Program Translation

CSCI 3136: Principles of Programming Languages

Agenda

1. Program Translation
2. Introduction to Formal Languages
3. Regular Languages

Why do we use Programming Languages?

Because reading & writing machine code is slow and difficult!

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 (i.e., good only for theory and discussion)
- Forms of Program Translation:
  - Compilation:
    - Translates program to machine code
    - User can then run the machine code on the computer

Interpretation

Features of Interpretation

- Faster development (maybe/probably)
- More expressiveness (dynamic program generation)
- Late binding and dynamic features
- Slower execution (compilers translate once, interpreters translate for every execution)
- Interpreters translate programs as they run them (i.e., during execution)
  - Perform program analysis (syntax and semantic) during execution
    - e.g., Perl, Python, Basic

Compilation

- Translate once
- Run Many

Features of Compilation

- Stand-alone code
- Efficient code and execution
- Compiler’s translate programs into intermediate or byte-code representations
  - Perform as much as possible syntax and semantic analysis up front.

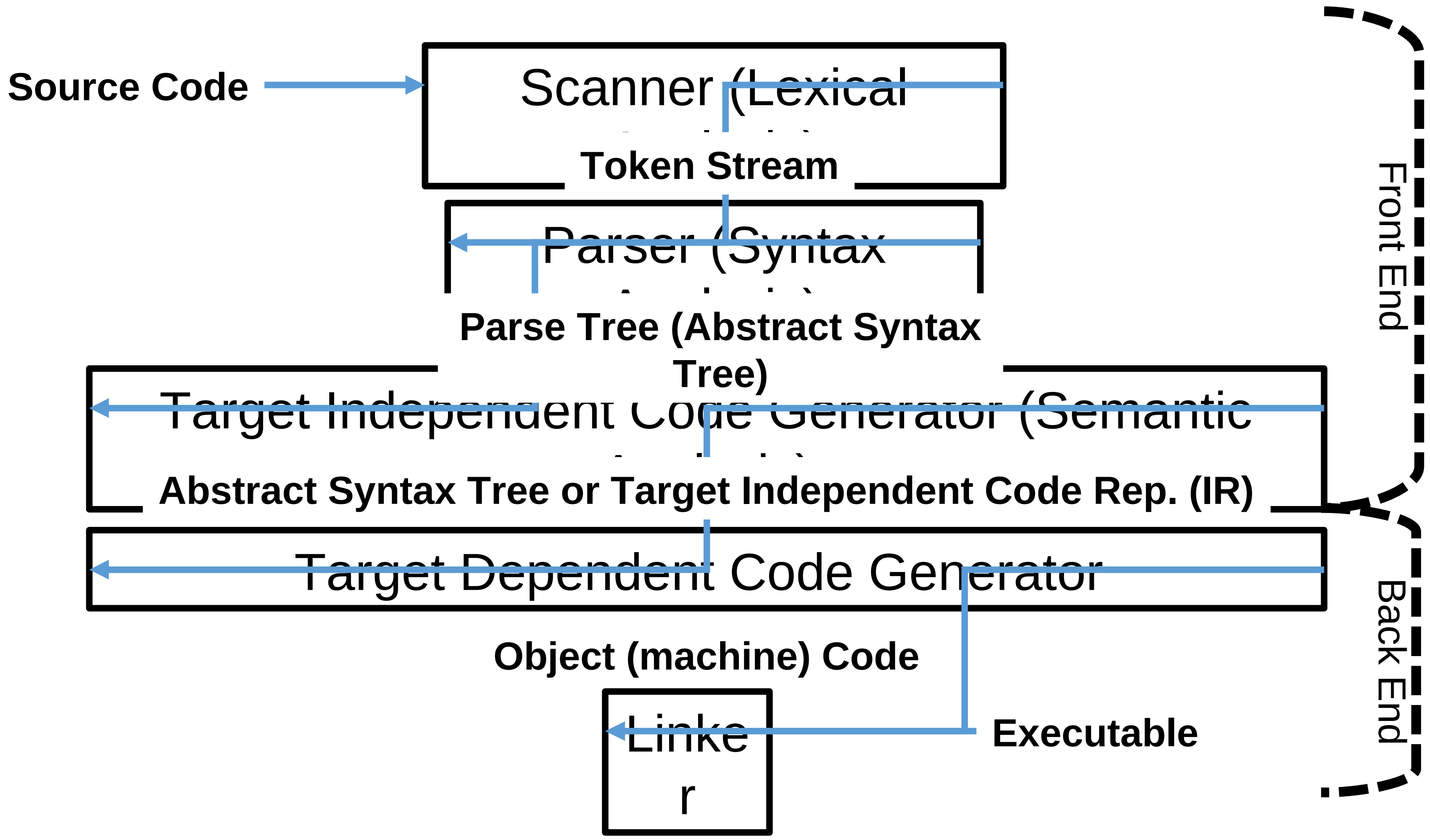
    e.g., Java, C, Fortran, etc.
  - You will get fewer errors during execution (never syntax, only due to bad data)

For the first part of the course we focus on compilation

Aside: The Best of Both Worlds?

- Just-in-time compiling (JIT; Perl, Java, etc.)
- Include interpreter in executable
- Late binding and dynamic features
- Attempt at producing greater portability

Phases of Compilation


Source Code
Scanner (Lexical
Token Stream
Parser (Syntax
Parse Tree (Abstract Syntax Tree)
Target Independent Code Generator (Semantic
Abstract Syntax Tree or Target Independent Code Rep. (IR)
Target Dependent Code Generator
Object (machine) Code
Linke
r
Executable
Front End
Back End

Preprocessing

- A lot of the code we write cannot be compiled!
- Many languages have "preprocessing" to produce a compilable file
- Examples:
  - Macros: C, C++ (#define), Lisp
  - Conditional compilation: C, C++ (#ifdef)
  - Generics: C++, Java
  - Embedded code: Oracle SQL Pre-compiler
  - Ad hoc scripts: Perl on GCC files during Make

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 ("words" or Lexemes)
  - E.g., keywords, literals, identifiers, punctuation
- Strip out items ignored by the compiler
  - E.g., white space and comments
- Flag any unknown tokens (or characters) 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
```

$\boxed{\text{def}}\ \boxed{\text{gcd}}\ \boxed{(}\ \boxed{\text{m}}\ \boxed{,}\ \boxed{\text{n}}\ \boxed{)}\ \boxed{:}\ \boxed{\text{while}}\ \boxed{\text{m}}\ \boxed{\text{!=}}\ \boxed{\text{n}}\ \boxed{:}$

$\boxed{\text{if}}\ \boxed{\text{m}}\ \boxed{>}\ \boxed{\text{n}}\ \boxed{:}\ \boxed{\text{m}}\ \boxed{=}\ \boxed{\text{m}}\ \boxed{-}\ \boxed{\text{n}}\ \boxed{\text{else}}\ \boxed{:}$

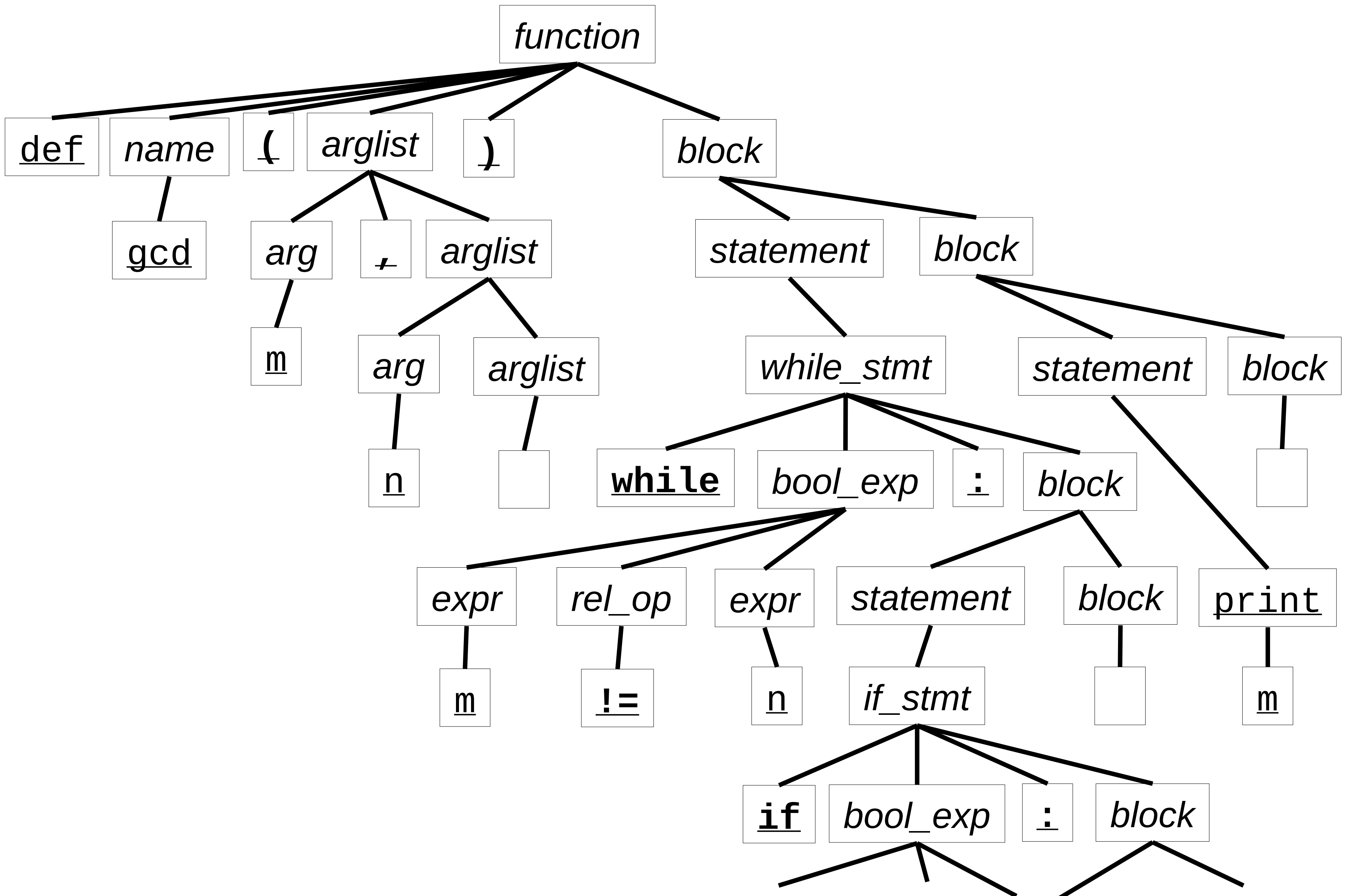
$\boxed{\text{n}}\ \boxed{=}\ \boxed{\text{m}}\ \boxed{-}\ \boxed{\text{n}}\ \boxed{\text{print}}\ \boxed{\text{m}}$

Note: Some tokens have values (e.g., identifiers, constants)

Syntax Analysis

- Organize tokens into a parse tree according to language grammar (i.e., the language's "syntax")
  - 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
- For efficiency, most parsers go directly to an Abstract Syntax Tree (AST) and don't bother with parse trees (since they are so similar)

Example: Parse Tree

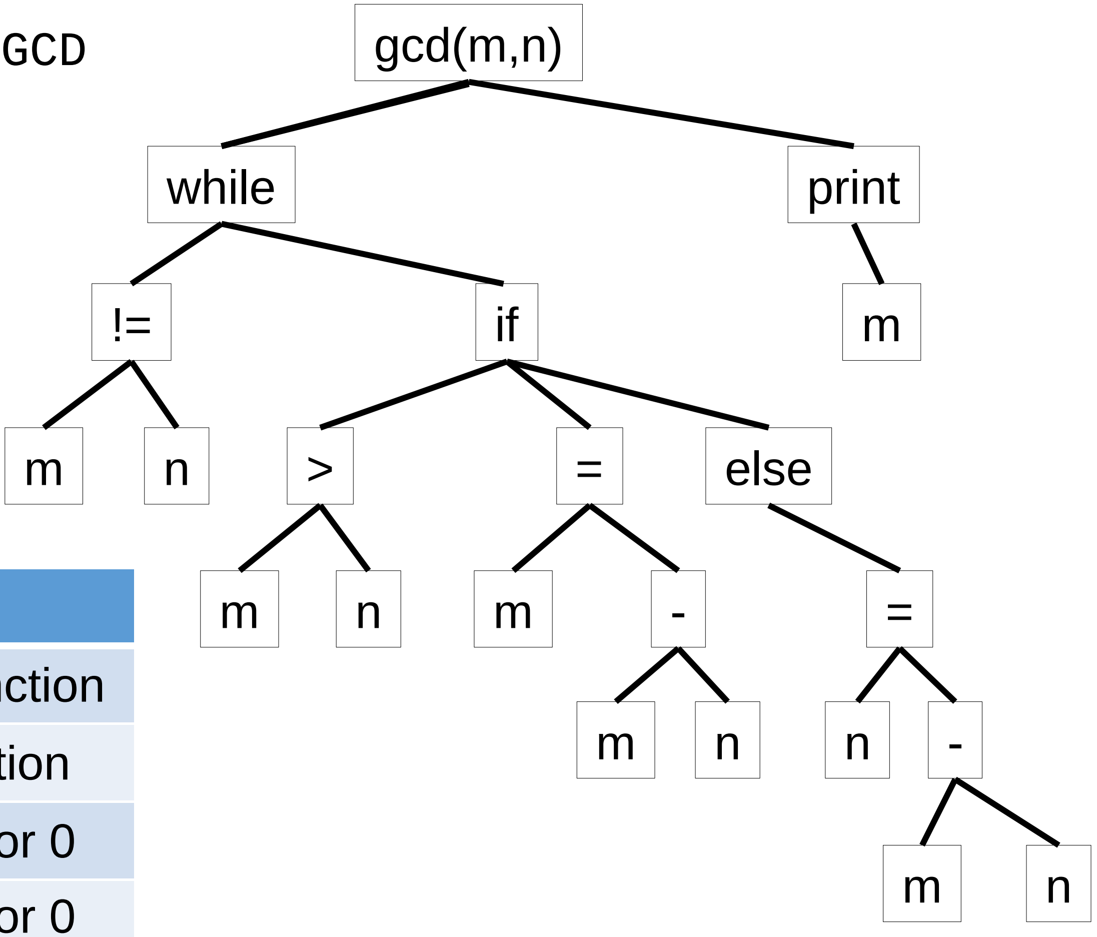


Semantic Analysis / Code Generation

- Generate symbol table of all identifiers (i.e., names)
- Ascribe meaning to all identifiers
- Condense syntax tree to only important nodes (This is usually done during parsing)
- Generate intermediate representation for each node
- Ensure the 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:
```



Symbol Table

Index	Symbol	Properties
0	gcd	user-def function
1	print	built-in function
2	m	parameter for 0
3	n	parameter for 0

Example: Code Generation

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

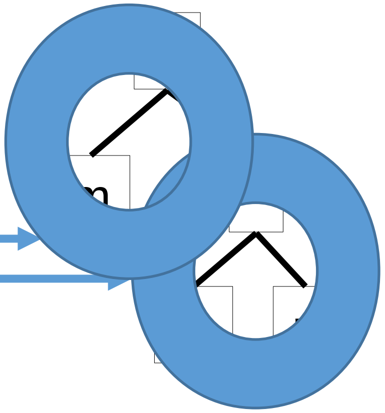
    mov idx2 → r1

    mov idx3 → r2

    sub r1, r2 → r3

    mov r3 → idx4
  - For node: m =

    mov idx4 → r1



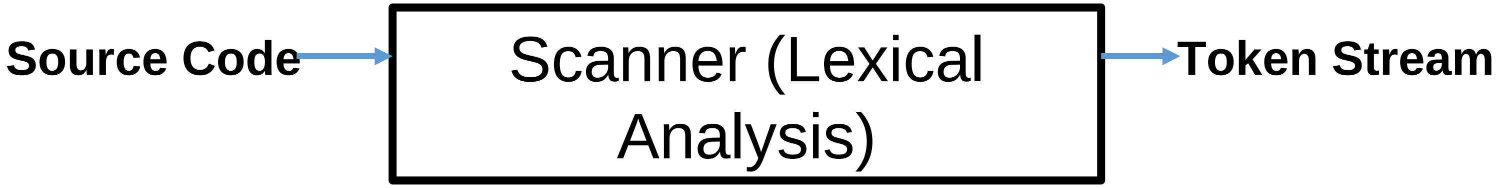
Index	Symbol	Properties
2	m	parameter
3	n	parameter
4	tmp1	temporary var

The Full Translation Process

1. Lexical Analysis
2. Syntax Analysis (Parsing)
3. Conversion to Abstract Syntax Tree
4. Semantic Analysis
5. Conversion to Intermediate Representation (IR; e.g., RTL)
6. Optimisation of IR (This is most of what makes compilers complicated)
7. Code Generation and Register Allocation
8. Code Optimisation

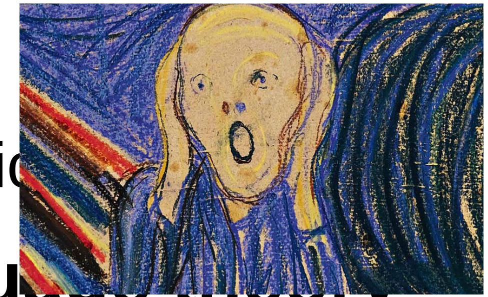
Where do we start?

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



How Do We Build a Scanner?

- Need to be able to
  - **Specify unambiguously the tokens of a language!**
  - Build a scanner from the specification (programming)
  - 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 specific
- We now need some **formal langu**



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

Thus ...

- A Language is a set of strings
- A string is a sequence of characters
- The characters come from the alphabet of the language
- ε is a special string called "the empty string"

More Definitions

- |σ| denotes the length of σ (# of chars)

  Examples

  - |ε| = 0
  - |1001001| = 7
  - |tami| = 4
  - |yyz| = 3

ST = {st|s E S,t E T}

- Σ^i denotes languages (sets of strings) of length i over Σ
- Example: If Σ = {a, b}
  - $\Sigma^0 = \{\varepsilon\}$
  - $\Sigma^1 = \Sigma = \{a,b\}$

Examples of Languages

- Finite Languages
  - Σi for any fixed i
  - English words in the OED
  - Java tokens
- Infinite Languages
  - ΣX
  - Set of all English sentences
  - Set of all Java programs
  - $\{0n \mid n \geq 0\}$
  - $\{0n1n \mid n \geq 0\}$

I will listen to my friends next time... I will listen to...
I will listen to my friends next time... I will listen to...
I will listen to my friends next time... I will listen to...
I will listen to my friends next time... I will listen to...
I will listen to my friends next time... I will listen to...
I will listen to my friends next time... I will listen to...
I will listen to my friends next time... I will listen to...
I will listen to my friends next time... I will listen to...
I will listen to my friends next time... I will listen to...
I will listen to my friends next time... I will listen to...
I will listen to my friends next time... I will listen to...

Types of Languages (Chomsky Hierarchy of

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.

Ummm … What’s a "regular" language?

Regular Languages

- Regular languages are the "simplest" languages
- Limited in what they can express – can't express everything
- Can be infinite
- Can easily build a "recogniser" for the language (a device/tool/program that determines if a set of characters – usually from the alphabet – is a member of the language)

Definition of Regular Languages

We use a recursive Definition ...

- Base Cases:
  - A (Empty language)
  - {ε} (Language consisting of the empty string)
  - {a}, a Ε Σ (Language consisting of one symbol)

  All base cases are all regular languages
- Inductive Step: If L1 and L2 are regular then so are
  - L1L2 = {στ | σ Ε L1,τ Ε L2} *concatenation*
  - L1 ν L2 = {σ | σ Ε L1 λ σ Ε L2} *union*
  - L1Χ = {σi | σΕL1, i ≥ 0} *Kleene closure*

Examples (Regular)

- {a,ab,abc}
- Any finite language
- {a}X
- {a,b,c}X
- {1n0 | n > 0}
- Set of all positive integers (base 10)
- {ΣΣ*@ ΣΣ*(. ΣΣ*)n| Σ = {a,b,c,...z},n ≥ 0}

Hmmm, it seems like a lot of stuff is "regular"

Examples (NOT Regular)

Non-regular languages

- {aibjck | i > 0, j > i, k > j}
- {0n1n | n ≥ 0}
- {ap | p E PRIMES}
- Set of all correct Java programs

Hints:

- Non-regular languages might need "memory" in the description
- Non-regular languages might need "context"