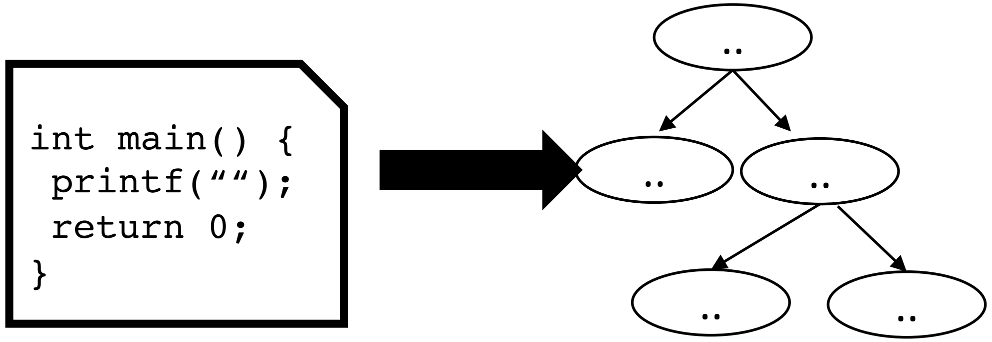
CSE110A: Compilers

April 11, 2022

Topics:

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



Announcements

- HW 1 is out
  - You have 1 week left to do it
- Due April 18 by midnight
- Visit us for office hours
  - Sign up for Piazza
  - And thanks to those who are asking/answering questions

Announcements

- Homework clarifications

- For part 2:

- **HNUM**: a hexidecimal number. Like in C, it should start with a 0x followed by digits, which can include a-f. The characters should be case insensitive.

All characters are case insensitive!

Announcements

- Homework clarifications

- For part 4: You should ***not*** hard code the RegEx: you should generate it given the list of tokens

Announcements

- Homework clarifications

- For part 4: You should ***not*** hard code the RegEx: you should generate it given the list of tokens

- How we will grade:
  - Your tokens will be graded using the our solution scanner importing your tokens
  - We will then put in our own tokens to grade your SOS and NG scanners

Announcements

- Homework clarifications
- General question: How to scan “+++”

Announcements

- Homework clarifications
- General question: How to scan “+++”
- `[(INCR, “++”), (PLUS, “+”)]`

If you aren’t careful, in the SOS or NG scanner you could get:
`[(PLUS, “+”), (PLUS, “+”), (PLUS, “+”)]`

This is not correct!

Quiz

Quiz

Which of the following are token actions NOT great for:

- ○ Changing the value of a token
- ○ Changing the token type
- ○ Splitting a token into multiple tokens
- ○ Keeping track of scanning statistics (e.g. the number of IDs seen)

Examples

Modifying a value

```
def cat_dog(x):
    if x[1] == "Cat":
        return (x[0], "Dog")
    return x
```

Modifying a token type

```
keywords = [("INT", "int"), ("FLOAT", "float"), ("IF", "if")]

def check_keywords(t):
    keyvalues = [x[1] for x in keywords]
    if t[1] in keyvalues:
        lexeme = keywords[keyvalues.index(t[1])]
        return lexeme
    return t
```

Examples

Keeping track of statistics

```
def count_lines(x):
    if x[1] == "\n":
        s.lineno += 1
    return x
```

What other statistics might you want?

Quiz

Which of the following are token actions NOT great for:

- ○ Changing the value of a token
- ○ Changing the token type
- ○ Splitting a token into multiple tokens
- ○ Keeping track of scanning statistics (e.g. the number of IDs seen)

This is really difficult to do with token actions:
token actions take a single lexeme and return a single lexeme

Quiz

All scanner generators have the same interface, which makes it very easy to switch from one generator (e.g. Lex) to another (e.g. PLY)

- ○ True
- ○ False

Scanner generators

- You can assume that all take in Regular expressions
  - Most of the time they have nice optional operators, e.g. [0-9], +, ?

- You can assume that all of them support token actions, but they may be expressed differently.

- You can assume that all of them have a function similar to `token()`
  - In lex it is called `yylex()`

PLY Example

Defining a token with no token action

```
t_PRONOUN = "her|his|their"
```

Defining a token with a token action

```
def t_PRONOUN(t):
    "her|his|their"
    if t.value in ["his", "her"]:
        t.value = "their"
    return t
```

Quiz

Which of the following language features make scanner implementations easier?

- [ ] Regular expression matcher
- [ ] Higher order functions
- [ ] Types
- [ ] Interpreted languages

Quiz

Which of the following language features make scanner implementations easier?

- ☐ Regular expression matcher
- ☐ Higher order functions
- ☐ Types
- ☐ Interpreted languages

Required unless you want to write your own (take CSE211 for an example)

Quiz

Which of the following language features make scanner implementations easier?

- [ ] Regular expression matcher
- [ ] Higher order functions
- [ ] Types
- [ ] Interpreted languages

Great for token actions, custom error functions, etc.

Quiz

Which of the following language features make scanner implementations easier?

- [ ] Regular expression matcher
- [ ] Higher order functions
- [ ] Types
- [ ] Interpreted languages

Great for making sure your token actions are consistent. This is a shortcoming of Python

Quiz

Which of the following language features make scanner implementations easier?

- [ ] Regular expression matcher
- [ ] Higher order functions
- [ ] Types
- [ ] Interpreted languages

Doesn't really matter.
Ocaml is great for compilers (compiled)
Scheme is great for compilers (interpreted)

Quiz

It is the last lecture of Module 1; please let me know any feedback you might have about the module: e.g. what you enjoyed or what you think could be improved.

This is a new class for me and I'm trying to revamp the lectures and homework so your feedback is very useful to me!

Thanks for all your feedback! I have ideas for next time

It sounds like the pacing can be picked up a bit and spend more time on:

- *Python code*
- *Corner cases (e.g. how to deal with regular expressions that share common prefixes)*

Review

- We covered token actions in the quiz.

- Token actions are an optional part of a token definition
  - In our case, you can just send the idy function if you don't need one

- Token actions take in a lexeme and return a possibly refined lexeme
  - KEYWORDS refine IDs
  - Swap cat and dogs

- Token actions can also modify state

Review

- We also looked at using first class functions to implement our own error functions
  - We can throw an exception
  - Or try to recover at a ”synchronization” point (e.g. whitespace, or ;)

```
# Assume a scanner object s with the member istring
# to keep track of the string being scanned
def recover_err():
    v = s.istring[0] # Get first character
    s.istring = s.istring[1:] # Chop the first
    return ("ERROR", v); # Return a special ERROR lexeme


# set the error function in the scanner
s.set_error(recover_err)
```

Review

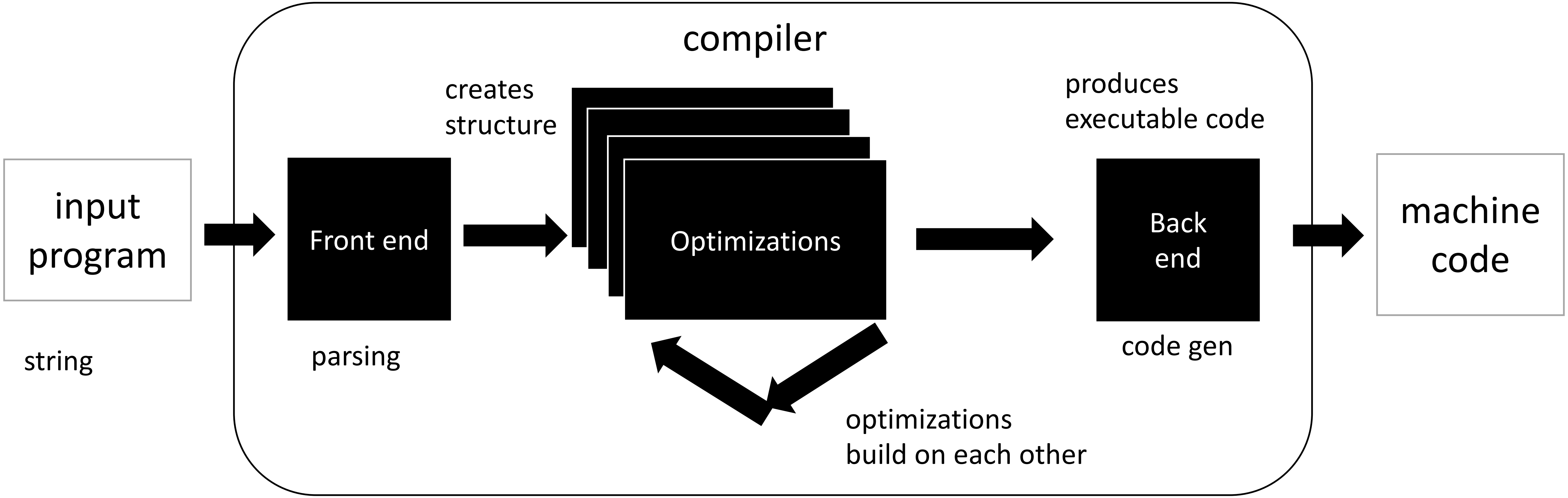
- You should know that Scanner generators exist
  - Lex
    - Classic C-based Scanner
  - PLY
    - Python implementation of Lex
  - Antlr
    - Modern scanner/parser generator

- Similar interfaces, but not exactly the same
  - PLY lexemes contain line/column numbers
  - PLY using “token()”, lex uses “yylex()”

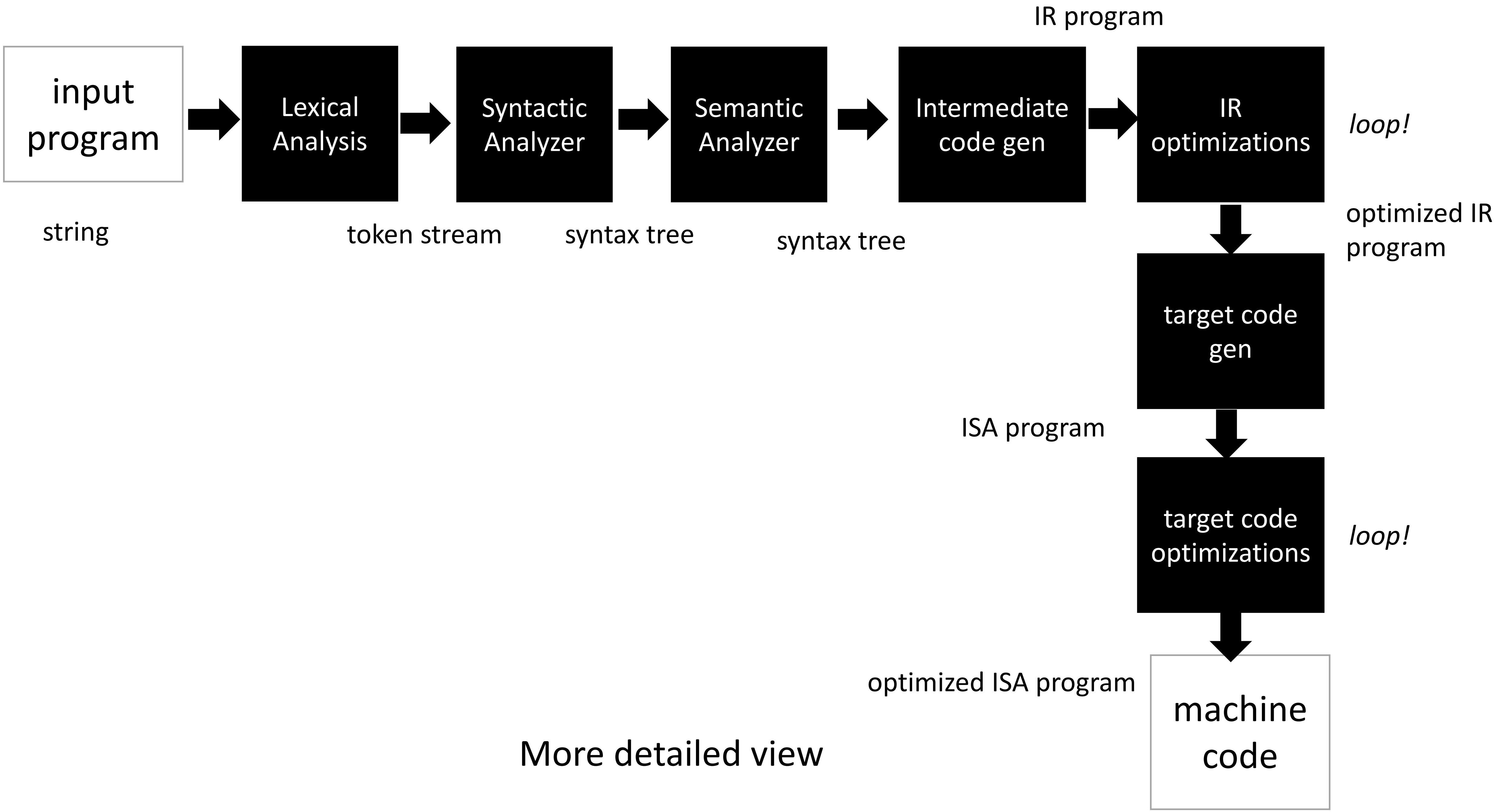
New module

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

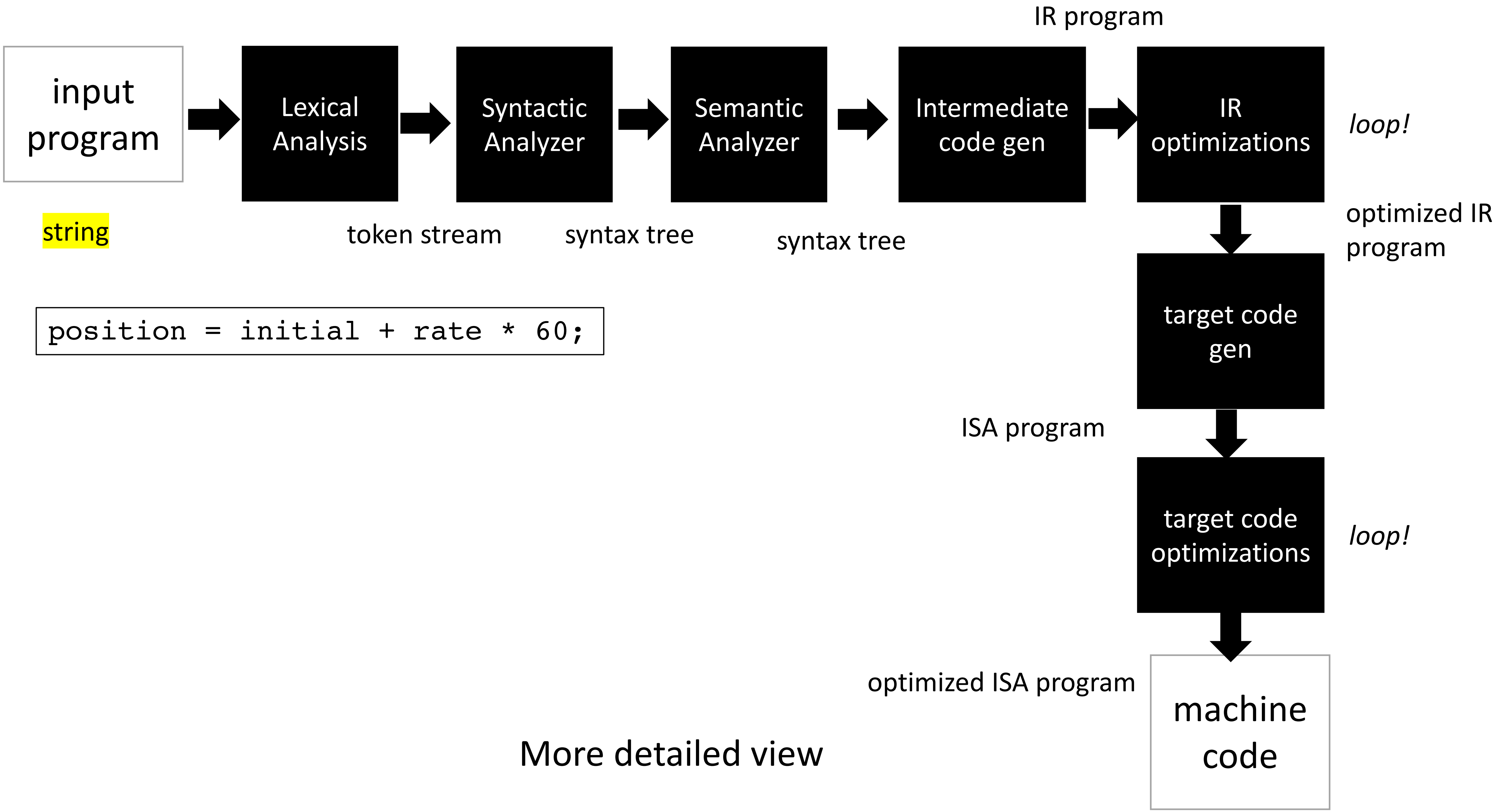
Compiler Architecture



This module will finish up the front end

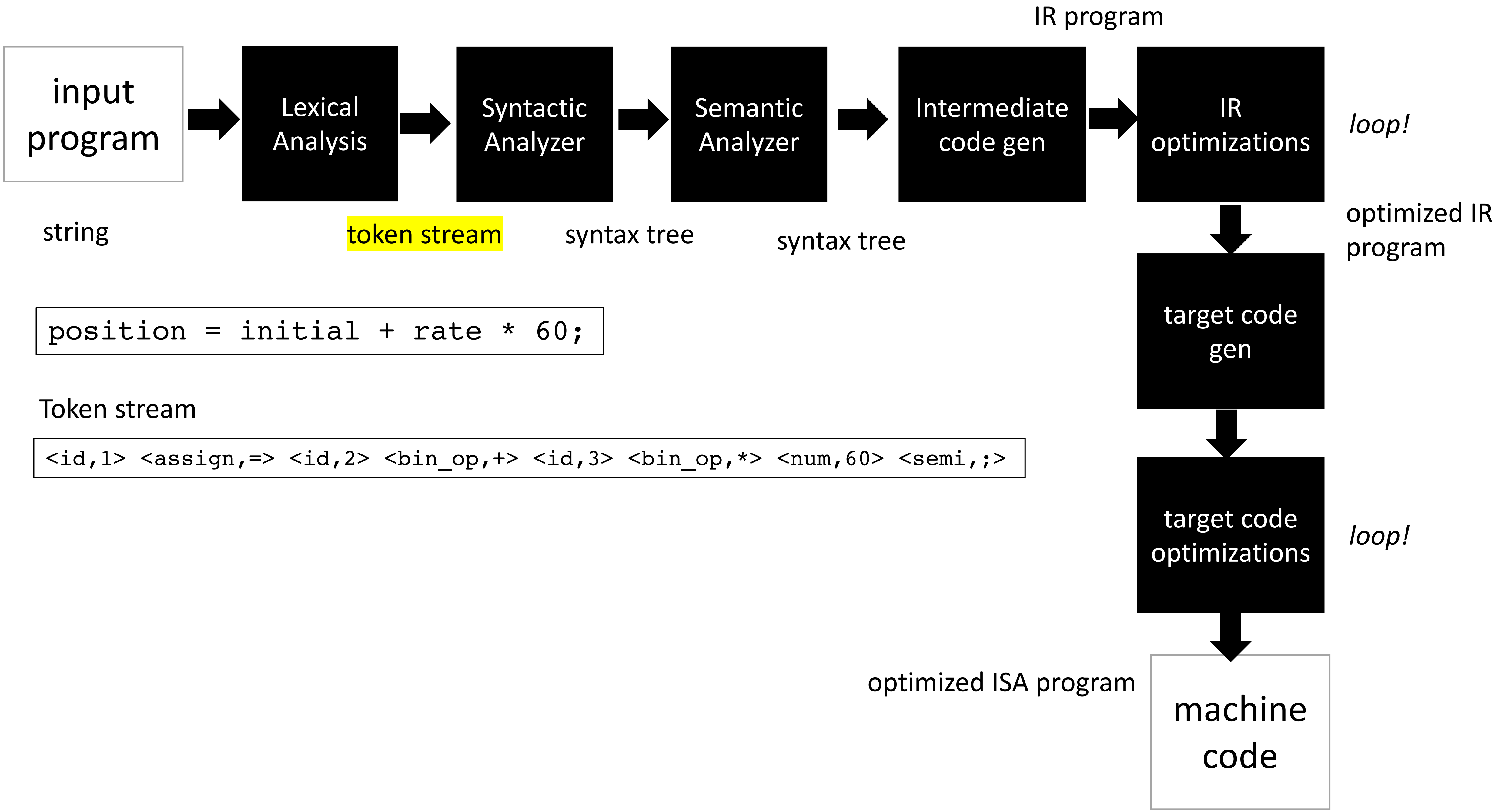


More detailed view

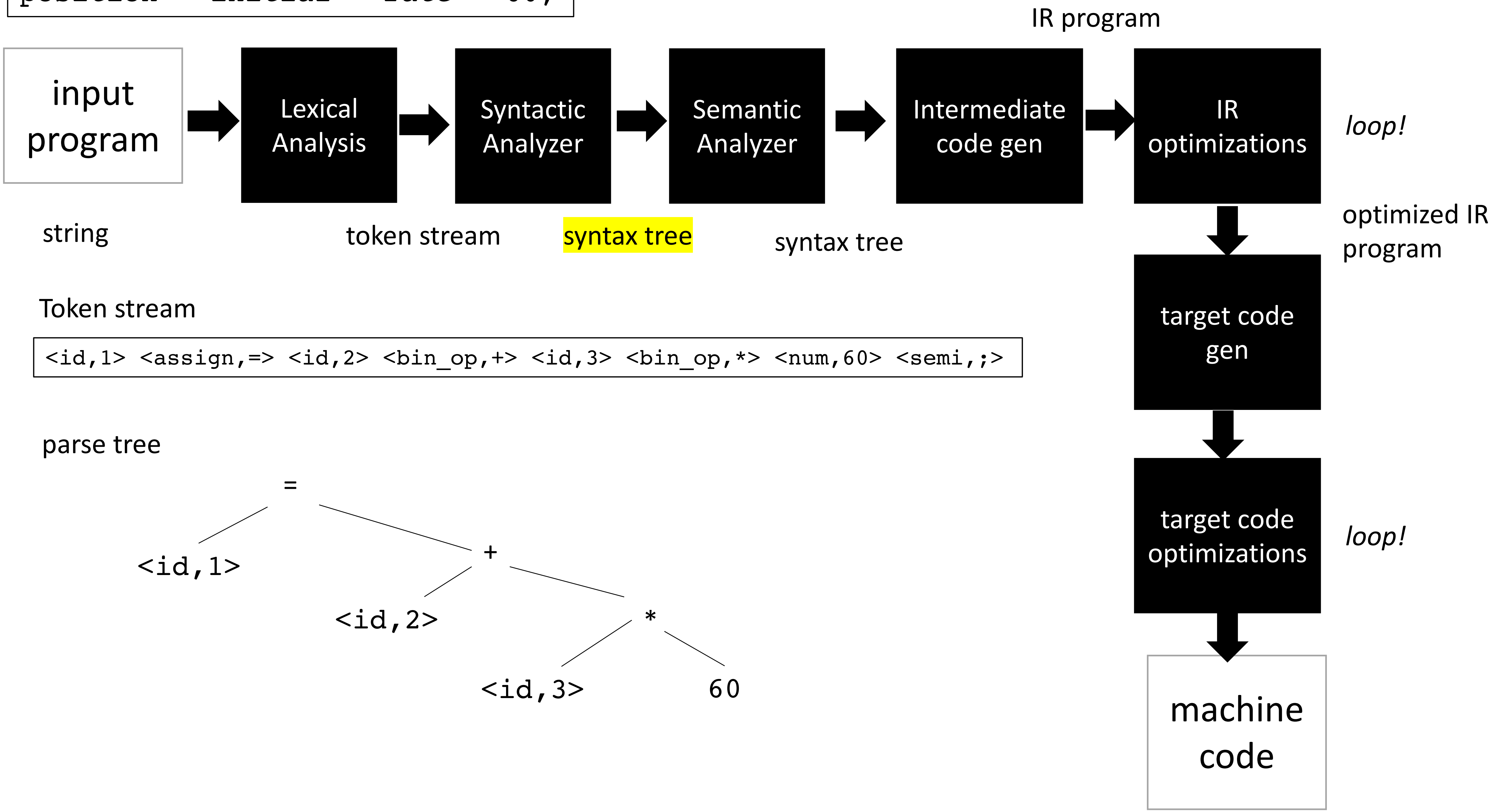



```
position = initial + rate * 60;
```

More detailed view


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

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

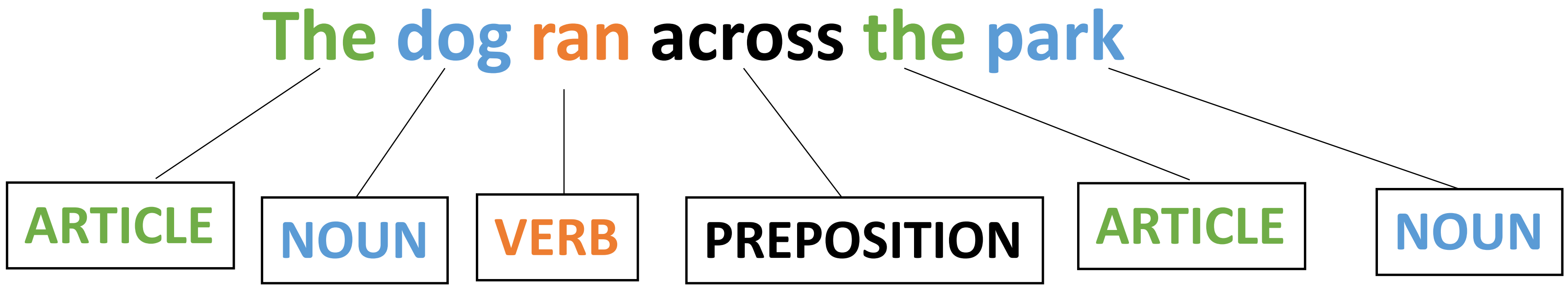


Syntactic Analysis

- 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



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.

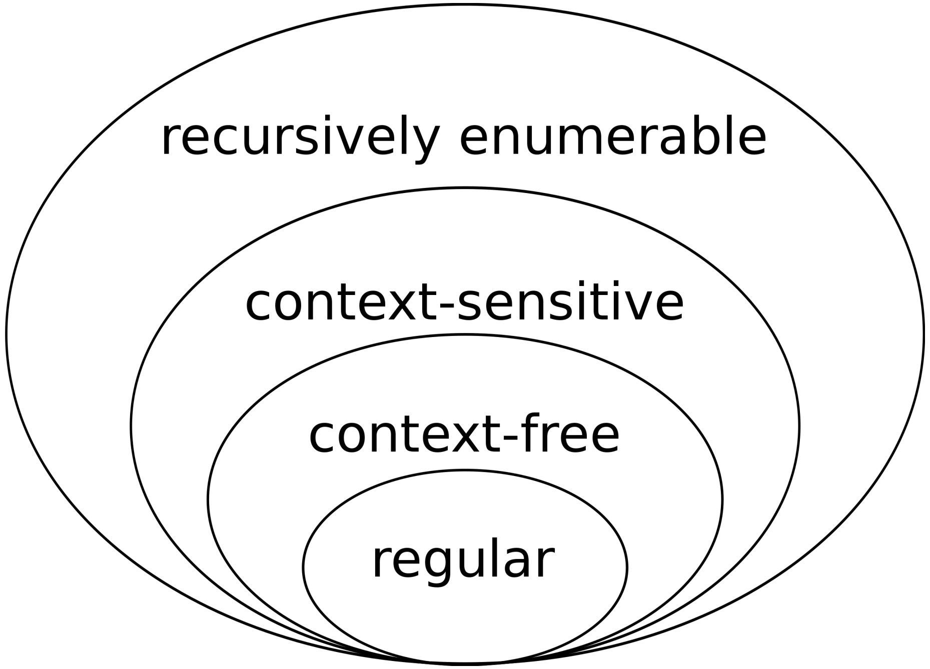
- We will use ***context free grammars***

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 like 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 like 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

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

- Production rules contain a sequence of either non-terminals or terminals
- In our class, terminals will either be string constants or tokens

```
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'
```

RULE	Sentential Form
start	SheepNoise

RULE	Sentential Form
start	SheepNoise

Deriving strings

Give each production rule a numeric id

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

RULE	Sentential Form
start	SheepNoise
2	baa

RULE	Sentential Form
start	SheepNoise
1	baa SheepNoise
2	baa baa

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`

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:

Expr

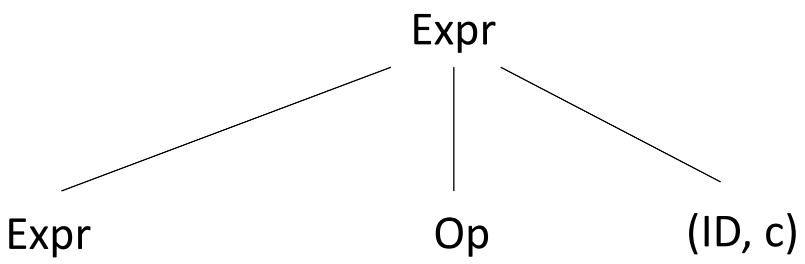
A more complicated example

```
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

We can visualize this as a tree:



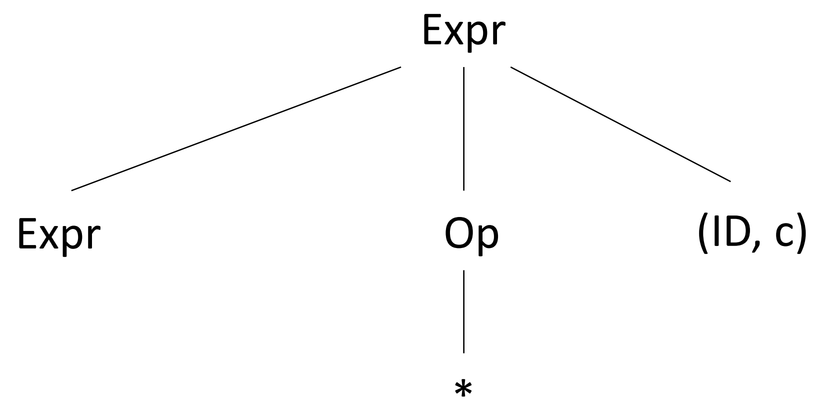
A more complicated example

```
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:



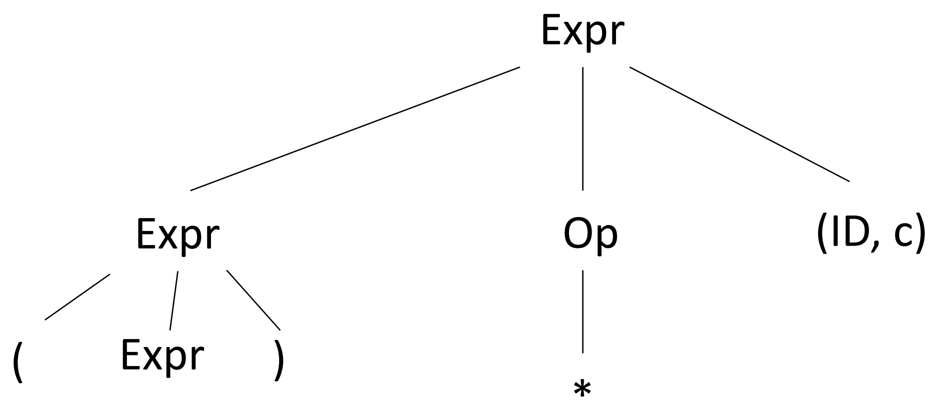
A more complicated example

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

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:



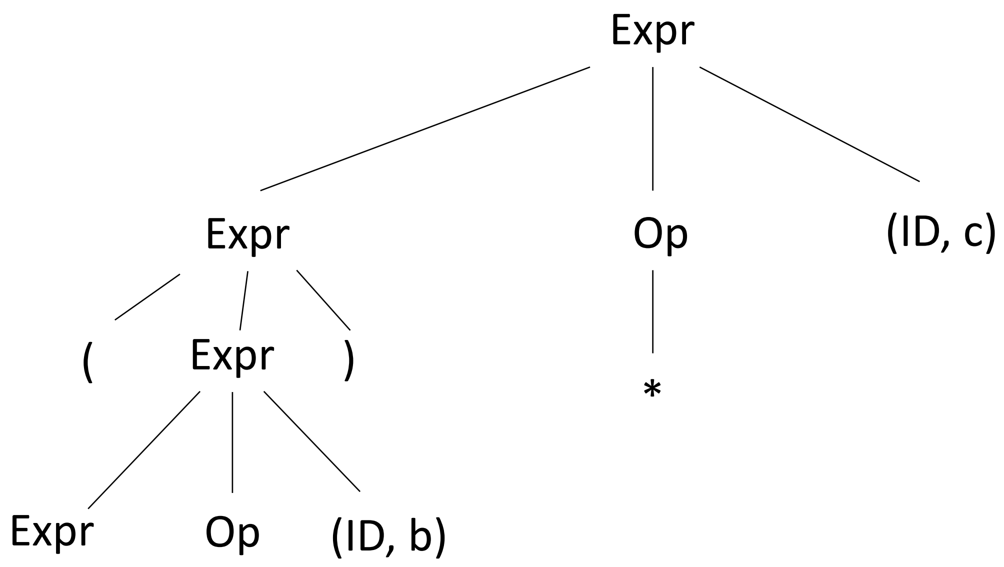
A more complicated example

```
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:



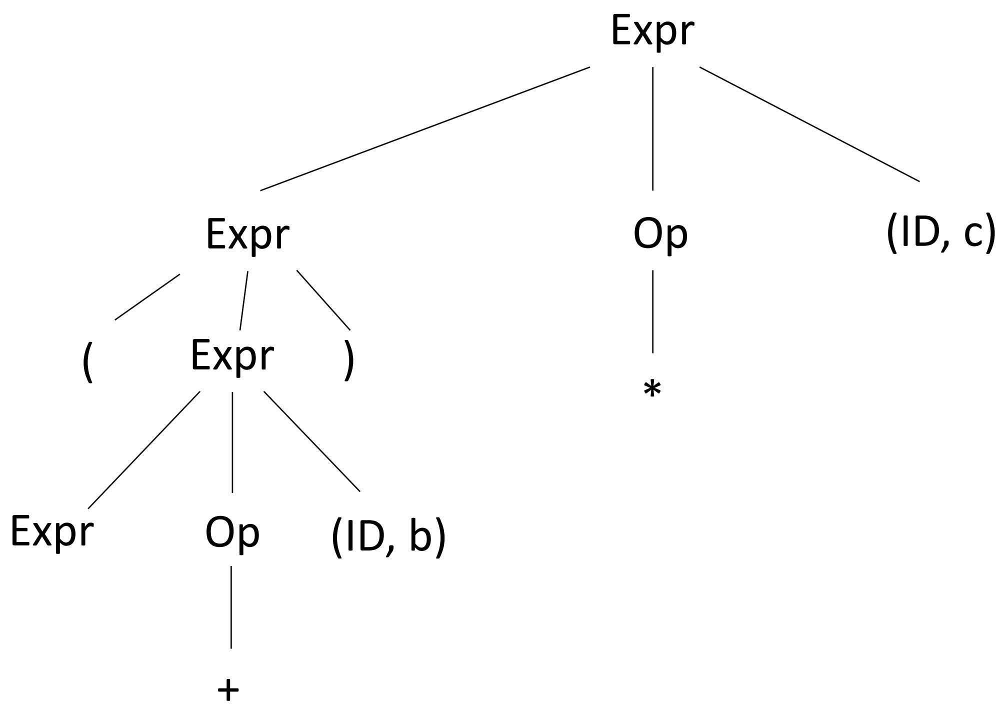
A more complicated example

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

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:



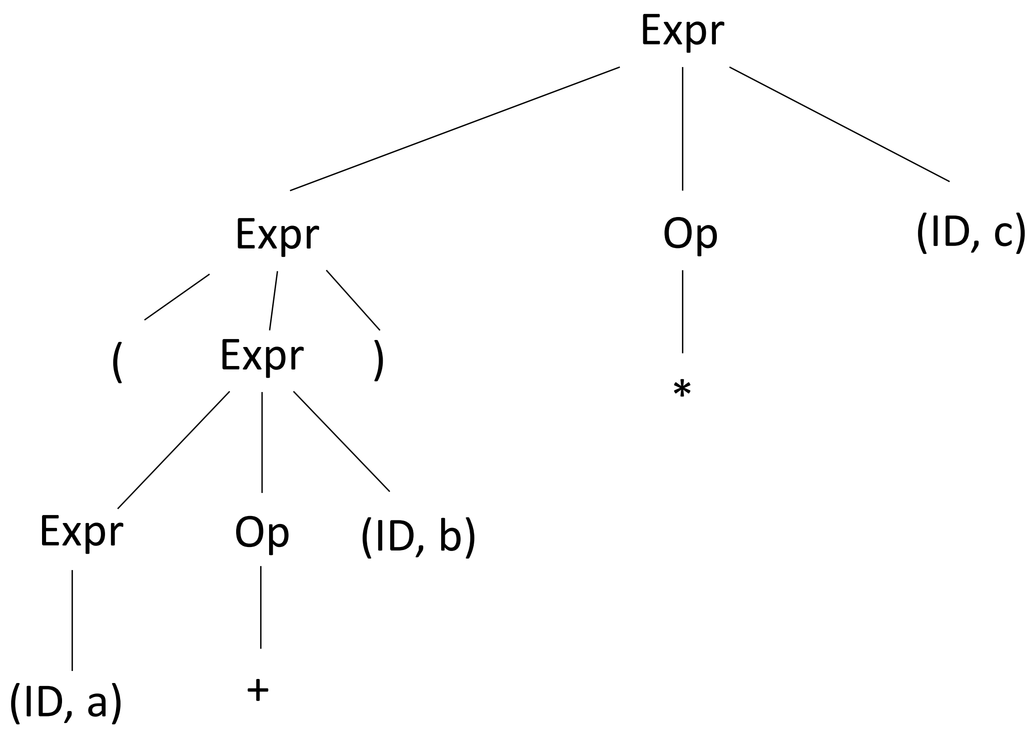
A more complicated example

```
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

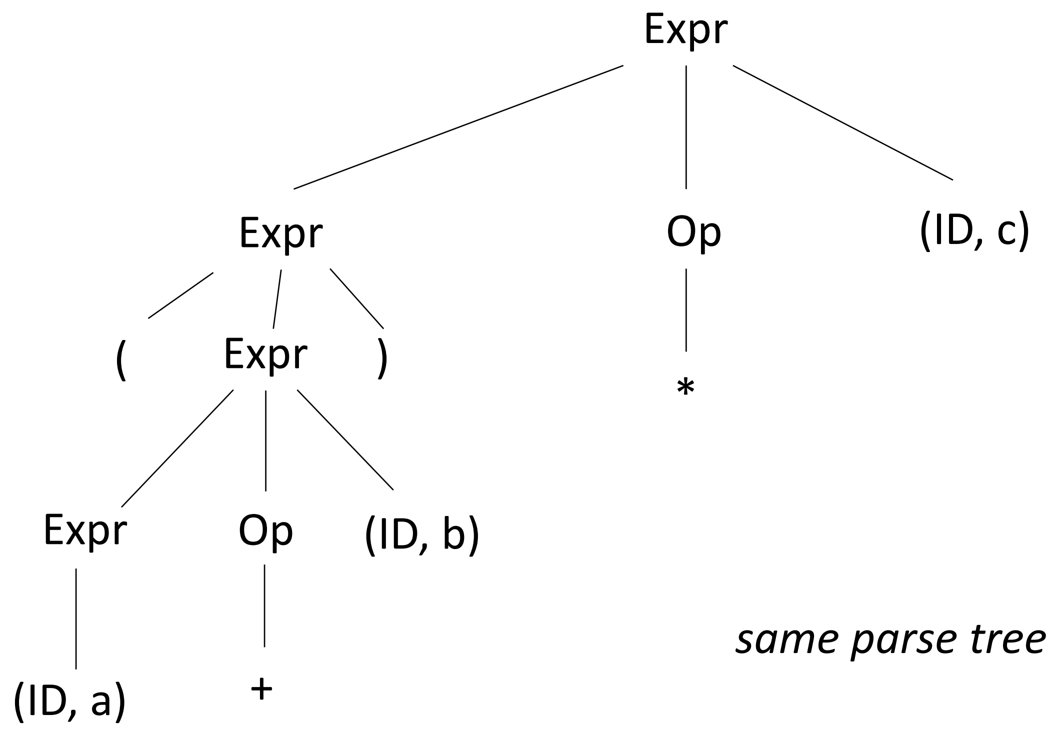
RULE	Sentential Form
start	Expr
2	Expr Op ID
1	(Expr) Op ID
2	(Expr Op ID) Op ID
3	(ID Op ID) Op ID
4	(ID + ID) Op ID
5	(ID + ID) + ID

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?

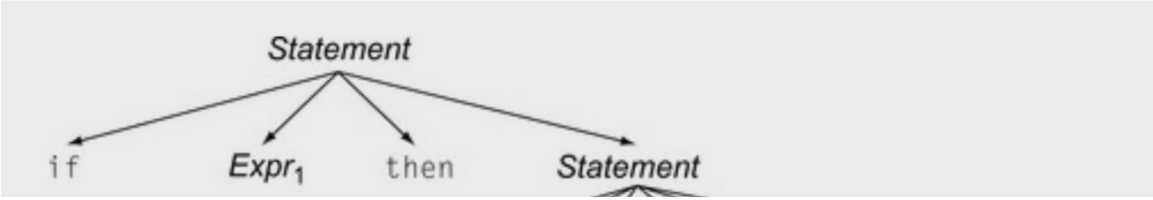
if *Expr1* **then** **if** *Expr2* **then** *Assignment1* **else** *Assignment2*

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:             |   ....
```

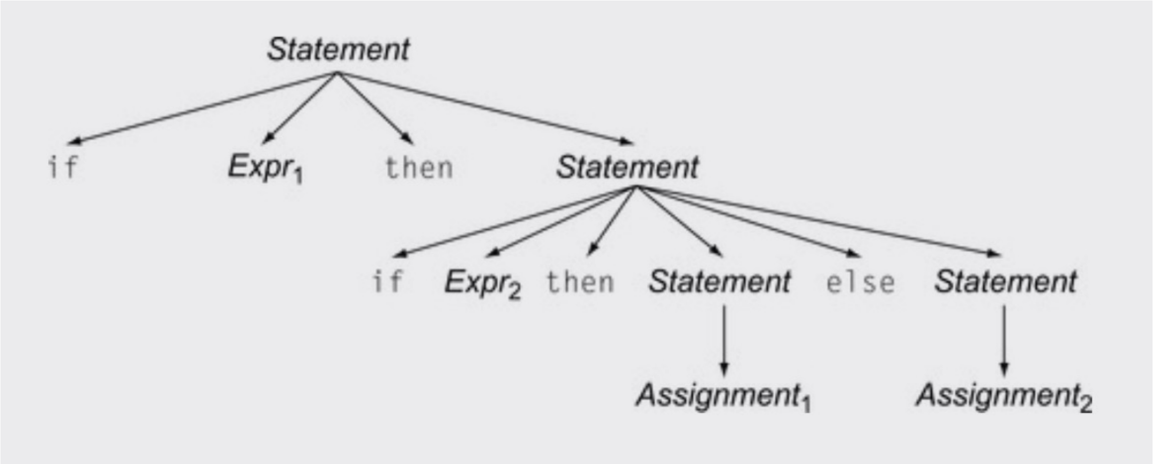
if *Expr*$_1$ **then** **if** *Expr*$_2$ **then** *Assignment*$_1$ **else** *Assignment*$_2$



Ambiguous grammars

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

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

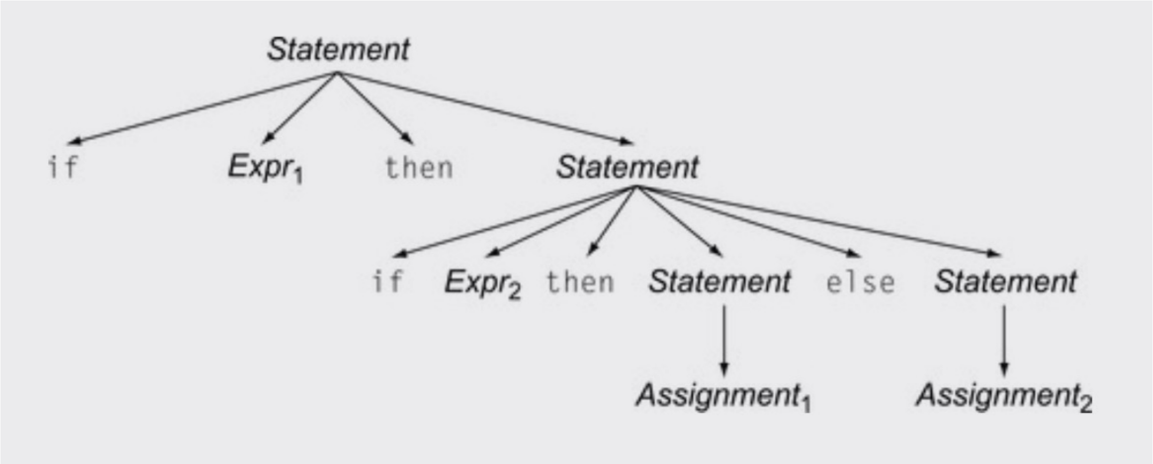


Valid derivation

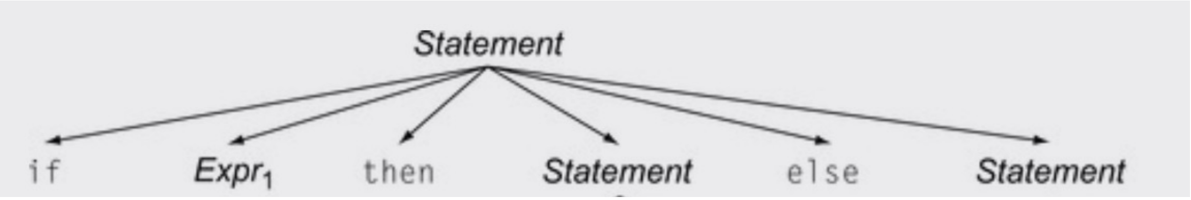
Ambiguous grammars

```
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** $Assignment_2$



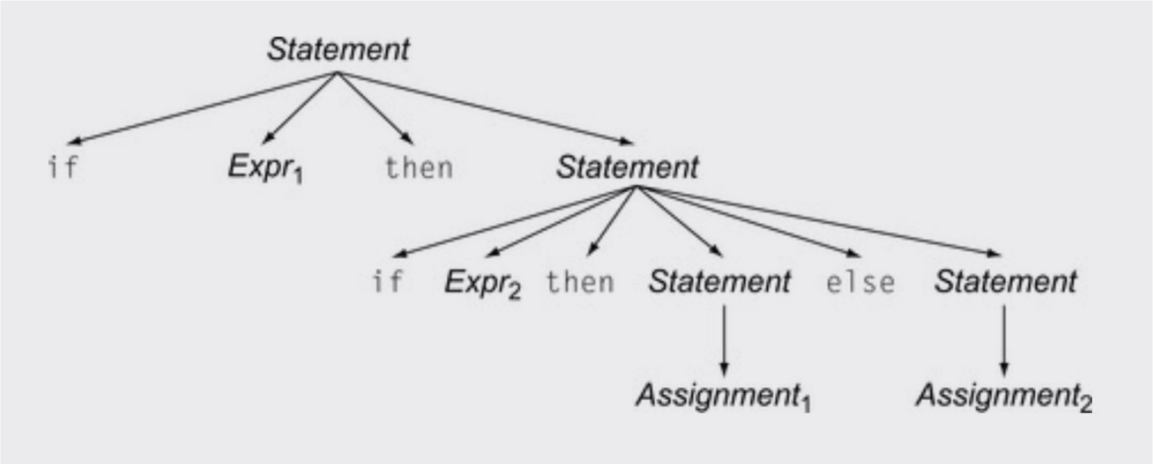
Valid derivation



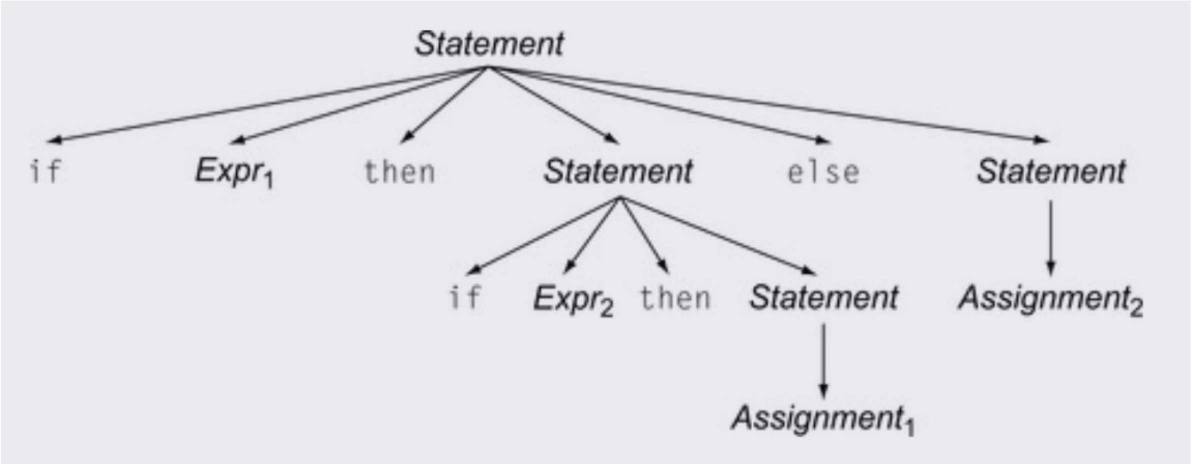
Ambiguous grammars

```
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*$_2$



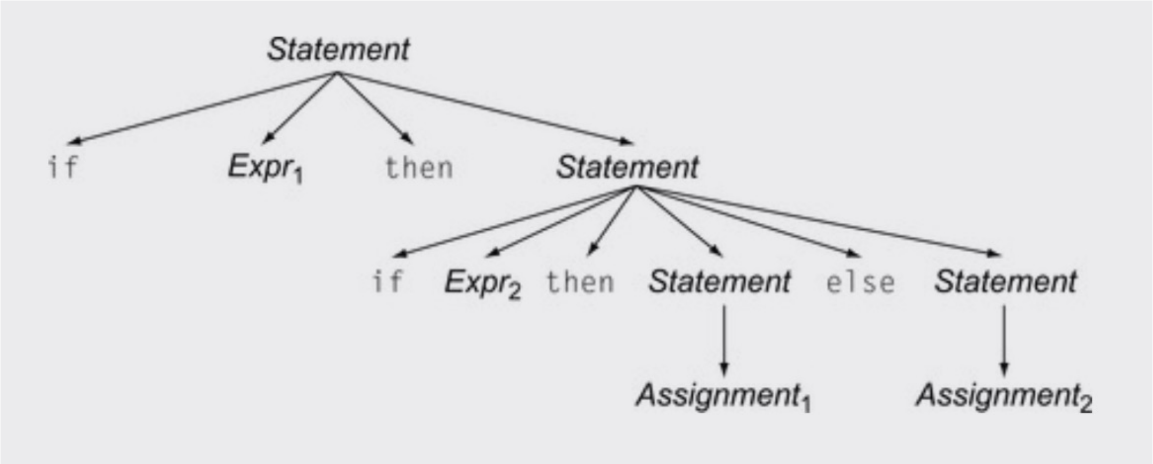
Valid derivation



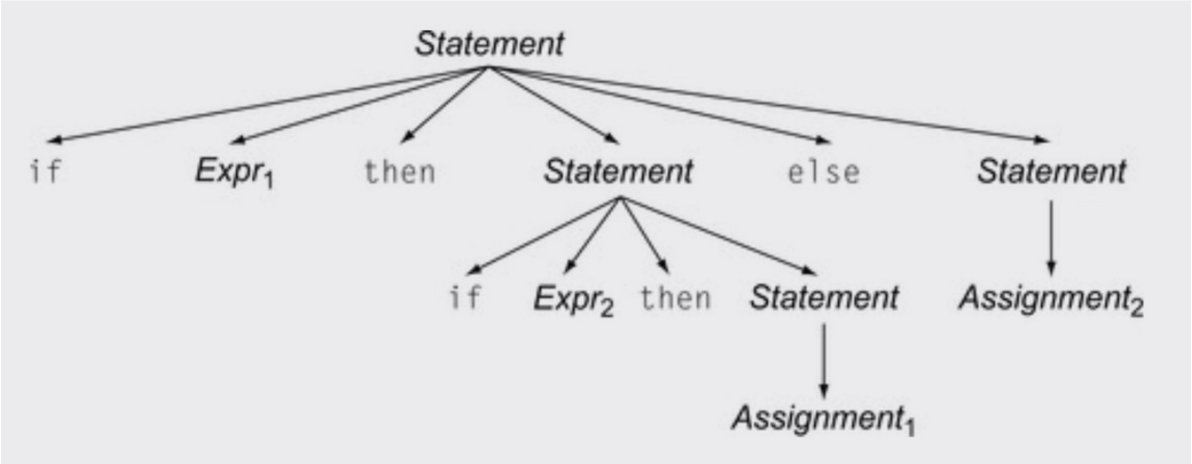
Also a valid derivation

Ambiguous grammars

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



Valid derivation



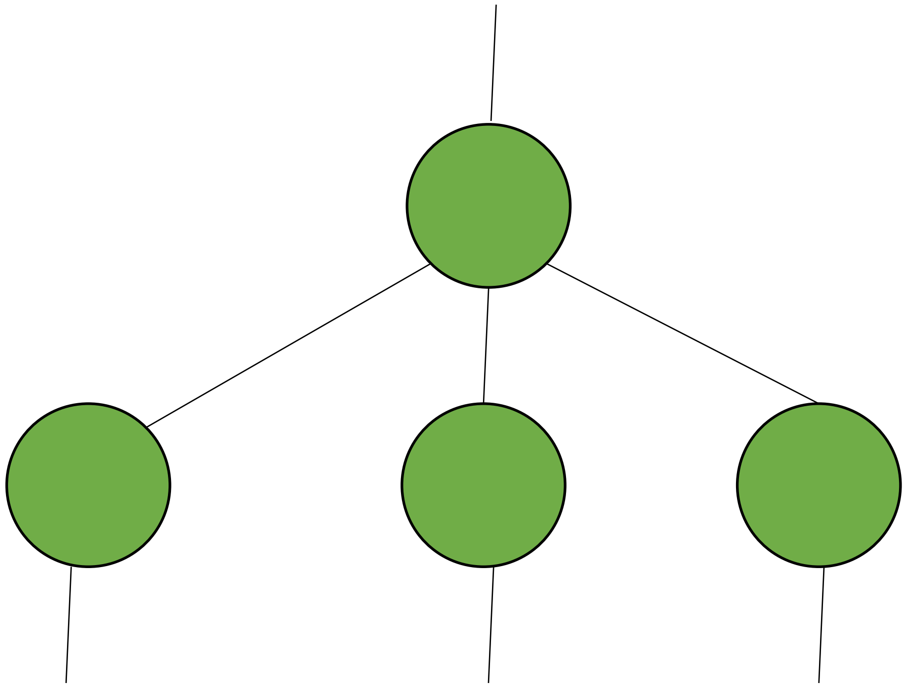
Also a valid derivation

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 the compiler

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

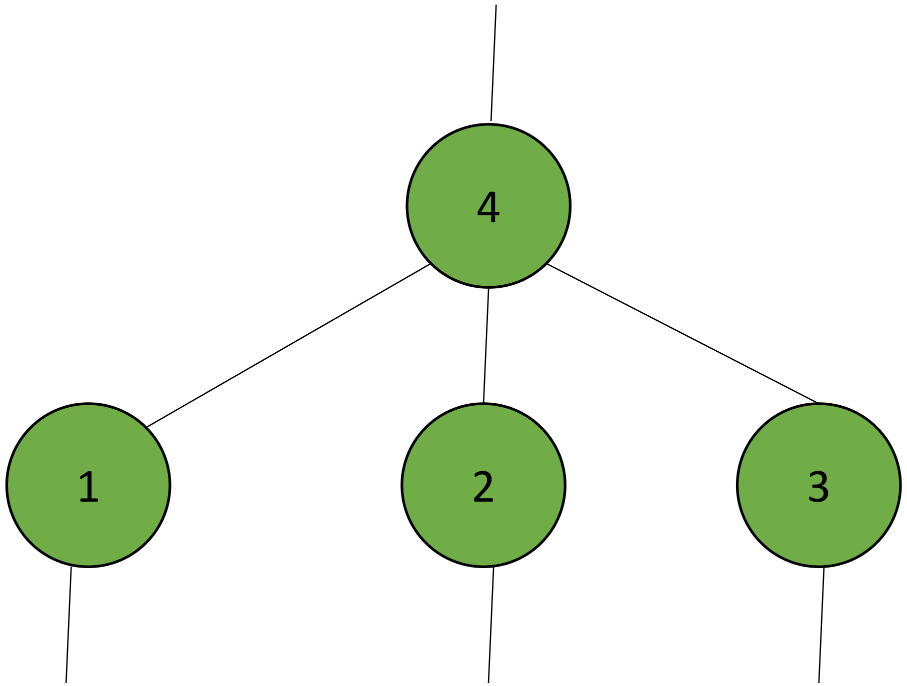
Post order traversal



visiting for for different types
of traversals:

pre order?
in order?
post order

Post order traversal

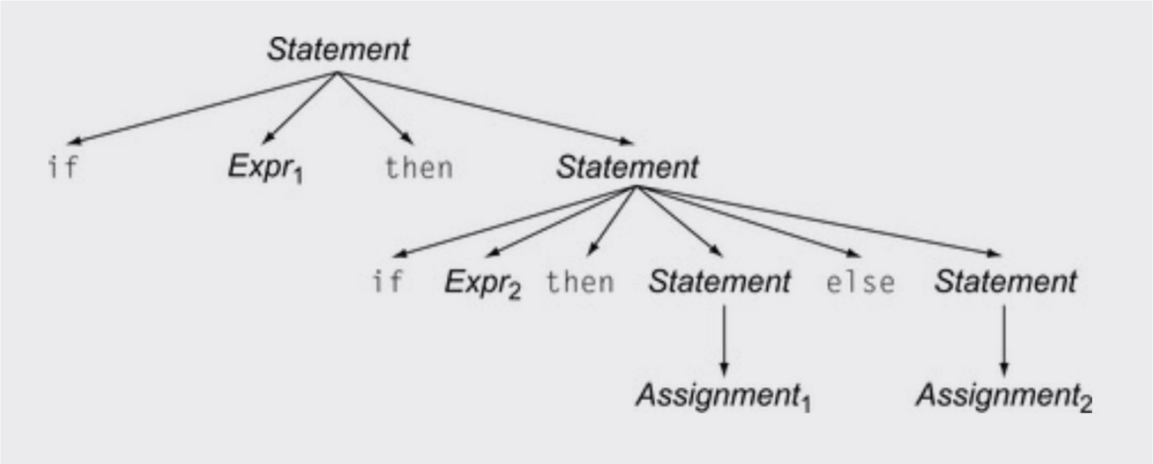


visiting for for different types of traversals:

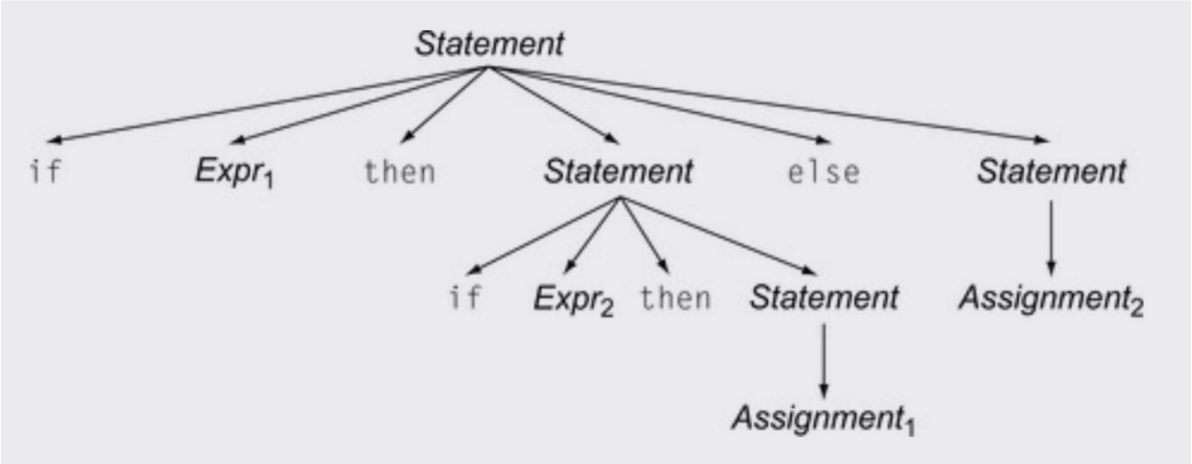
post order

Ambiguous grammars

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



Valid derivation



Also a valid derivation

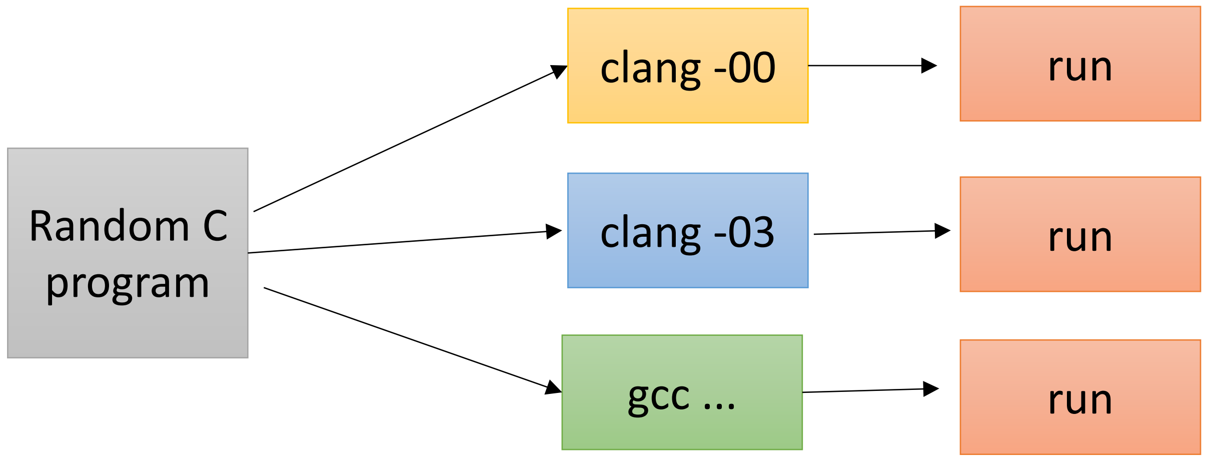
We will study how to eliminate ambiguity

- But I want to close out today with an interesting case study

Case study

- Using a CFG, you can derive random strings in a language

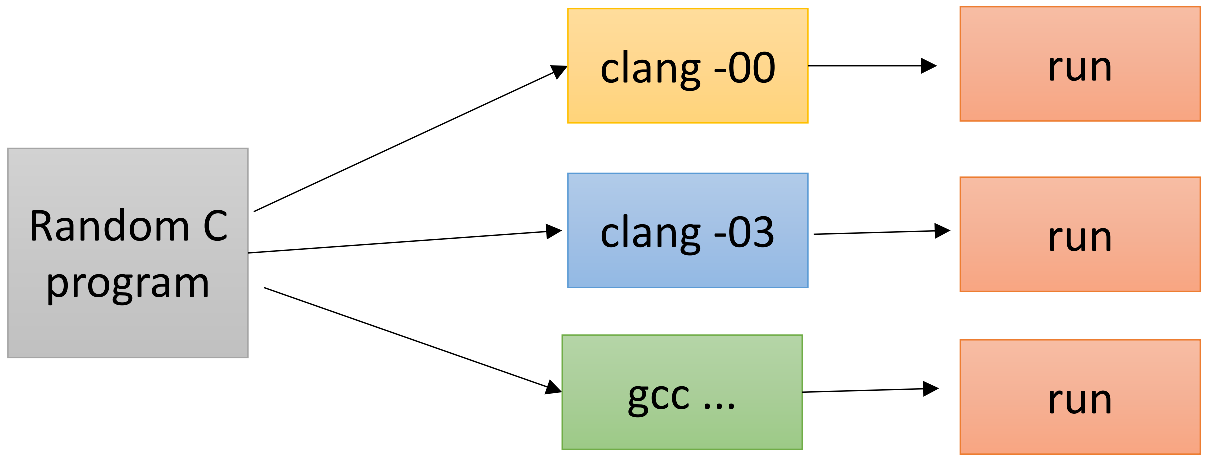
- C-Smith
  - Generates random C programs
  - Used to test compiler correctness



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

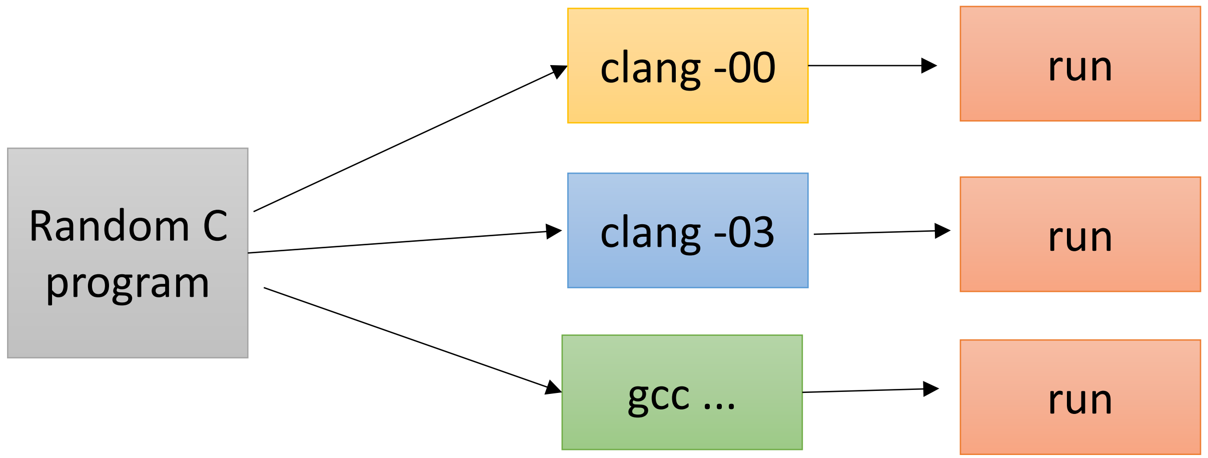
Case study

- 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

On Wednesday

- How to remove ambiguity from grammars
  - Precedence
  - Associativity