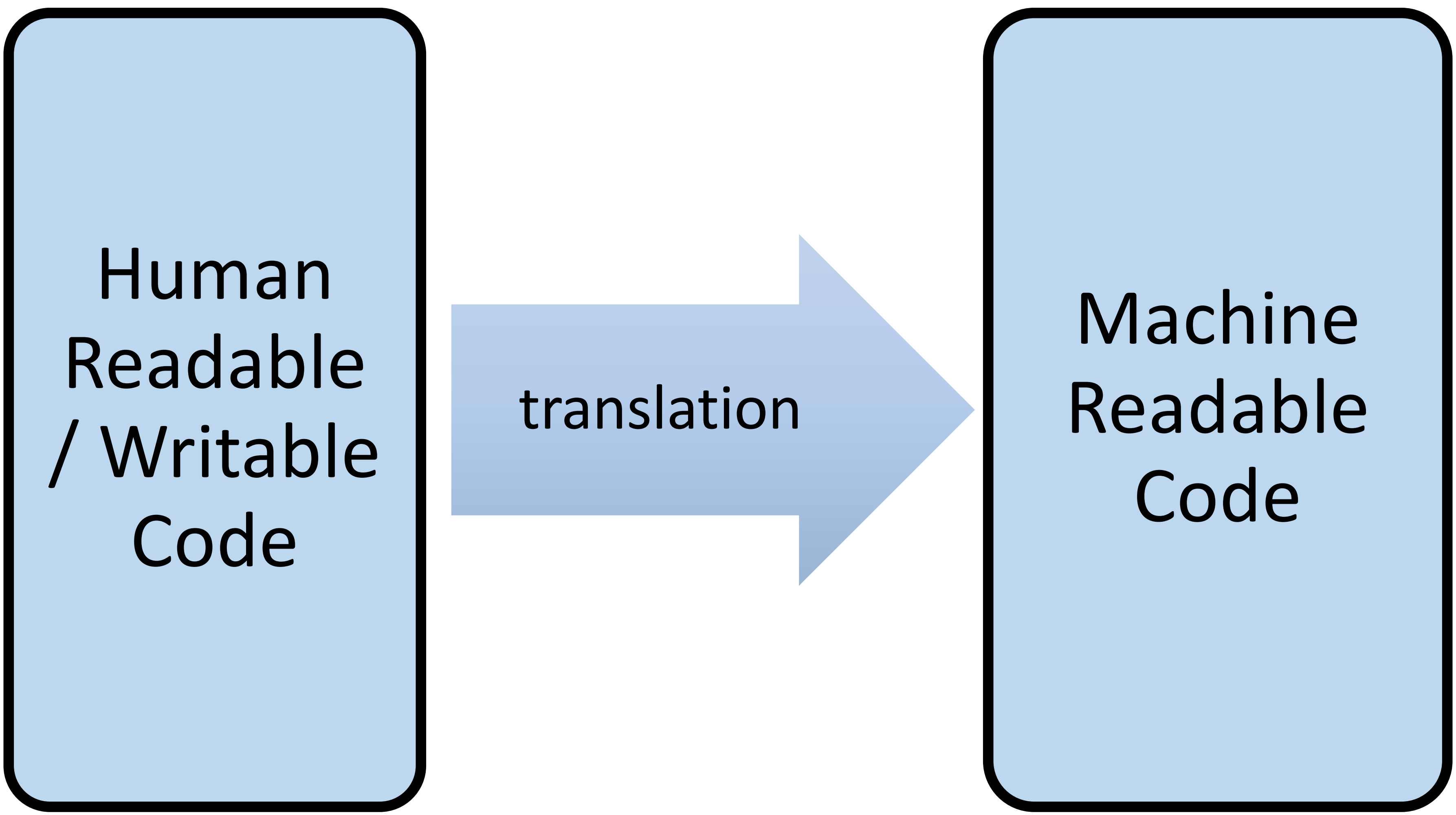
Program Translation

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
- Lecture Contents
  - Trial Top Hat Quiz
  - Program Translation
  - Introduction to Formal Languages
  - Regular Languages

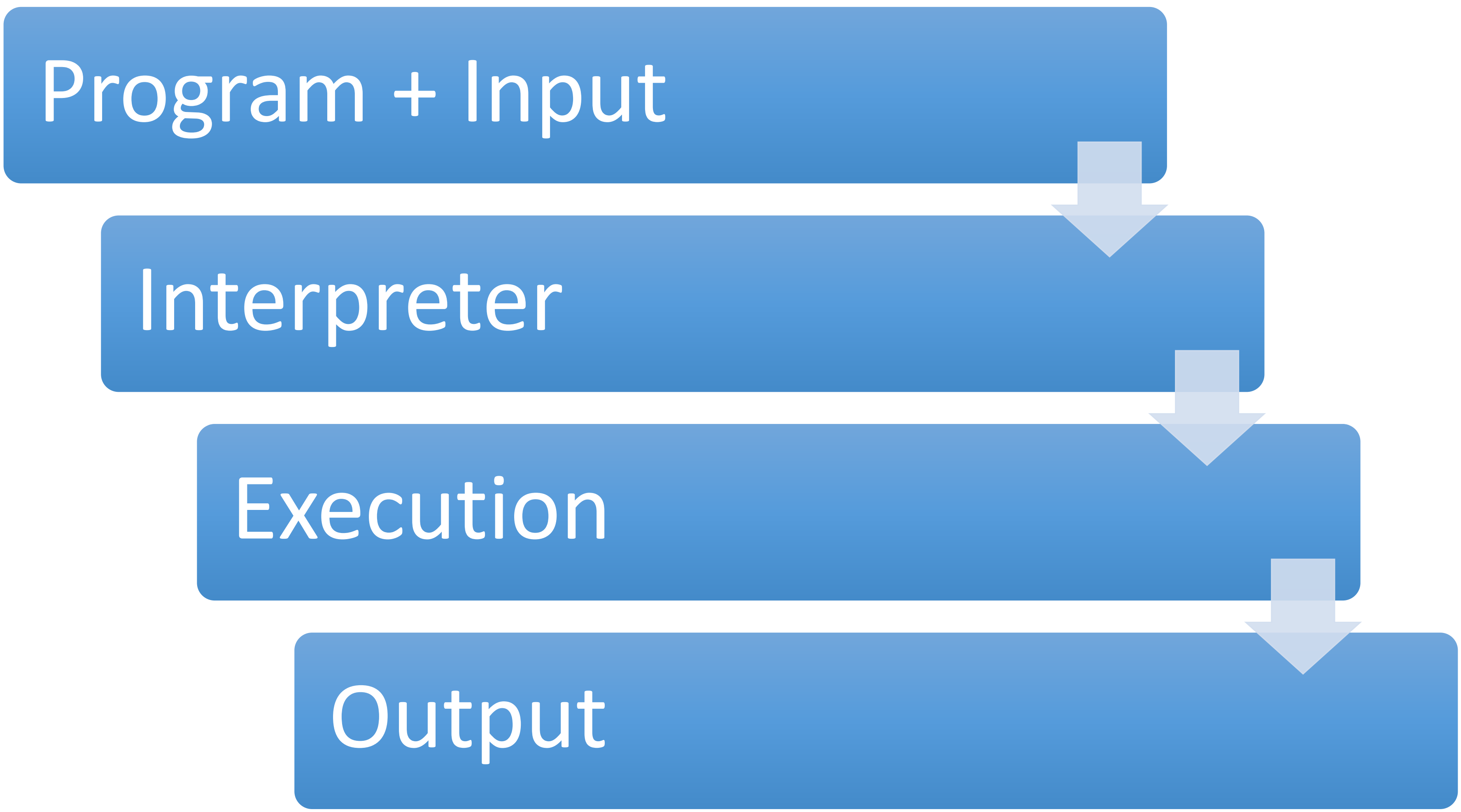
What makes languages useful?



Program Translation

- Motivation
  - All programs (unless written in machine code) are meaningless (to a computer)
  - These programs must be translated into machine code
  - Without this step all languages would be academic
- Forms of Program Translation:
  - Compilation:
    - Translates program to machine code
    - User can then run the machine code on the computer
  - Interpretation:
    - Executes program as it is translating it

Interpretation



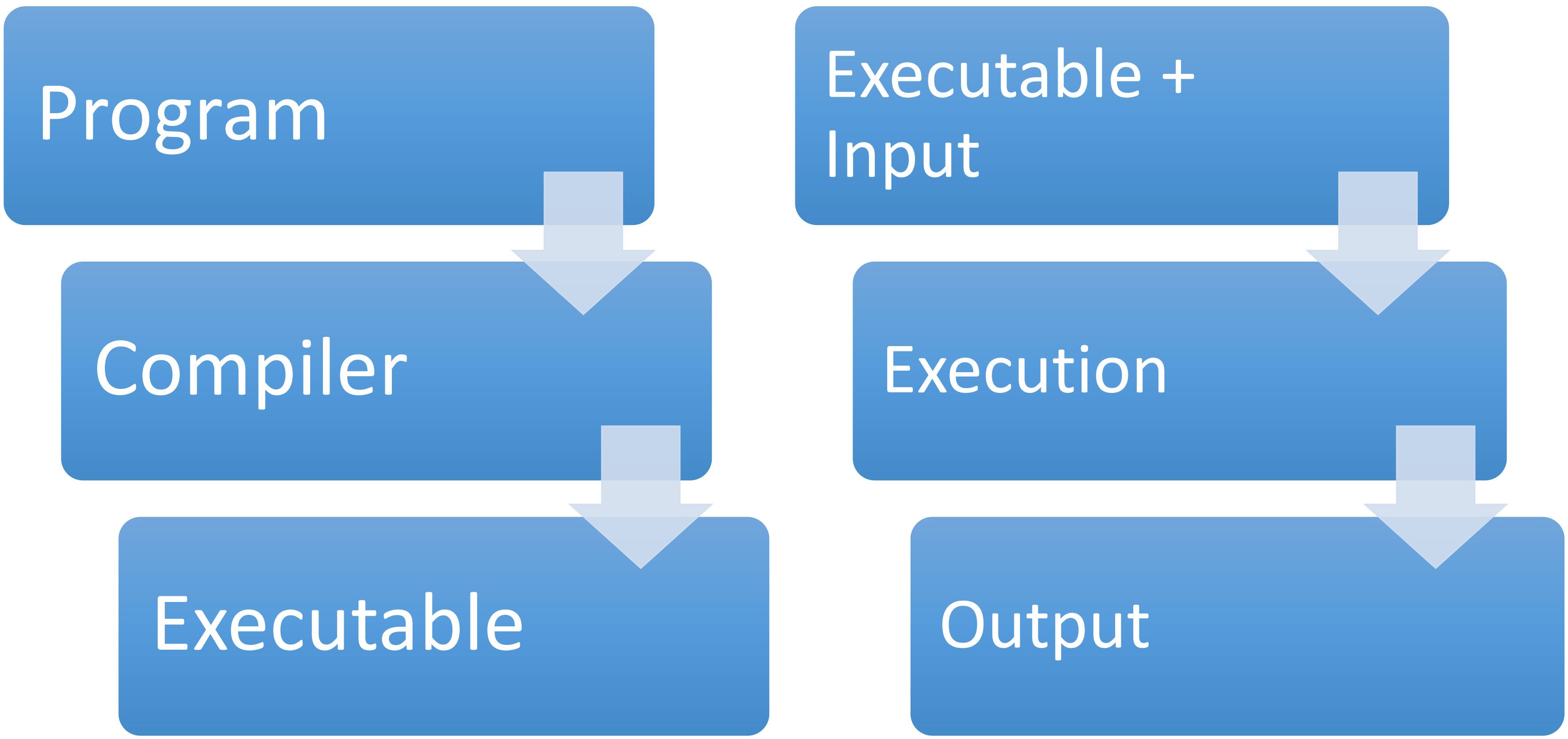
Features of Interpretation

- Faster development (maybe)
- More expressiveness (dynamic program generation)
- Late binding and dynamic features
- Interpreters translate programs as they run them
  - Perform program analysis (syntax and semantic) during execution

    e.g., Perl, Python, Basic,
  - You can get a syntax error during execution

Compilation

Translate once **Run Many**



Features of Compilation

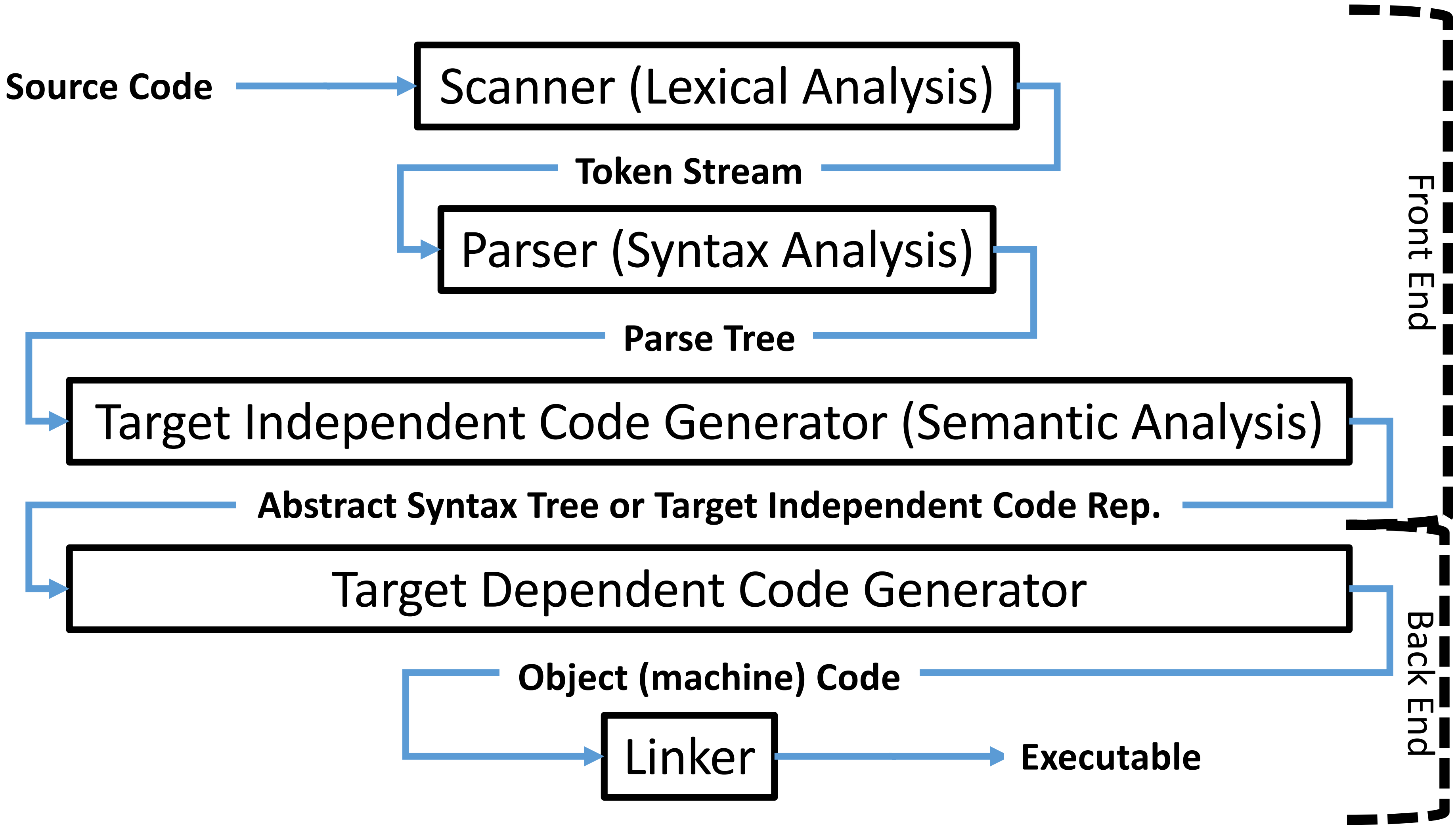
- Stand-alone code
- Efficient code and execution
- Compilers translate programs into machine, intermediate or byte-code representations
  - Perform all syntax and semantic analysis up front.

    e.g., Java, C, Fortran, etc.
  - You will never get a syntax error during execution
- **For the first part of the course we focus on compilation**

Aside: The Best of Both Worlds?

- Just-in-time compiling (Perl, Java, etc)
- Include interpreter in executable
- Late binding and dynamic features

Phases of Compilation



Example: Source Code

```
# This function takes 2 positive integers
# and prints their GCD
def gcd(m,n):
  while m != n:
    if m > n:
      m = m - n
    else:
      n = n - m
print m
```

Lexical Analysis

- Group characters into tokens
  E.g., keywords, literals, identifiers, punctuation
- Strip out items ignored by the compiler
  E.g., white space and comments
- Flag any unknown tokens as errors

Example: Token Stream

```
# This function takes 2 positive integers
# and prints their GCD
def gcd(m,n):
  while m != n:
    if m > n:
      m = m - n
    else:
      n = n - m
print m
```

def	gcd	(	m	,	n	)	:		while	m	!=	n	:	
		if	m	>	n	:				m	=	m	-	n
		else	:				n	=	m	-	n	print	m	

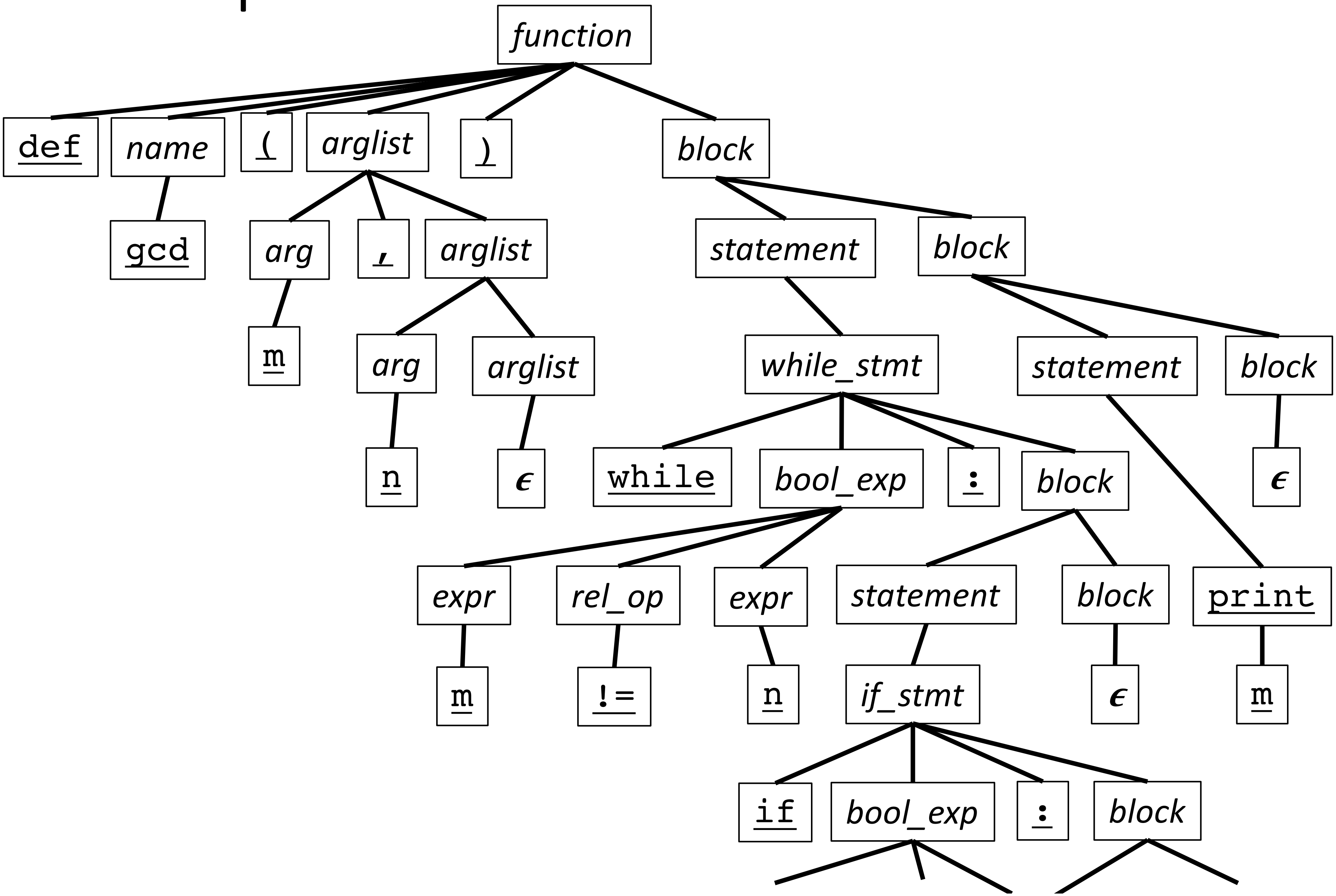
Syntax Analysis

- Organize tokens into a parse tree according to language grammar
  - E.g. an assignment statement is of the form:

    ```
    lvalue = expression
    ```

- Ensure token sequence conforms to the grammar
  - E.g., generate syntax errors if not.

Example: Parse Tree

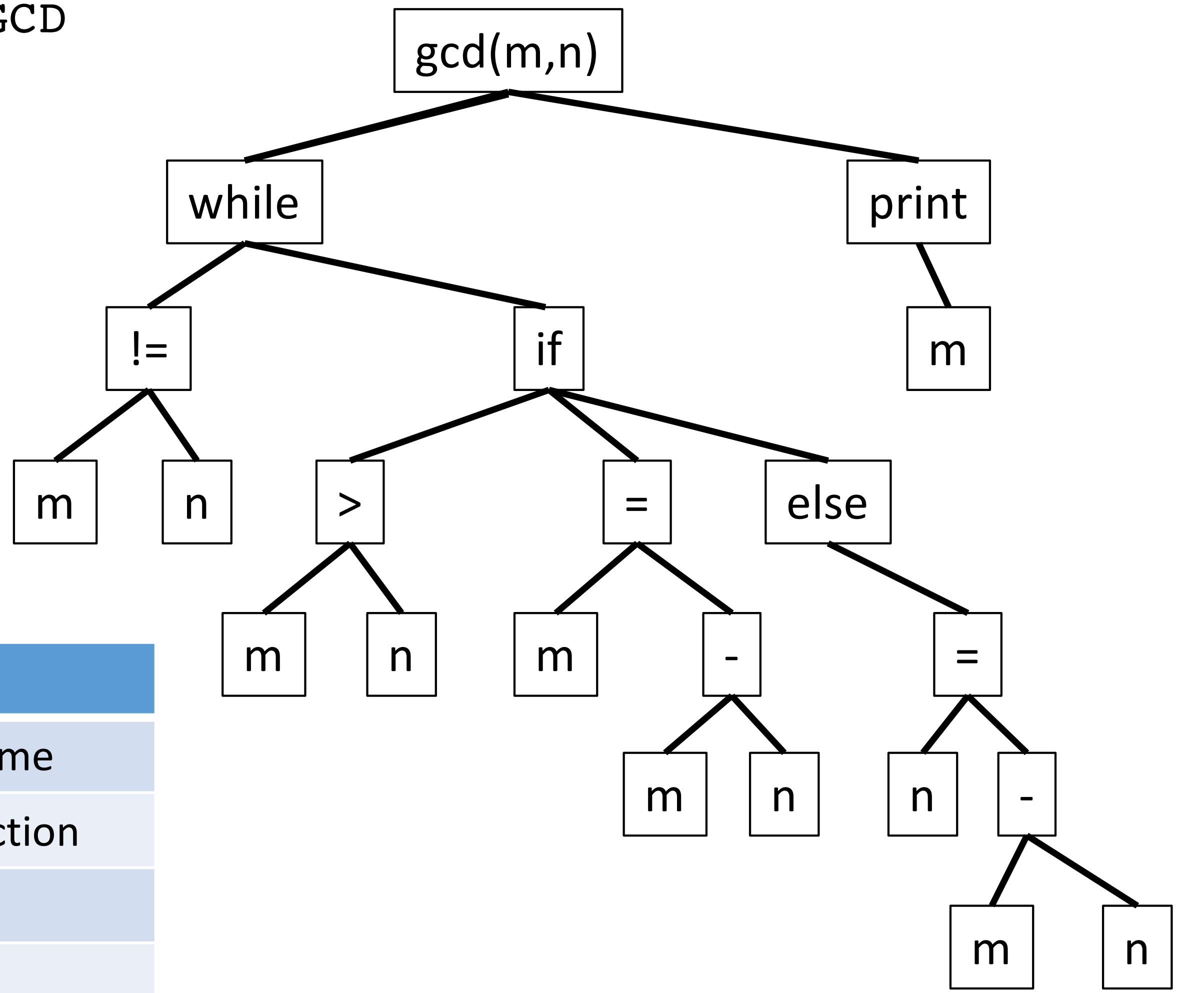


Semantic Analysis / Code Generation

- Generate symbol table of all identifiers
- Ascribe meaning to all identifiers
- Condense abstract syntax tree to only important nodes
- Generate intermediate code for each node
- Ensure the abstract syntax tree is meaningful
  - E.g., foo() makes no sense if foo is a variable instead of a function name

Example: Symbol Table and Abstract Syntax Tree

```
# This function takes 2 positive integers
# and prints their GCD
def gcd(m,n):
  while m != n:
    if m > n:
      m = m – n
    else:
      n = n – m
print m
```



Symbol Table

Index	Symbol	type
0	gcd	function name
1	print	built-in function
2	m	parameter
3	n	parameter

Example: Code Generation

- Generate code for each statement using the abstract syntax tree.
- E.g., m = m - n
  - For nodes: m – n

    move idx2 → r1

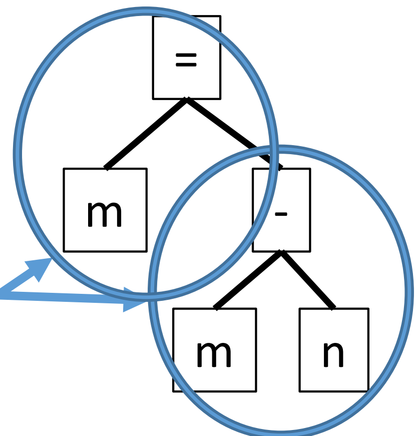
    move idx3 → r2

    sub r1, r2 → r3

    move r3 → idx4
  - For node: m =

    move idx4 → r1

    move r1 → idx2



Index	Symbol	type
0	gcd	function name
1	print	built-in function
2	m	parameter
3	n	parameter
4	tmp1	temporary var

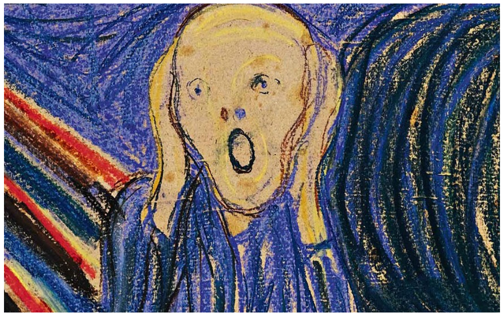
Where do we start?

- At the beginning: Lexical Analysis
- We need a scanner:

Source Code → Scanner (Lexical Analysis) → **Token Stream**

How Do We Build a Scanner?

- Need to be able to
  - Specify unambiguously the tokens of a language
  - Build a scanner from the specification
  - Generate the token stream in one pass (no going back)
- How do we do this?
  - Specify the tokens of a language?
  - Generate a scanner from the specification?
- We now need some **formal language theory**



Definitions

- A **language** L is set of strings over an alphabet Σ
- **Alphabet** Σ: is a finite set of characters (symbols)

  Examples
  - 0, 1
  - a, b, ... z
  - x, y, z

- A **string** σ is a finite sequence of characters from Σ

  Examples
  - 1001001
  - alex
  - yyz

- ε (epsilon) denotes the **empty string** (no characters)

More Definitions

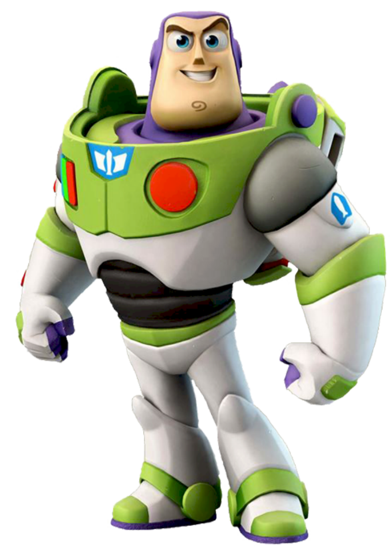
- $|\sigma|$ denotes the length of σ (# of chars)
  Examples
  - $|\varepsilon| = 0$
  - $|1001001| = 7$
  - $|alex| = 4$
  - $|yyz| = 3$
- Σ^i denotes languages (sets of strings) of length i over Σ
- Example: If $\Sigma = \{a, b\}$
  - $\Sigma^0 = \{\varepsilon\}$
  - $\Sigma^1 = \Sigma = \{a,b\}$
  - $\Sigma^2 = \Sigma\Sigma = \{aa,ab,ba,bb\}$
  - $\Sigma^3 = \Sigma\Sigma\Sigma = \{aaa,aab,aba,abb,baa,bab,bba,bbb\}$

$ST = \{st | s \in S, t \in T\}$

- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 ... = \cup_i \Sigma^i$ = set of all words composed of characters in Σ.
- This is called the Kleene-* notation.
- Note: If language L is over alphabet Σ, then $L \subseteq \Sigma*$

Examples of Languages

- Finite Languages
  - Σ^i for any fixed i
  - English words over the Latin alphabet
  - Java keywords
- Infinite Languages
  - Σ^*
  - Set of all English sentences
  - Set of all Java programs
  - $\{0^n | n \geq 0\}$
  - $\{0^n 1^n | n \geq 0\}$
  - $\{a^p | p \in \text{PRIMES}\}$
  - $\{\Sigma^+ @ \Sigma^+ (. \Sigma^+)^n | \Sigma = \{a,b,c,...z\}, n \geq 0\}$



Types of Languages

Type	Name	Recognizer	Compiler Phase
3	Regular	DFA	Scanner/Tokenizer
2	Context Free	NPDA	Parser
1	Context Sensitive	Linearly bound NTMs	Semantic Analyzer / Code Generator
0	Recursive Enumerable	Turing Machines	

Note: Tokens of a programming language almost always form a regular language.

Errrrr… What's a regular language?

Definition of Regular Languages

Recursive Definition:

- Base Cases:
  - $\emptyset$ (Empty language)
  - $\{\varepsilon\}$ (Language consisting of the empty string)
  - $\{a\}, a \in \Sigma$ (Language consisting of one symbol)

  are all regular languages
- Inductive Step: If L_1 and L_2 are regular then so are
  - $L_1L_2 = \{\sigma\tau \mid \sigma \in L_1, \tau \in L_2\}$
  - $L_1 \cup L_2 = \{\sigma \mid \sigma \in L_1 \vee \sigma \in L_2\}$
  - $L_1^* = \{\sigma_1\sigma_2\sigma_3 \ldots \sigma_{i-1}\sigma_i \mid \sigma_j \in L_1, i \geq 0\}$

Examples

Regular Languages

- {a,ab,abc}
- Any finite language
- $\{a\}*$
- $\{a,b,c\}*$
- $\{1^n0 | n > 0\}$

Non-regular languages

- $\{a^ib^jc^k | i > 0, j > i, k > j\}$
- $\{0^n1^n | n \geq 0\}$
- $\{a^p | p \in \text{PRIMES}\}$
- Set of all correct Java programs.

- Set of all positive integers (base 10)
- $\{\Sigma\Sigma^*@\ \Sigma\Sigma^*(.\ \Sigma\Sigma^*)^n | \ \Sigma = \{a,b,c,...z\}, n \geq 0\}$

Does this notation look somewhat familiar?