INTRODUCTION

Based on Chapter 1 of Aho, Lam, Sethi, Ullman:

Compilers: Principles, Techniques, & Tools

2nd Ed, Addison Wesley, 2007

Table of Contents

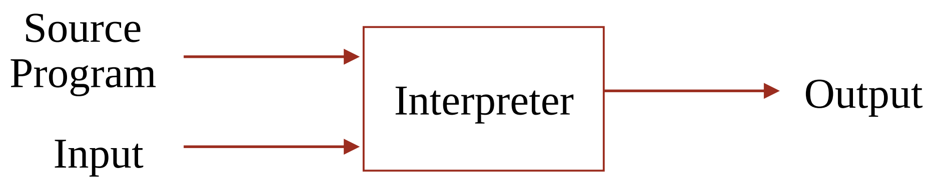
Language Processors - Compilers

- Compilers are computer programs that translate one language to another
- Source language and target language

- Variety of source languages and target languages and variety of compilers
  - the basic tasks that any compiler must perform are essentially the same

Language Processors - Interpreters

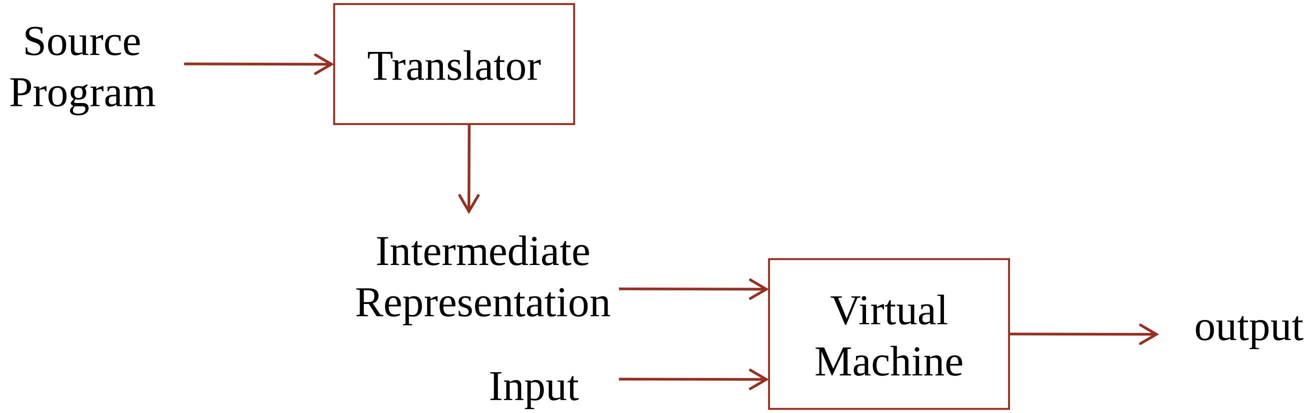
- An interpreter is a program that reads a source program and inputs and directly executes it without translating it into machine language



- Interpreters are more likely to support
  - interactive execution
  - dynamic typing
  - better error diagnostics
  - but, suffer from slower execution than machine-language programs

Language Processors – Hybrid Approaches

- Hybrid approaches combine compilation and interpretation

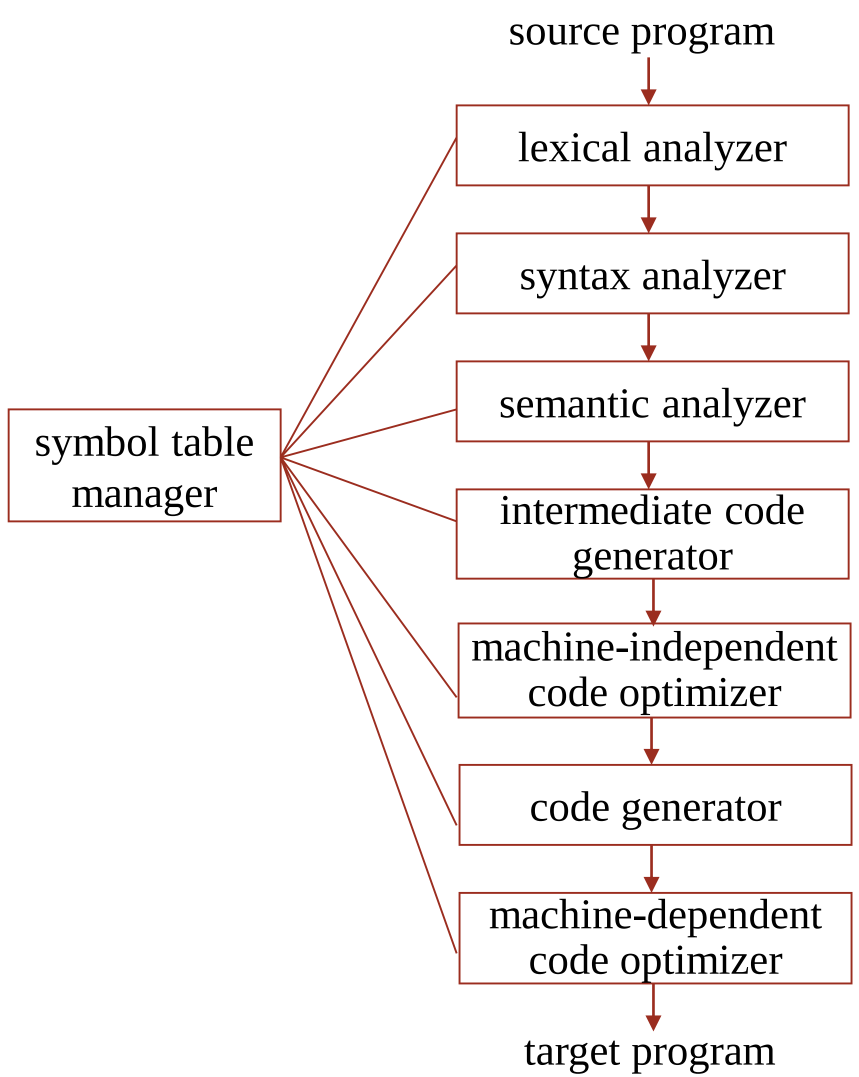


- Intermediate representation(e.g. Java bytecode) can be interpreted by another machines (e.g. Java Virtual Machine)
- Slow execution speed can be complemented by just-in-time compilation

Programs Related to Compilers

- Assemblers
- Linkers
- Loaders
- Preprocessors
- Editors (esp. Syntax-Directed Editors)
- Debuggers
- Project Managers
- Integrated Development Environment
- etc.

The Logical Structure of a Compiler



Lexical Analysis

- Lexical analyzer(or scanner) divides input string into a sequence of smallest meaningful units(*lexemes*) and produces as output *tokens* of the form <*token_name, token_attribute*>
- Example

  position = initial + rate * 60

lexemes	tokens	
position	<id, 1>	// id : token name for identifiers, 1 : token value
=	<=,>	// = : token name for operator =
initial	<id, 2>	
+	<+,>	
rate	<id, 3>	
*	<*,>	
60	<num, 60>	// num : token name for numbers

  - white spaces and comments are eliminated during lexical analysis

Lexical Analysis

- Lexical analysis may perform management of *symbol table* and *literal table*
  - example

1	position	...
2	initial	...
3	rate	...
...	...	...

  - literal
    - number : 3.141592
    - string : "Hello World"

- *will be discussed in Chapter 3*

Syntax Analysis

- Syntax analyzer or parser performs syntax analysis (or parsing)
  - to group tokens into grammatical phrases, e.g. expression, statement, etc.
  - to generate a *parse tree* or a *syntax tree*
- Context Free Grammar
  - describes well-formed programs by *recursive rules*
  - example: expression
    1. any identifier is an expression
    2. any number is an expression
    3. if exp1 and exp2 are expressions, then so are

       exp1 + exp2

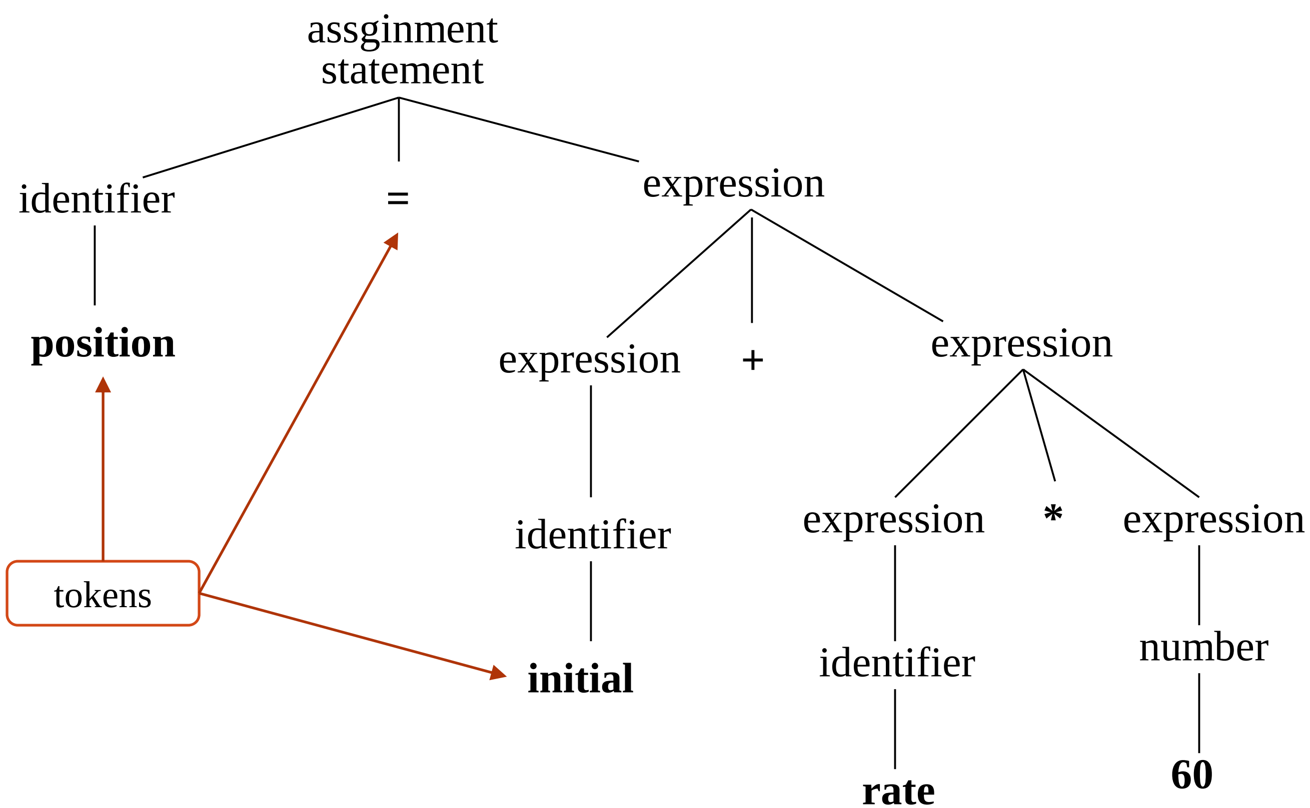
       exp1 * exp2

       (exp1)
  - example: statement
    1. id = expression
    2. while (expression) statement
    3. if (expression) statement

Context Free Grammar is a formalization of recursive rules

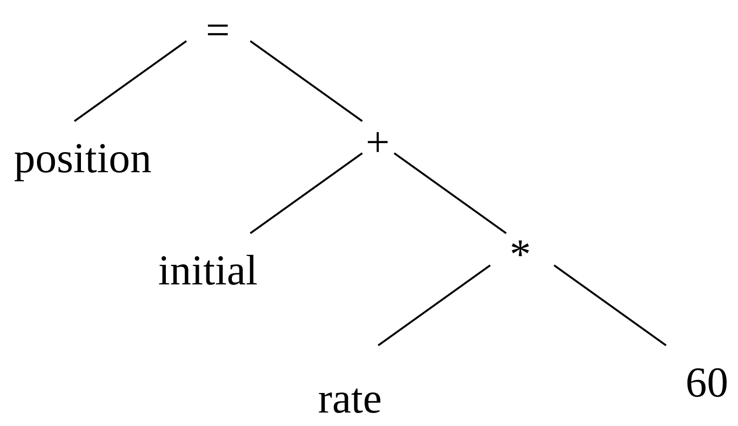
Syntax Analysis

- Parse tree
  - hierarchical representation of a program



Syntax Analysis

- Syntax tree
  - a compressed representation of the parse tree
  - a more common internal representation of syntactic structure



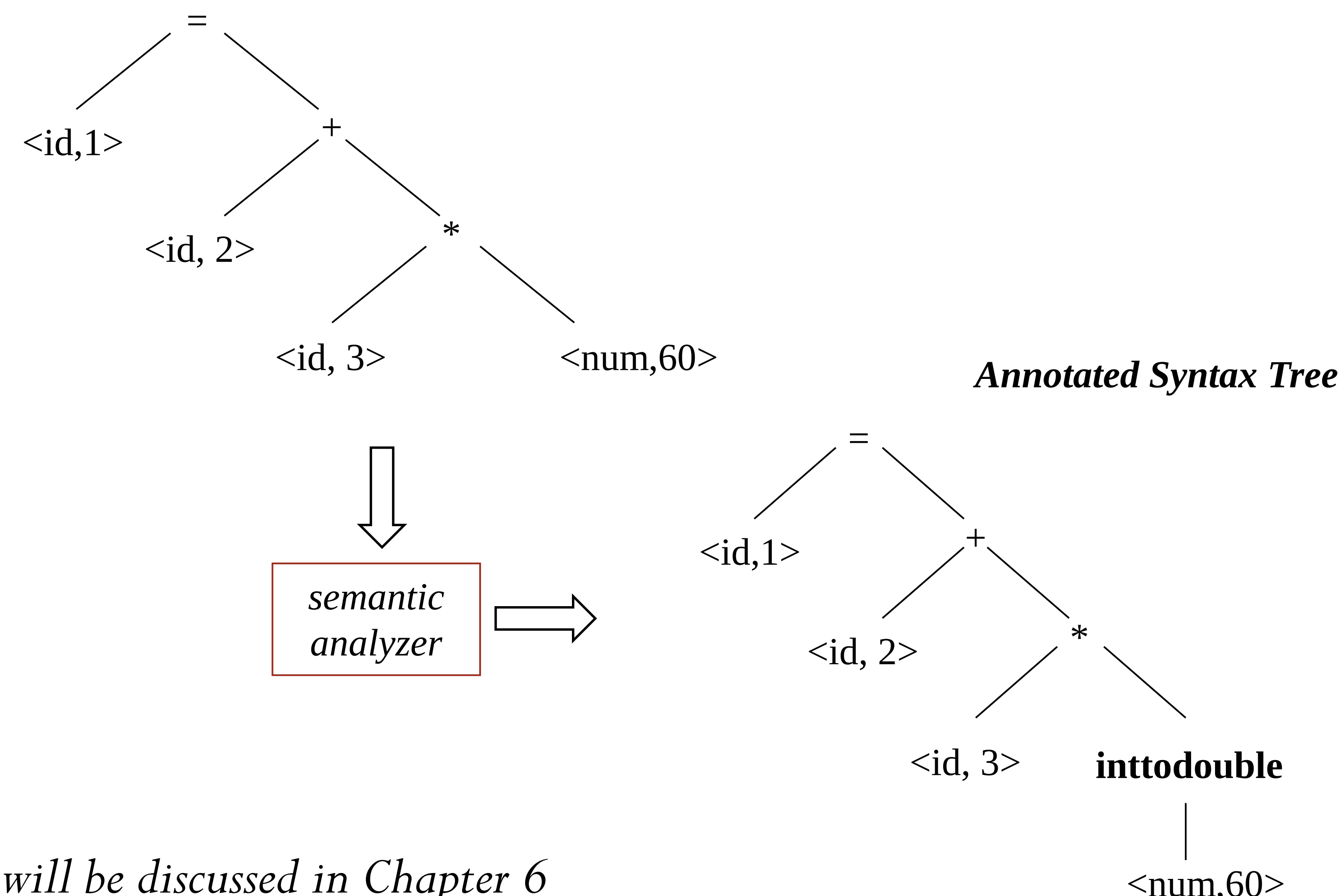
- *will be discussed in Chapter 4*

Semantic Analysis

- Semantic analyzer uses the syntax tree and symbol table to check the program for *semantic consistency* with language definition
  - aspects of semantics where the behavior of a program can be predicted at compile-time (*static semantics* or *context-sensitive syntax*)
    - type checking
    - some special control (e.g break)
    - name uniqueness (e.g. identifier uniqueness)
- Semantic analysis performs gathering additional information (attributes) suchs as types, checking semantic error, and some related work
  - type checking and automatic type conversion is the most important work
    - type error exmaple - "string"/2
    - coercions example - rate * 60   // assumes the type of *rate* is double

Semantic Analysis

- Semantic analyzer produces an annotated syntax tree



- *will be discussed in Chapter 6*

Intermediate Code Generator

- Intermediate code
  - a program of an abstract machine
  - should be easy to produce and easy to translate into target program
- A variety of forms
  - three address code
  - stack machine code (e.g. P-code, U-code, Java bytecode)
  - syntax tree
- Example: three address code
  - at most one operator on the right side
  - temporary names to hold values computed
  - may have fewer than three operands

```
temp1 = inttodouble(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

- *will be discussed in Chapter 6*

Code Optimization

- Code optimization improves intermediate code to result in *better* target code
  - faster, shorter, or less power
- Various optimization points and techniques
  - folding
  - common subexpression elimination
  - dead code elimination
  - loop optimization
  - etc.
- Example: folding

```
temp1 = inttodouble(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

⇨

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

- Machine dependent vs independent optimization (*ref. Chapter 8 & 9*)

Code Generation

- Code generator takes the intermediate code and generates machine code or assembly code for target machine
  - machine instructions
  - data representation
  - registers
  - storage allocation
- Example

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

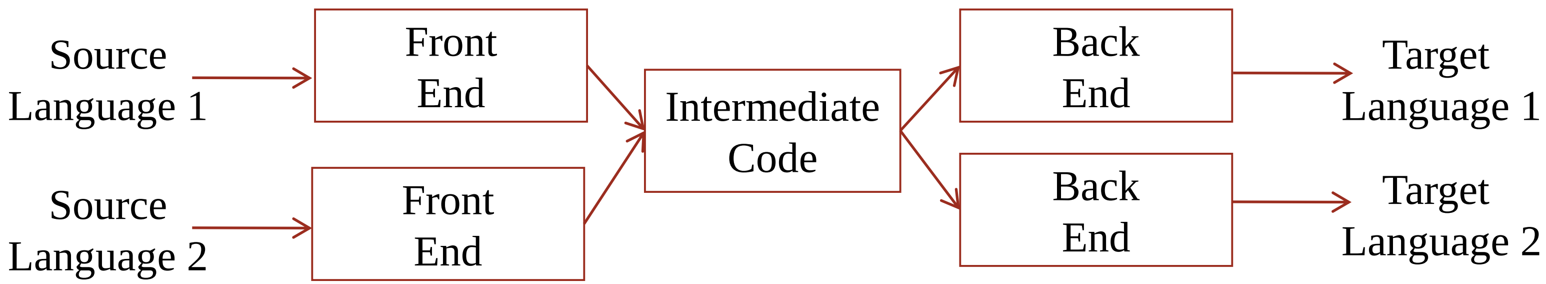
- *Ref. Chapter 8*

Major Data Structures

- Tokens
- Syntax Tree
- Annotated Syntax Tree
- Symbol Table
- Literal Table
- Intermediate Code
- Temporary Files

The Physical Structure of Compiler

- Analysis and synthesis
  - Analysis part of compiler
    - lexical analysis, syntax analysis, semantic analysis, optimization
  - Synthesis part
    - intermediate code generation, code generation, optimization
  - Analysis part has been studied more mathematically
- Front-end and back-end



  - increases compiler portability

The Physical Structure of Compiler

- Passes
  - one pass consists of reading an entire program and writing an output file
  - several phases can be grouped into one pass
    - trade off between efficient compilation and efficient object code
  - examples
    - one-pass compiler
    - two-pass compiler
      - pass one: from lexical analysis to intermediate code generation
        - parser plays an major role
      - pass two: code generation
    - three-pass compiler
      - pass three: code optimization
    - etc.

Compiler Construction Tools

- Parser generator – yacc
- Scanner generator – lex
- Syntax-directed translation engines
- Code-generator generators
- Data-flow analysis engines (key part of code optimization)
- Compiler-construction toolkits – LLVM

Applications of Compiler Technology

- Implementation of High-Level Programming Languages
- Optimization for Computer Architectures
  - Parallelism
  - Memory hierarchy
- Design of New Computer Architectures
  - RISC
  - Specialized Architectures: VLIW, SIMD, vector machines, etc.
- Program Translations
  - Binary translation
  - Hardware synthesis
  - Database query interpreter
  - Compiled simulation
  - Translation of non-programming languages(HTML, XML, Postscript, etc)

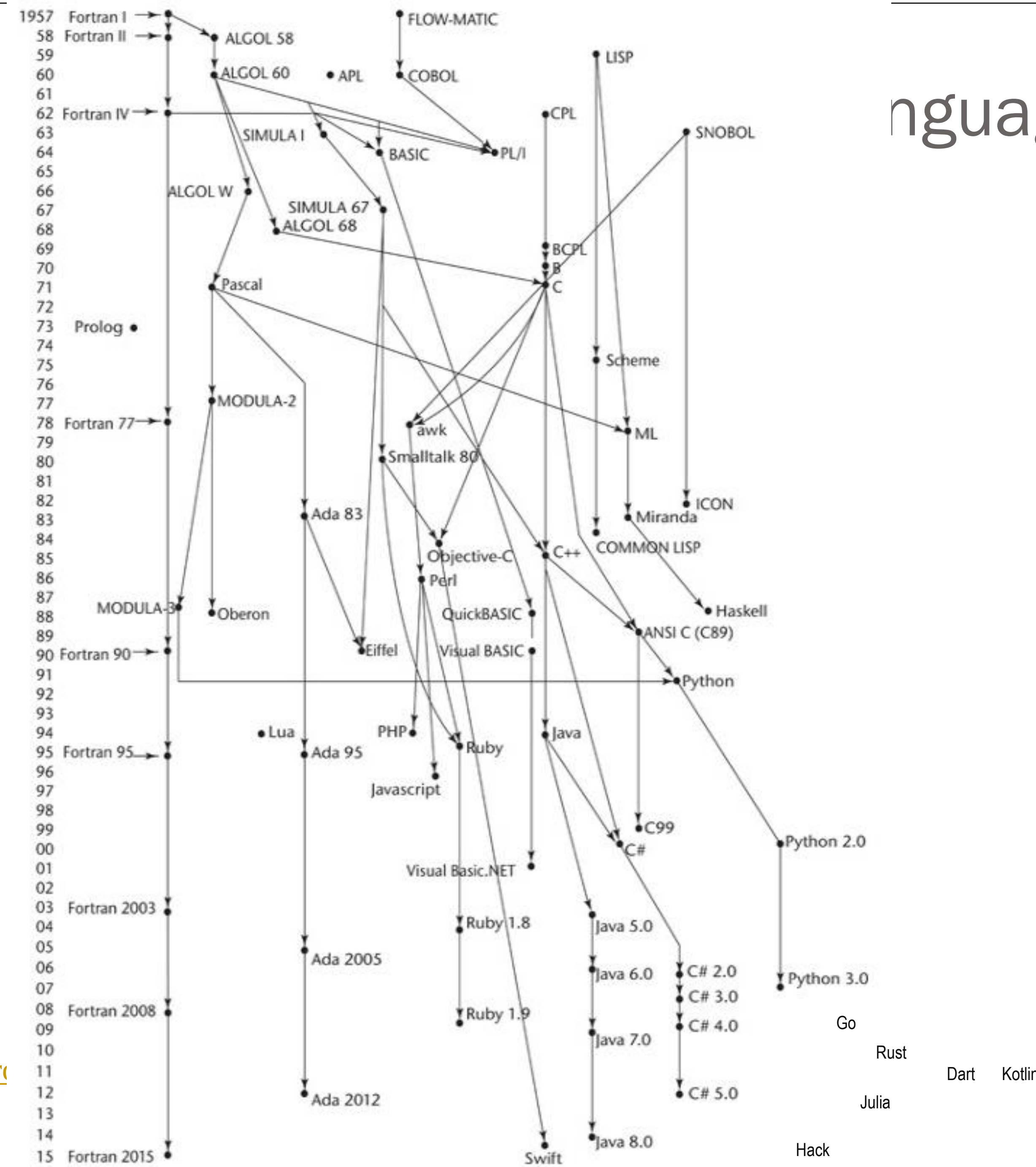
Applications of Compiler Technology

- Software Productivity Tools
  - Type checking and type inference
  - Dataflow analysis for checking errors or optimizing code
  - Static analysis for security
  - Static analysis for generating test data
  - Memory management tools
  - Syntax-directed editors

The Evolution of Programming Languages

- Programming languages has been related to the concept of the von Neumann Architecture (stored program)
- Classification by Generation
  - First-generation : machine languages
  - Second-generation : assembly languages
  - Third-generation : high-level languages (Fortran, COBOL, C, C++, etc)
  - Fourth-generation : application specific languages (SQL,NOMAD, etc)
  - Fifth-generation : languages for intelligent systems (Prolog, OPS5, etc)
- Classification by Programming Styles
  - Imperative vs Declarative
  - Procedural, Object-Oriented, Logic, Rule-based, Functional, Script Languages

The nguages



TIOBE Pr

The Evolution of Programming Languages

- Compiler technology has been largely affected by programming language evolution
- Assembly language and assembler
- High-level language and compiler
  - FORTRAN mid 1950s by John Backus, IBM
- Formal grammars and languages
  - Chomsky hierarchy – type 0, 1, 2, 3 – late 50s
  - Context free grammar
  - Algol 60, first language described in CFG
  - Automata – 60s and 70s
- Compiler generators
  - parser generator – yacc(1975, S. Johnson, AT&T Bell L.)
  - scanner generator – lex(1975, M. Lesk, AT&T Bell Lab.)

Basic Concepts of Programming Languages

- Data Types and Variable Declarations
  - static binding vs dynamic binding
  - local variables vs global variables
- Scope
  - static scope and block structure
  - dynamic scope
- Expressions
- Control Statements
- Subprograms and Parameter Passing
  - functions and procedures
  - call-by-value vs call-by-reference
  - generic subprograms
- Advanced concepts in other paradigms(e.g. object-oriented)

References

- Aho, Lam, Sethi, Ullman: *Compilers: Principles, Techniques, & Tools,* 2nd Ed, Addison Wesley, 2007
- Robert W. Sebesta, *Concepts of Programming Languages*, 12th Ed., Pearson, 2018.