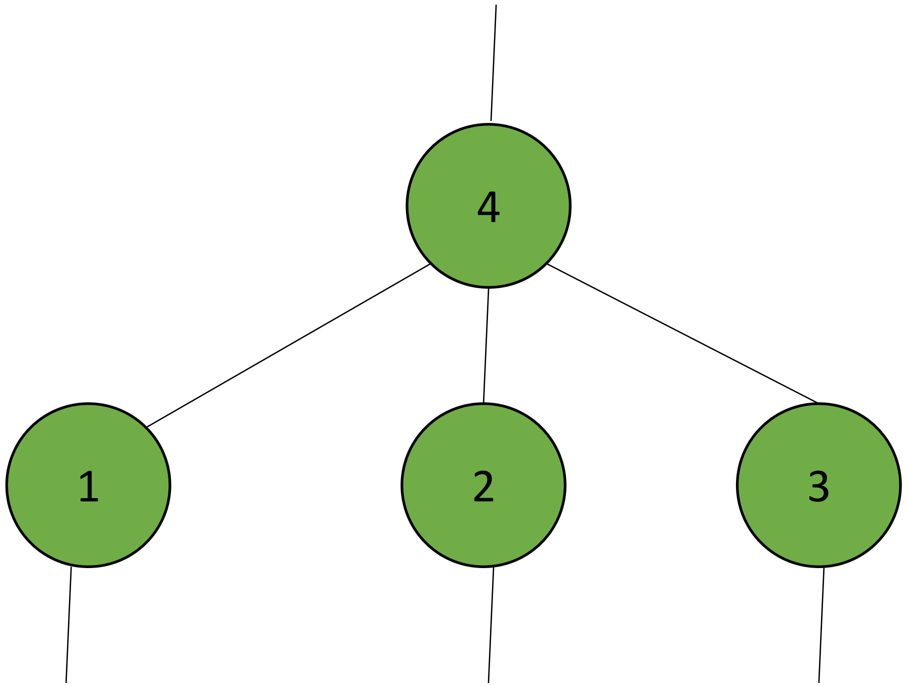
In-order Traversal (Left-Root-Right, Bottom to Top)

Prioritizes visits from bottom left to right visiting parent nodes on the way. Sometimes used for sorting.

Post-order (or natural) Traversal (Left-Right-Root)

Prioritizes traversing subtrees by visiting lowest nodes first, and parent nodes later. Useful for evaluating expression trees (e.g. parsing)

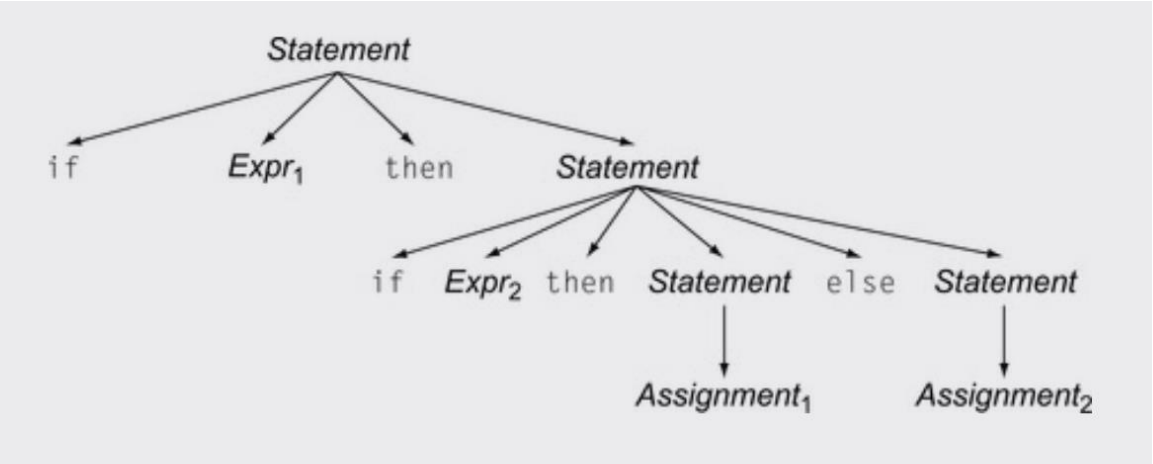
Post Order Traversal



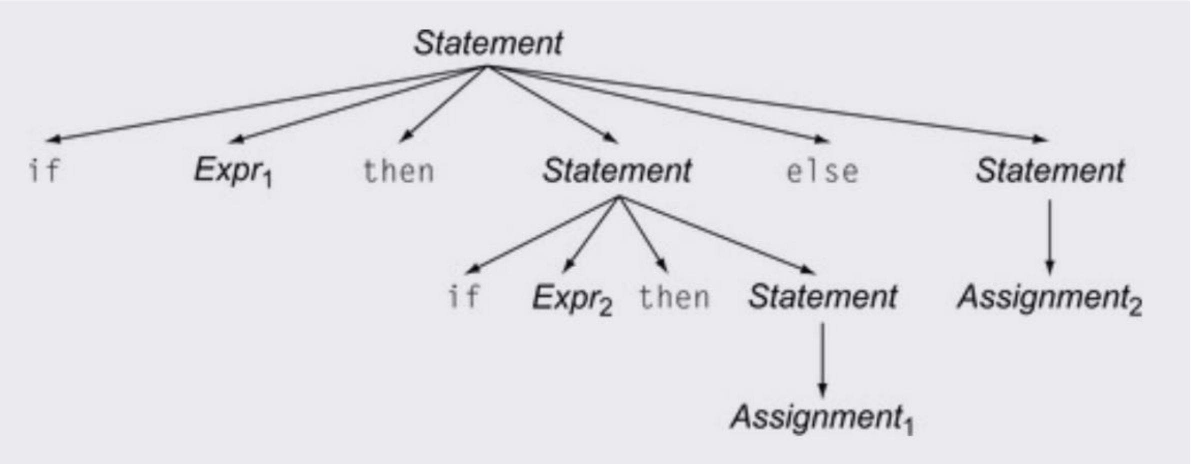
Post Order traversal defines the desired type of structure Considered for parse trees.

Ambiguous grammars

When we encode meaning into structure, these are very different programs



Valid derivation



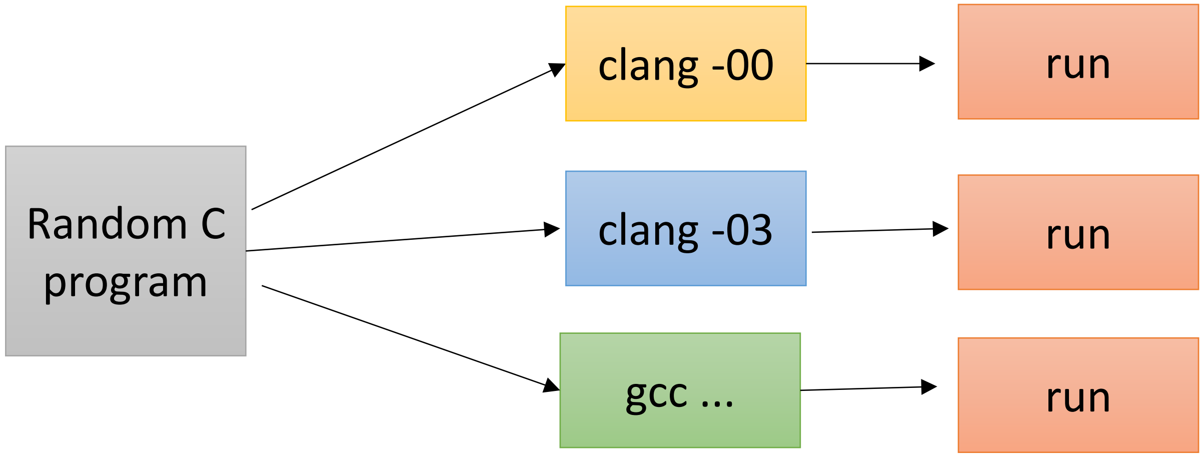
Also a valid derivation

We will study how to eliminate ambiguity

- But let us start with an interesting case study

Case study

- Using a CFG, you can derive random strings in a language
- C-Smith
  - Generates random C programs
  - Used to test compiler correctness

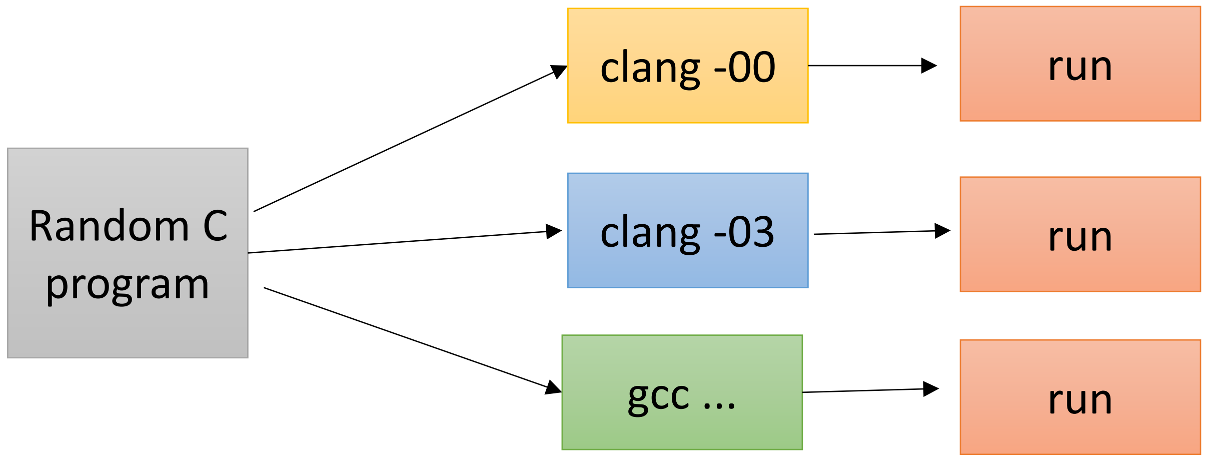


Check outcome. Is it the same? if not, then there is a bug in one of the compilers

Case study

- 400+ compiler bugs found

- Demo



Check outcome. Is it the same? if not, then there is a bug in one of the compilers

Case study

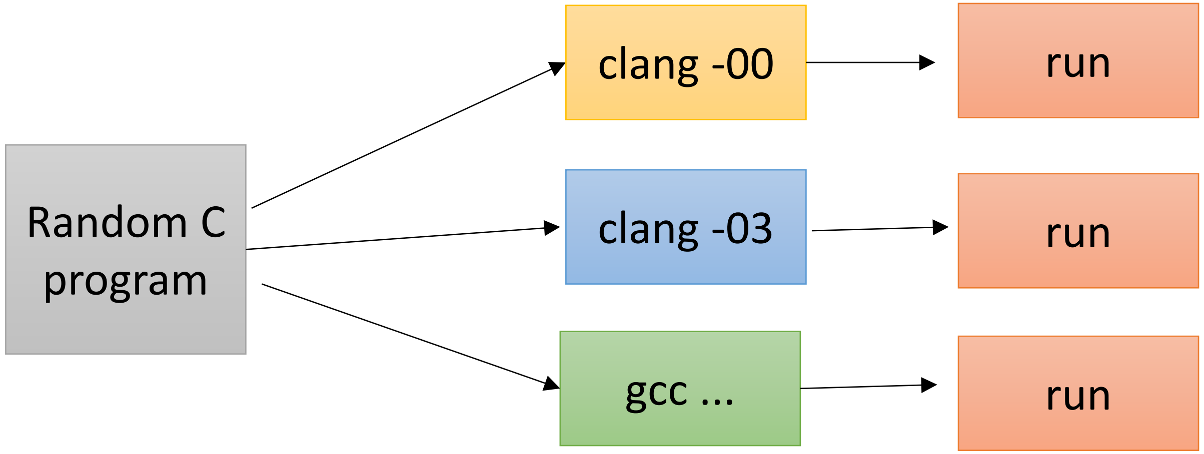
- Big challenge: Undefined behavior

- Even though the program is syntactically valid, the behavior may be undefined

```c
int main() {
    int x;
    printf("%d\n", x);
    return 0;
}
```

Uninitialized variables can return anything!

Use advanced compiler analysis to catch these issues



Check outcome. Is it the same?
if not, then there is a bug in one of the compilers

Next Topic:

- How to remove ambiguity from grammars
    - Precedence
    - Associativity