Organization of Programming Languages
CS 3200/5200D

Lecture 01

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

What is an algorithm?

- An **algorithm** is a computational procedure that takes values as *input* and produces values as *output*, in order to solve a well defined *computational problem*:
  - The statement of the problem specifies a desired relationship between the *input* and the *output*.
  - The algorithm specifies how to achieve that *input/output* relationship.
  - A particular value of the *input* corresponds to an instance of the *problem*.

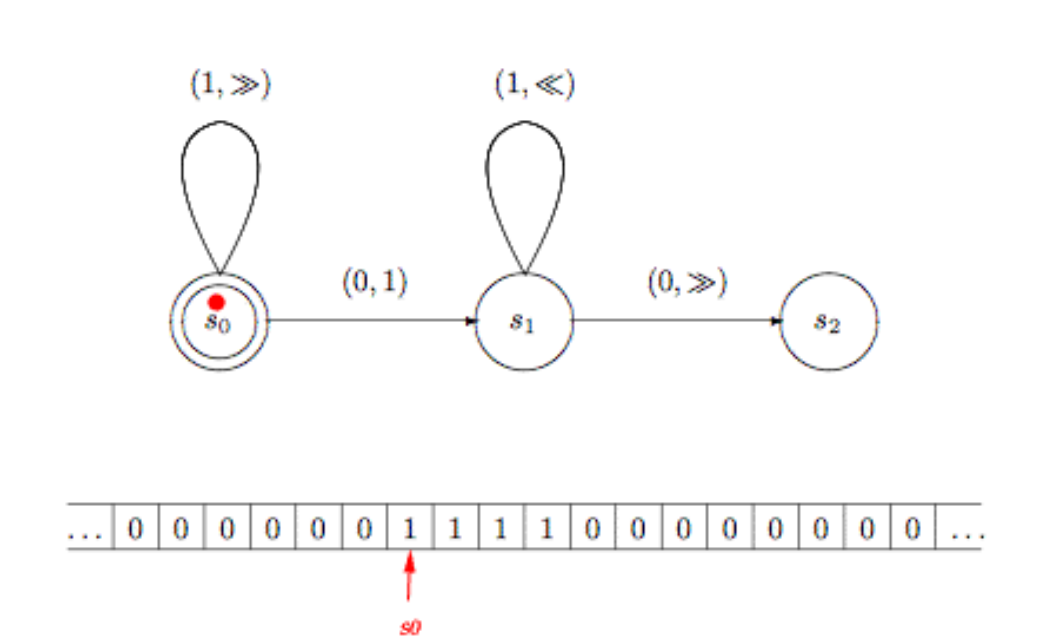
What is a programming language?

- A **programming language** is an artificial language designed for expressing algorithms on a computer:
  - Need to *express* an infinite number of algorithms (**Turing complete**).
  - Requires an unambiguous **syntax**, specified by a *finite* context free grammar.
  - Should have a well defined compositional **semantics** for each syntactic construct: *operational vs. axiomatic vs. denotational*.
  - Often requires a practical implementation i.e. **pragmatics**:
    - Implementation on a real machine vs. virtual machine
    - *translation vs. compilation vs. interpretation.*

Turing Machines

- An infinite one-dimensional **tape** divided into *cells*:
  - each cell may contain one symbol, either 0 or 1.
- A read-write **head** that can also move *left* or *right*.
- A set of **transition rules** (the program):
  - tuples $\langle State_{current},\ Symbol,\ State_{next},\ Action \rangle$
  - 3 possible actions:
    - *write* a symbol in the current cell on the tape.
    - *move* the head one cell to the *left* or *right*.
  - machine *halts* when no transition rule is specified for current situation.

Turing Machines



http://plato.stanford.edu/entries/turing-machine

Outline

- Reasons for Studying Programming Languages
- Influences on Language Design
- Language Paradigms
- Implementation Methods
- Overview of Compilation

Reasons for studying concepts of PLs

- Increased ability to express ideas/algorithms
  - Natural language:
    - The depth at which people can think is influenced by the expressive power of the language they use (also Sapir-Worf hypothesis).
    - http://fora.tv/2010/10/26/Lera_Boroditsky_How_Language_Shapes_Thought
  - Programming languages:
    - The complexity of the algorithms that people implement is influenced by the set of constructs available in the programming language.
    - Kenneth E. Iversion (inventor of APL programming language):
      - "Notation as a tool of thought" (Turing award lecture).

Notation and Complexity

```
int[] a = {2, 6, 1, 9, 7};
int[] b = {11, 4, 8, 2};

for (int i = 1; i < a.length; i++) {
    int key = a[i];
    int k = i – 1;
    while (k >= 0 && a[k] > key) {
        a[k+1] = a[k]
        k = k – 1;
    }
    a[k+1] = key;
}
```

```
for (int i = 1; i < b.length; i++) {
    int key = b[i];
    int k = i – 1;
    while (k >= 0 && b[k] > key) {
        b[k+1] = b[k]
        k = k – 1;
    }
    b[k+1] = key;
}
```

Reasons for studying concepts of PLs

- Improved background for choosing appropriate languages:
  - Many programmers use the language with which they are most familiar, even though poorly suited for the new project
  - Ideal: use the most appropriate language.
  - Features important for a given project are included already in the language design, as opposed to being simulated => elegance & safety.

Reasons for studying concepts of PLs

- Increased ability to learn new languages:
  - Once fundamental concepts are known, new languages are easier to learn (recognize same principles as incorporated in the new language).
  - Example:
    - Knowing the concepts of Object Oriented Programming (OOP) makes learning Java significantly easier.
    - Knowing the grammar of your native language makes it easier to learn another language.

Reasons for studying concepts of PLs

- Better understanding of significance of implementation:
  - Example: a small subprogram that is called very frequently can be a highly inefficient design choice.
  - More details can be learned by studying compiler design.
- Better use of languages that are already known.
- Overall advancement of computing:
  - Algol 60 vs. Fortran.

Outline

- Reasons for Studying Programming Languages
- Influences on Language Design
- Language Paradigms
- Implementation Methods
- Overview of Compilation

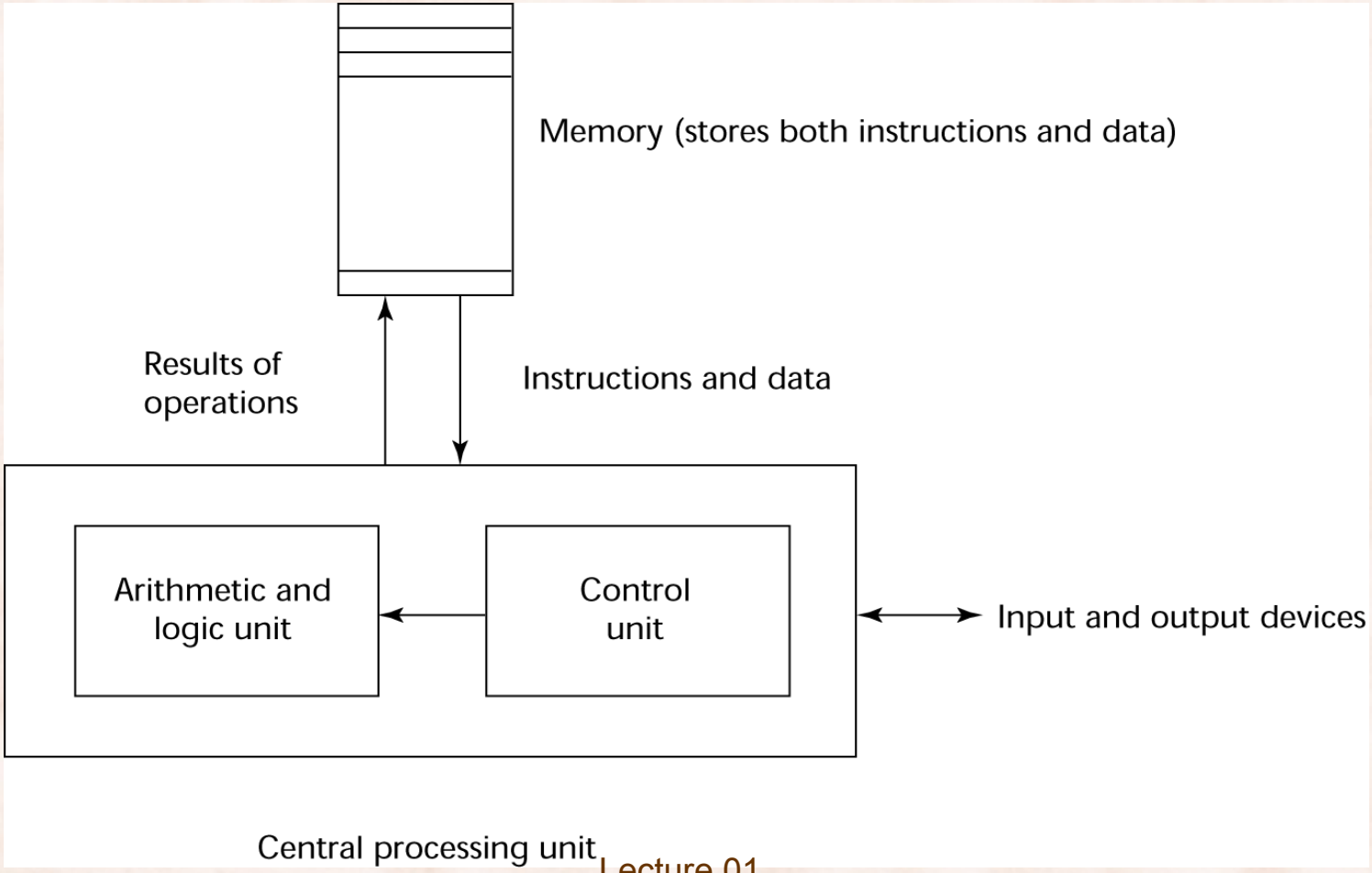
Influences on Language Design

- Computer Architecture:
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture.
- Programming Methodologies:
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, to new programming languages.
- Application Domains:
  - Scientific, Artificial Intelligence, Business, Systems, Web, …

Influences: Computer Architecture

- Most prevalent computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers:
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages:
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient

Von Neumann Architecture



Von Neumann Architecture

- Fetch-execute-cycle (on a von Neumann architecture computer)

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer.
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a *bottleneck*.
- Known as the *von Neumann bottleneck*; it is the primary limiting factor in the speed of computers

Influences: Programming Methodologies

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  - structured programming
  - top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented design
  - abstract data types (Simula 67)
- Middle 1980s: Object-oriented programming
  - data abstraction + inheritance + polymorphism
  - Smaltalk, Ada 95, C++, Java, CLOS, Prolog++

Influences: Application Domains

- Scientific applications
  - Large numbers of floating point computations; extensive use of arrays
  - Fortran, Matlab
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated; use of linked lists
    - LISP
  - Lately, many AI applications (e.g. statistical or connectionist approaches) are written in Java, C++, Python.

Influences: Application Domains

- Systems programming:
  - Need efficiency because of continuous use.
  - C (UNIX almost entirely written in C).

- Web Software:
  - Eclectic collection of languages:
    - markup (e.g., XHTML).
    - scripting for dynamic content:
      - *client side*, using scripts embedded in the XHTML document. Examples: Javascript, PHP.
      - *server side*, using the Common Gateway Interface. Examples: JSP, ASP, PHP.
    - general-purpose, executed on the Web server through CGI. Example: Java, C++, …

Is there value in knowing multiple languages?

[Lera Boroditsky's presentation]

- To have a second language is to have a second soul.
  - » Charlemagne, Holy Roman Emperor

- A man who knows four languages is worth four men.
  - » Charles V, Holy Roman Emperor

Is there value in knowing multiple languages?

[Lera Boroditsky's presentation]

- To have a second language is to have a second soul.
  - » Charlemagne, Holy Roman Emperor
- A man who knows four languages is worth four men.
  - » Charles V, Holy Roman Emperor
- I speak English to my accountants, French to my ambassadors, Italian to my mistress, Latin to my God, and German to my horse.
  - » Frederick the Great of Prussia

Is there value in knowing multiple languages?

- I speak English to my accountants, French to my ambassadors, Italian to my mistress, Latin to my God, and German to my horse.

  » Frederick the Great of Prussia

- I speak Matlab to my applied mathematician, Cobol to my accountant, C to my device driver writer, PHP to my web designer, Java to myself, …

  » CS3200 instructor, early 21st century

Is there value in knowing multiple languages?

- The ability to speak several languages is an asset, but the ability to keep your mouth shut in one language is priceless.

  » Anonymous

- Drawing on my fine command of language, I said nothing.

  » Robert Benchley

- A distinguished diplomat could hold his tongue in ten languages.

  » Anonymous

Outline

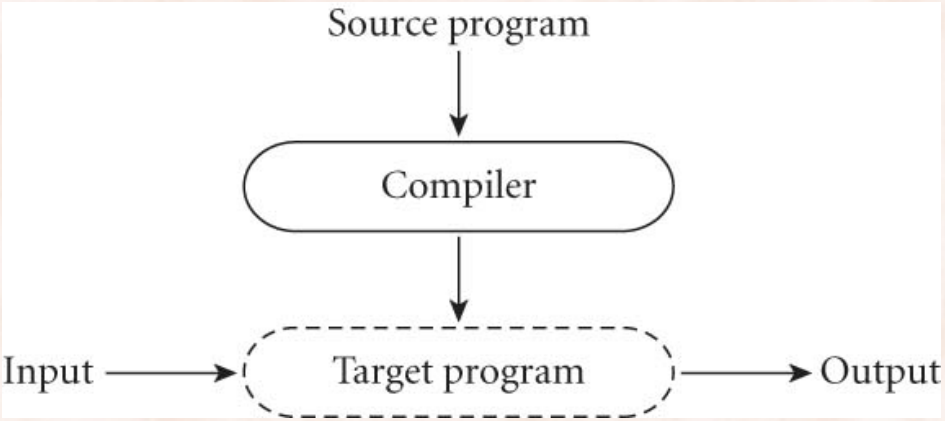
- Reasons for Studying Programming Languages
- Influences on Language Design
- Language Paradigms
- Implementation Methods
- Overview of Compilation

Implementation Methods

- Compilation:
  - Programs are translated into machine language & system calls.
- Interpretation:
  - Programs are interpreted by another program – an interpreter.
- Hybrid:
  - Programs are translated into an intermediate language that allows easy interpretation.
- Just-in-Time:
  - Hybird + compile subprograms' code the first time they are called.

Compilation

- Translates high-level program (source language) into equivalent target program (target language):
  - machine code, assembly code, byte-code, or high level language.



Interpretation

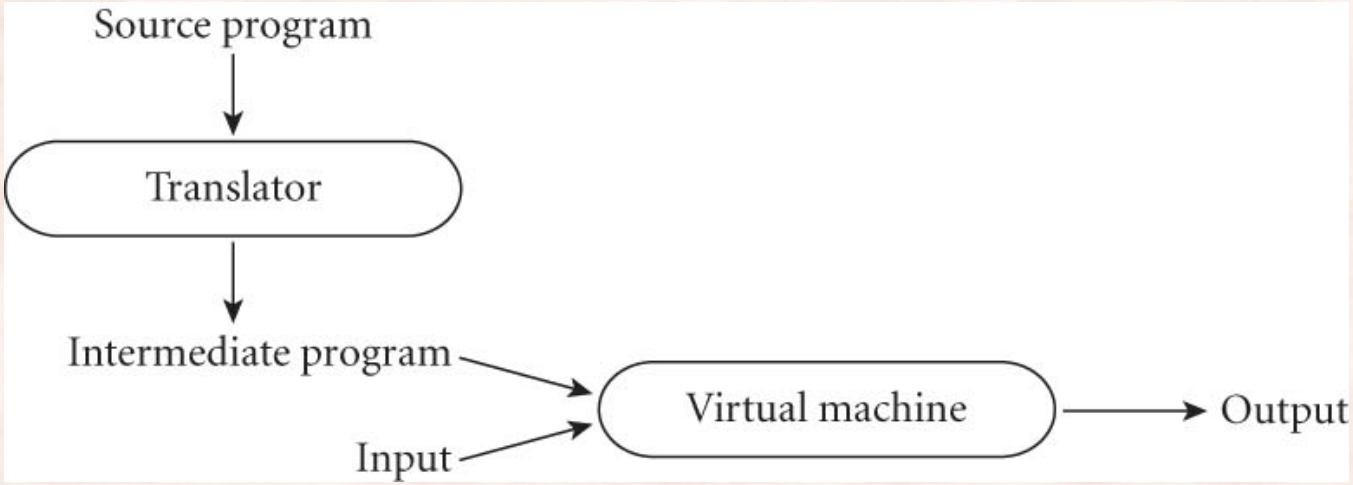
- Interpreter usually implemented as a *read-eval-print* loop:
  - **read** expression in the input language (usually translating it in some internal form).
  - **evaluates** the internal form of the expression.
  - **print** the result of the evaluation.
  - loops and reads the next input expression until exit.

Interpretation vs. Compilation

- Greater flexibility and better diagnostics:
  - run-time errors are immediately displayed.
  - excellent source-level debuggers.
  - can easily cope with languages that generate and execute code dynamically.
- Slower execution.
  - Often also requires more memory space.
- Now rare for traditional high-level languages.
  - Significant comeback with some Web scripting languages (e.g., JavaScript, PHP).

Hybrid Implementation

- A compromise between compilers and pure interpreters.
- A high-level language program is translated to an intermediate language that allows easy interpretation.
  - ⇒ Faster than pure interpretation.



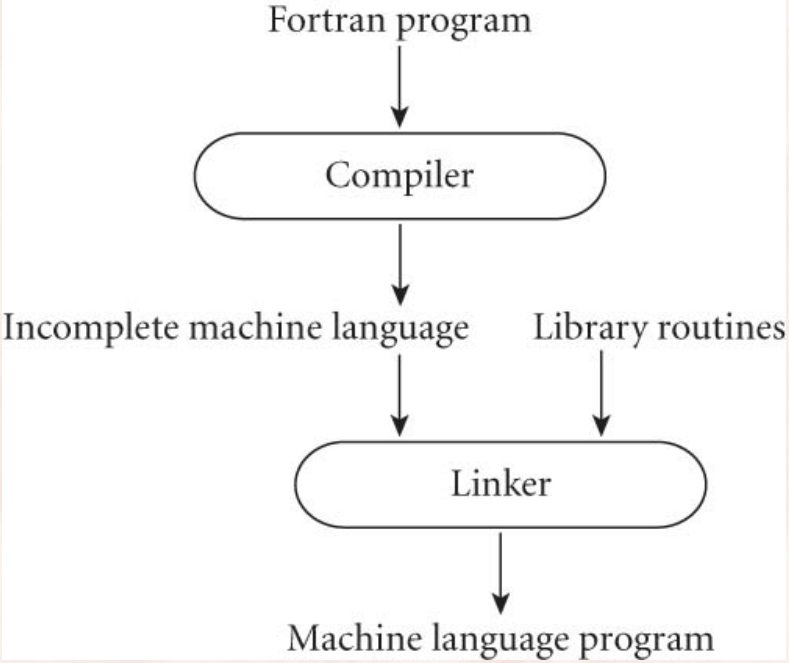
Just-in-Time Implementation

- Initially translate programs to an intermediate language.
- Then compile the intermediate language of the subprograms into machine code when they are called.
- Machine code version is kept for subsequent calls.

- JIT systems are widely used for Java programs:
  - byte-code as intermediate language.
- .NET languages (C#) are implemented with a JIT system:
  - .NET Common Intermediate Language (CIL).

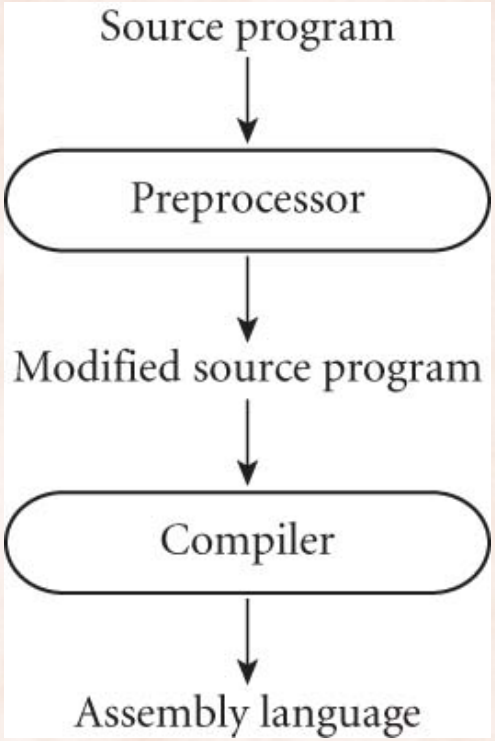
Linking

- The process of collecting system program units and other user libraries and "linking" them to a user program.



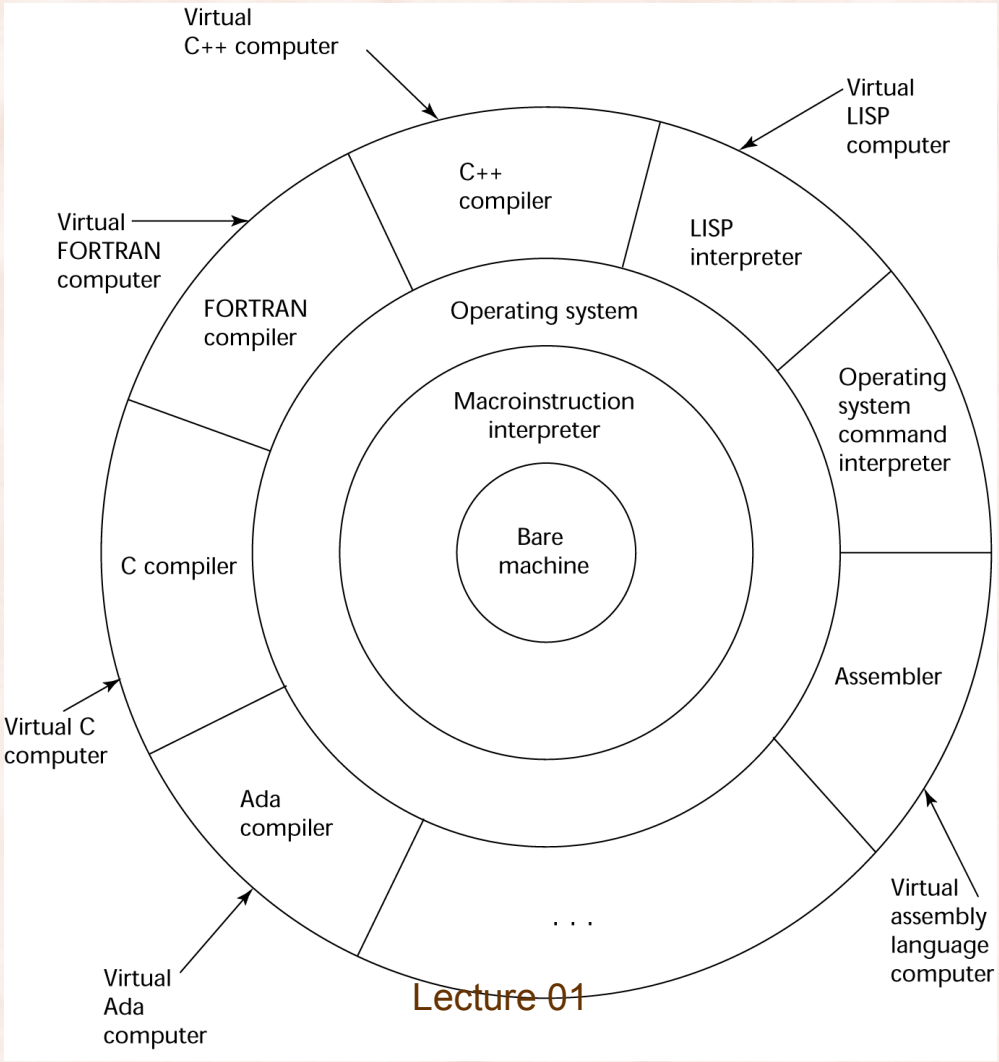
Preprocessors

- Preprocessor is run before compiler:
  - Removes comments.
  - Expands macros, commonly used to:
    - specify that code from another file is to be included;
    - define simple expressions/functions.



- C preprocessor:
  - expands `#include`, `#define`, and similar macros.
  - deletes portions of code ⇒ conditional compilation.

Layered Interface of Virtual Computers



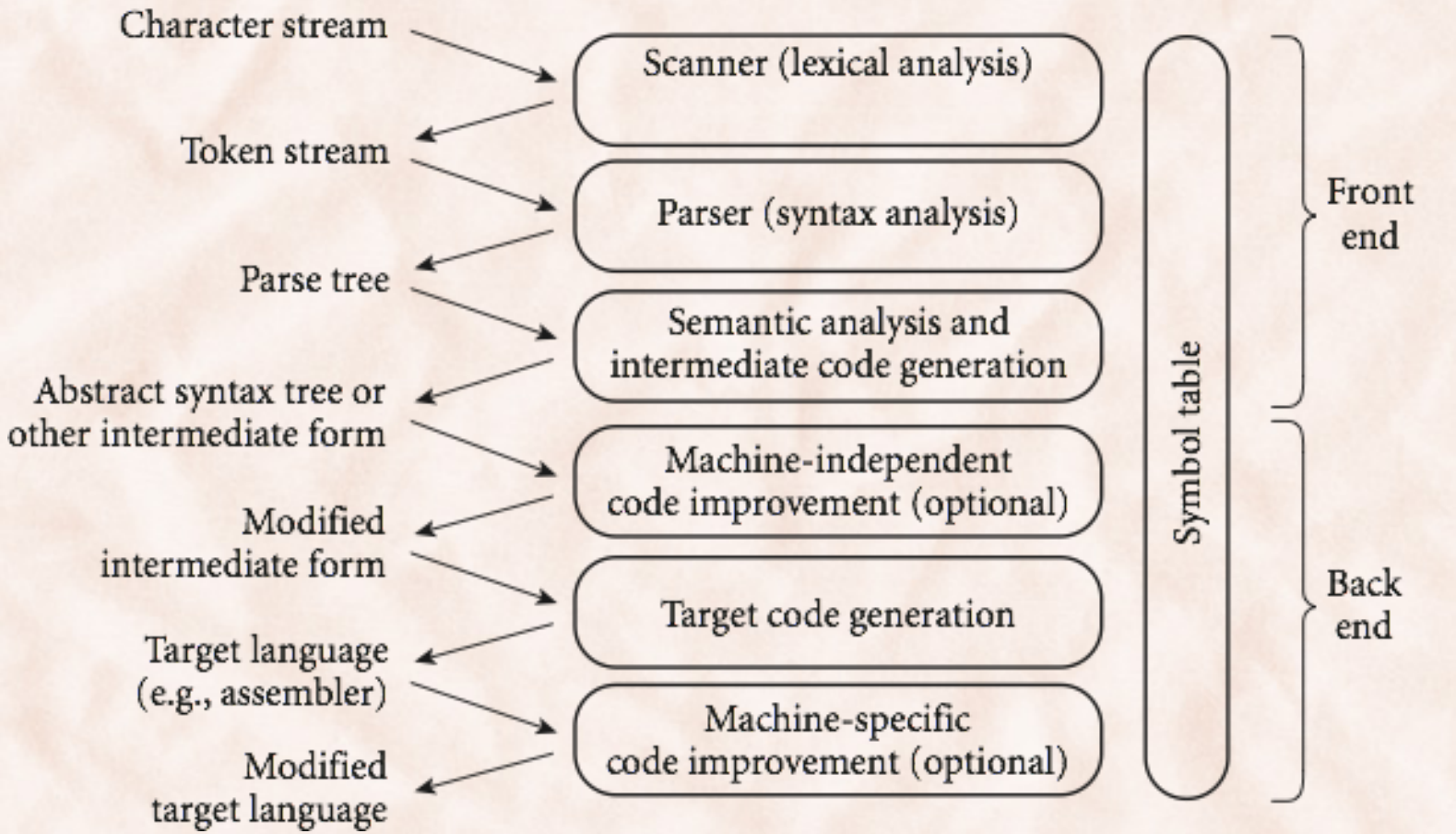
Outline

- Reasons for Studying Programming Languages
- Influences on Language Design
- Language Paradigms
- Implementation Methods
- Overview of Compilation

Sample Program: GCD

```
int main() {
   int i = getint(), j = getint();
   while (i != j) {
      if (i > j) i = i - j;
      else j = j - i;
   }
   putint(i);
}
```

Phases of Compilation

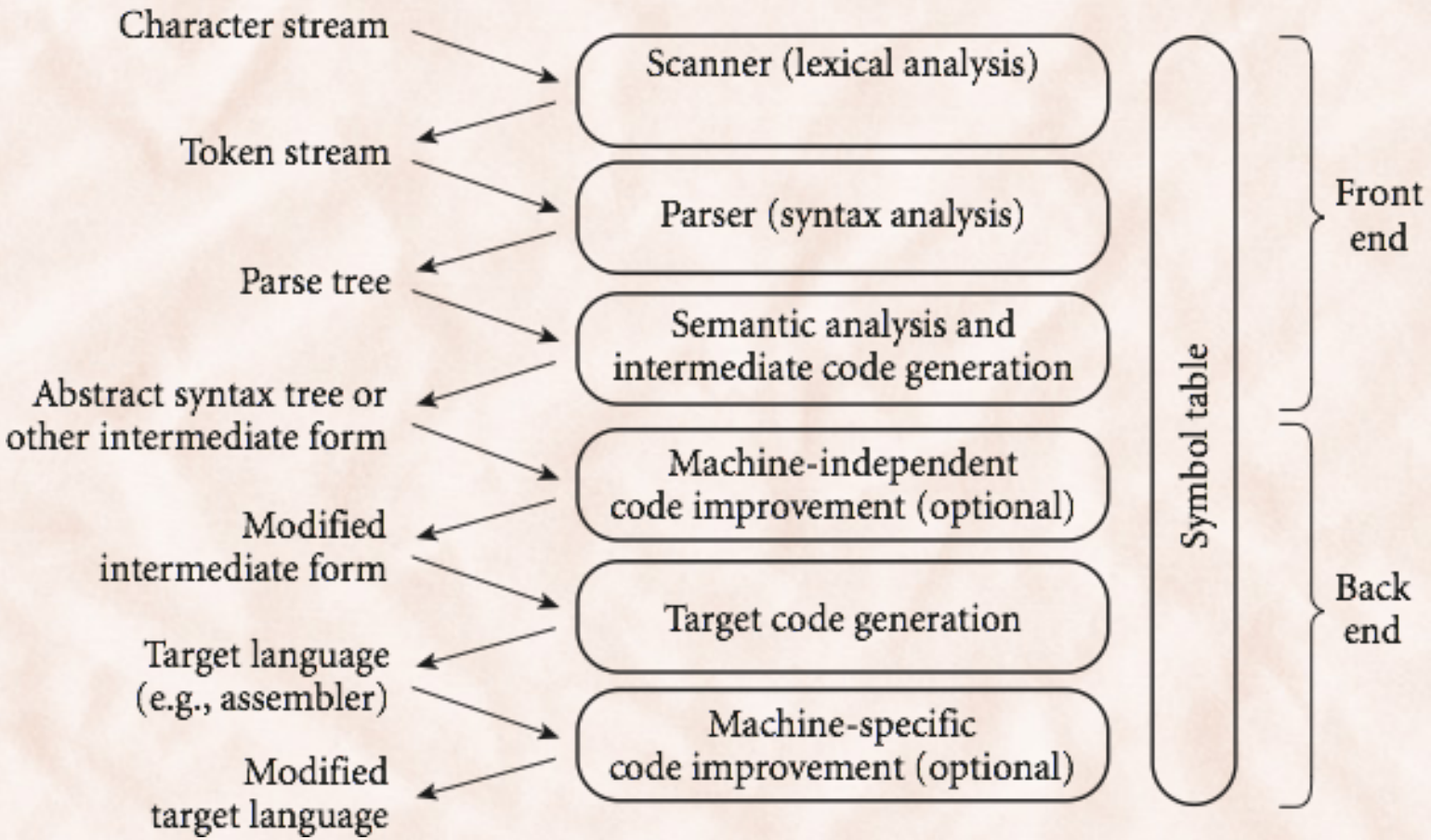


Lexical Analysis

- Groups characters into *tokens*, the smallest meaningful units of the program.

```
int      main   (   )        {
int      i      =   getint   (   )   ,   j   =   getint   (   )   ;
while    (      i   !=       j   )   {
if       (      i   >        j   )   i   =   i   -       j   ;
else     j      =   j        -   i   ;
}
putint   (      i   )        ;
}
```

Phases of Compilation



Syntactic Analysis

- Organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents:
  - Potentially recursive rules known as *context-free grammar* define the ways in which these constituents combine.

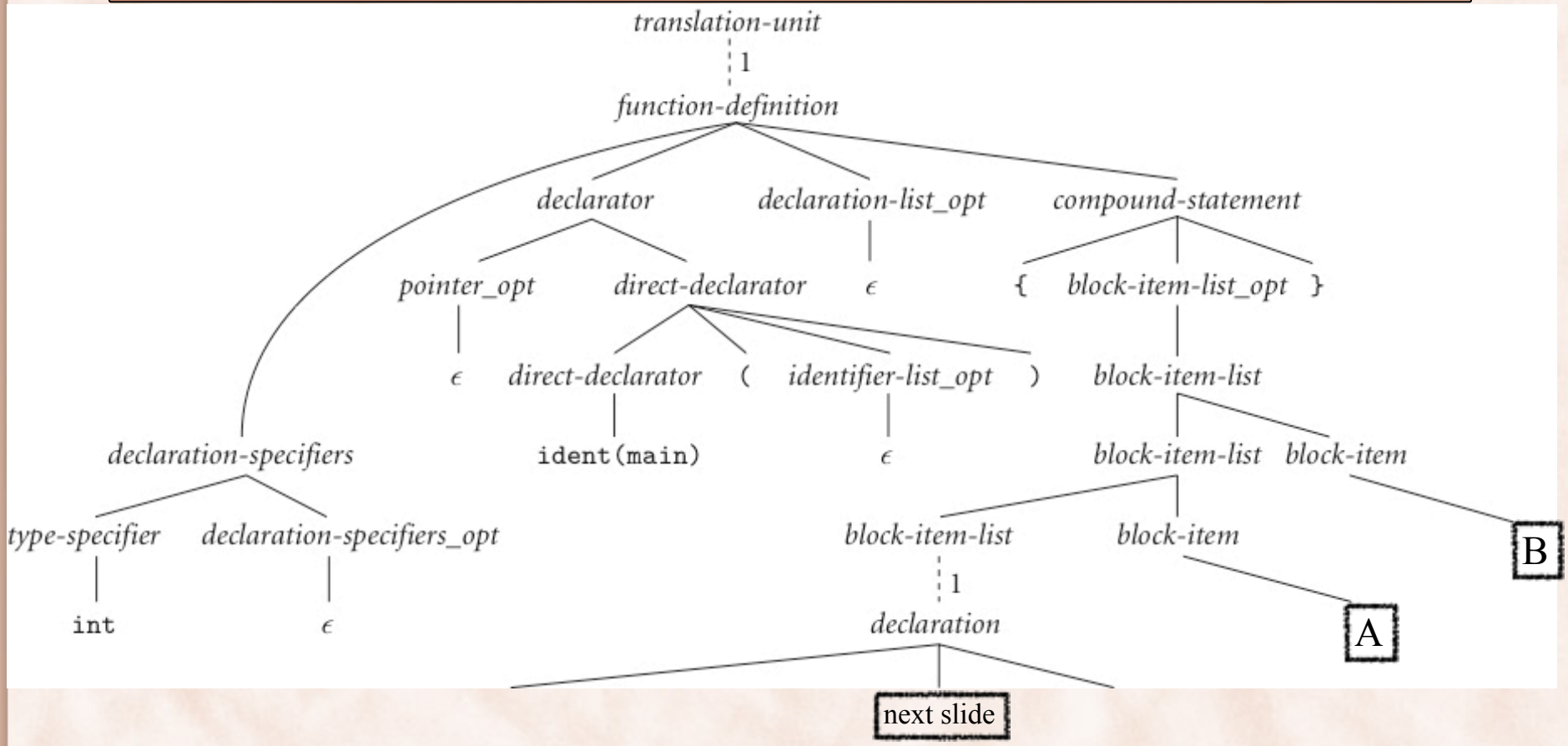
iteration-statement → **while (** *expression* **)** *statement*

statement → *compound-statement*
compound-statement → **{** *block-item-list opt* **}**

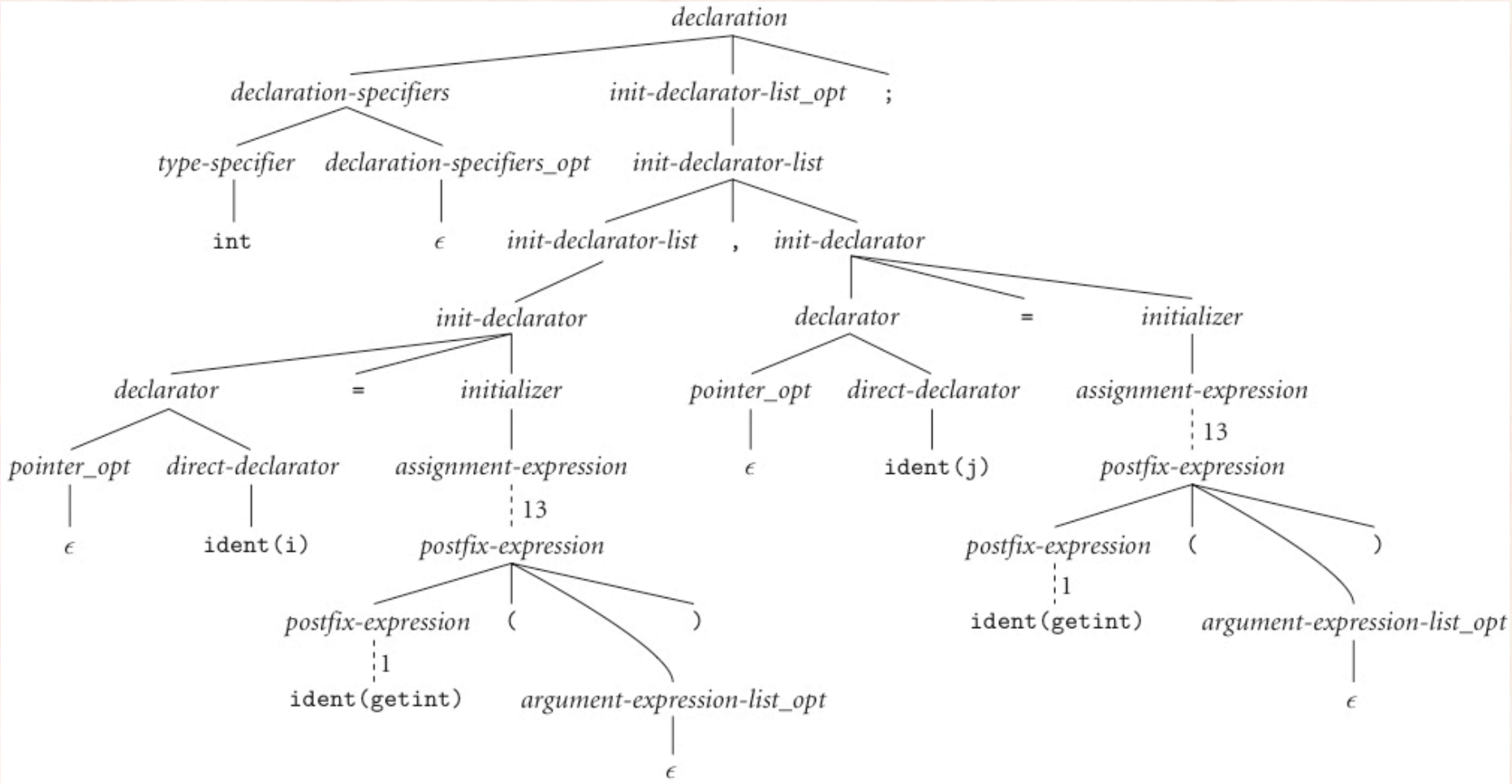
block-item-list opt → *block-item-list*
block-item-list opt → ϵ

block-item-list → *block-item*
block-item-list → *block-item-list block-item*
block-item → *declaration*
block-item → *statement*

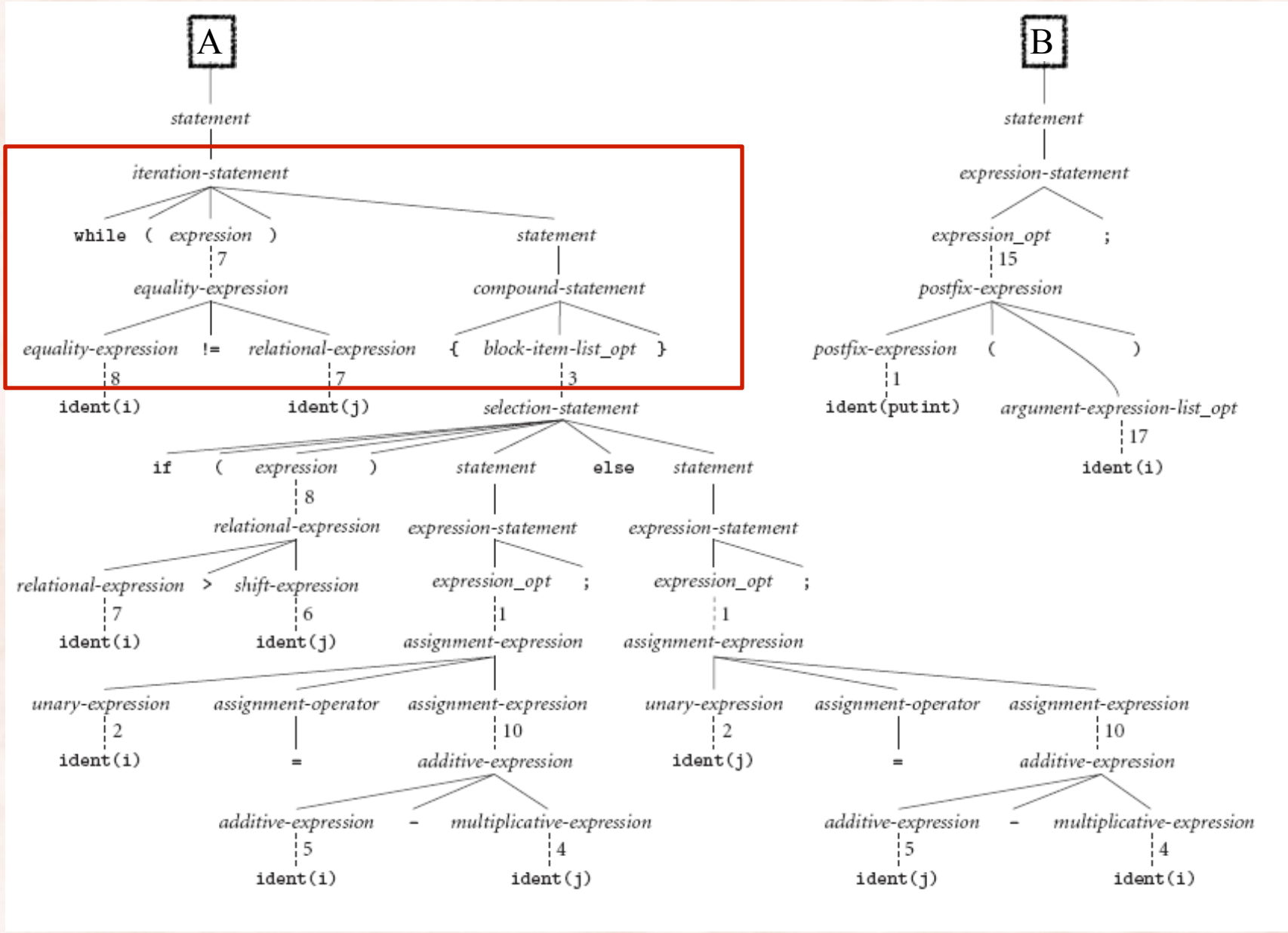
Parse Tree (Concrete Syntax Tree)



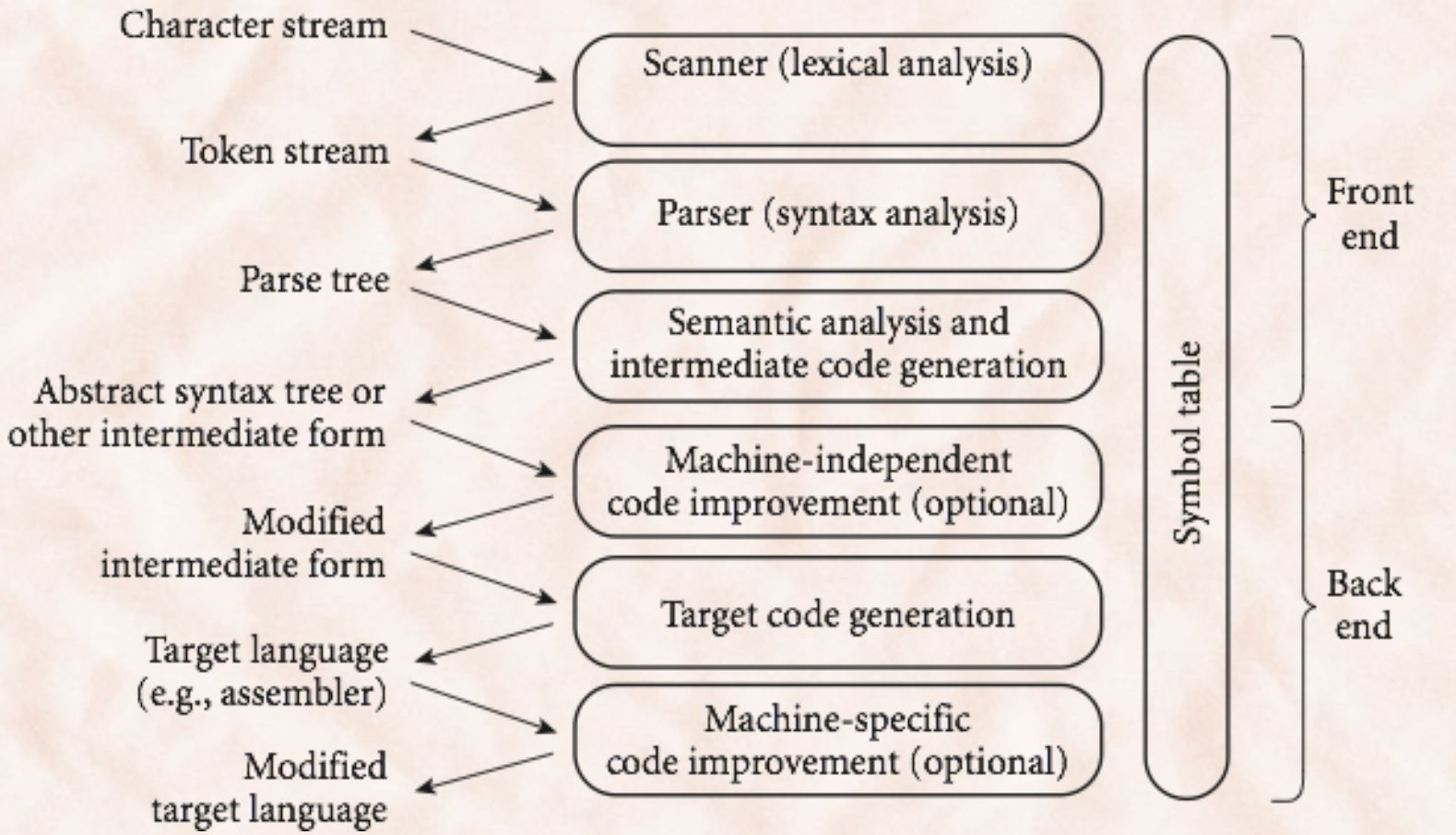
Parse Tree (continued)



Parse Tree (continued)



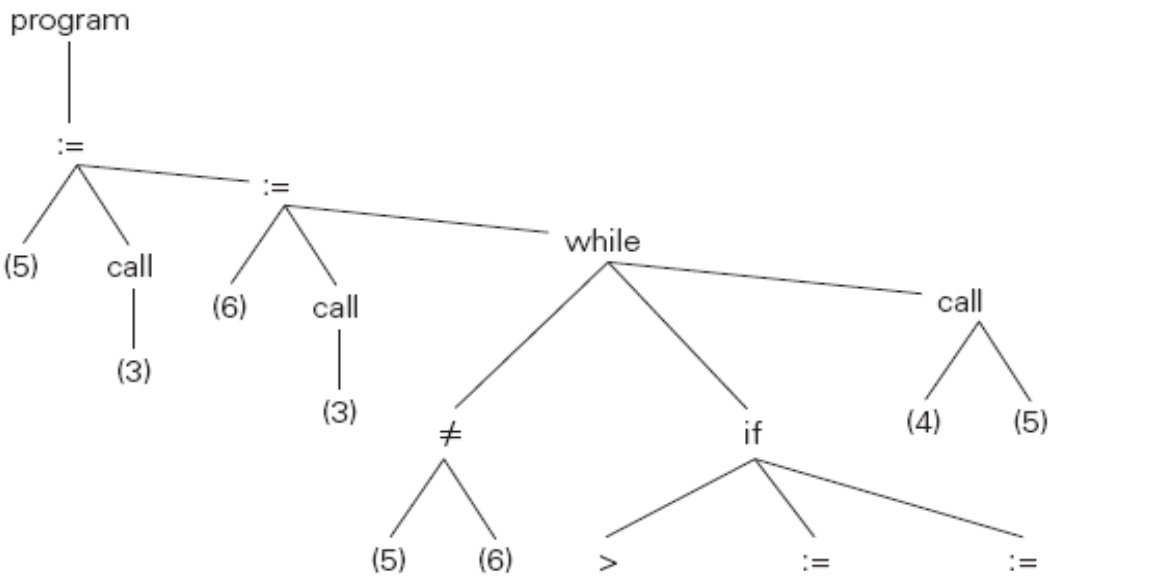
Phases of Compilation



Semantic Analysis

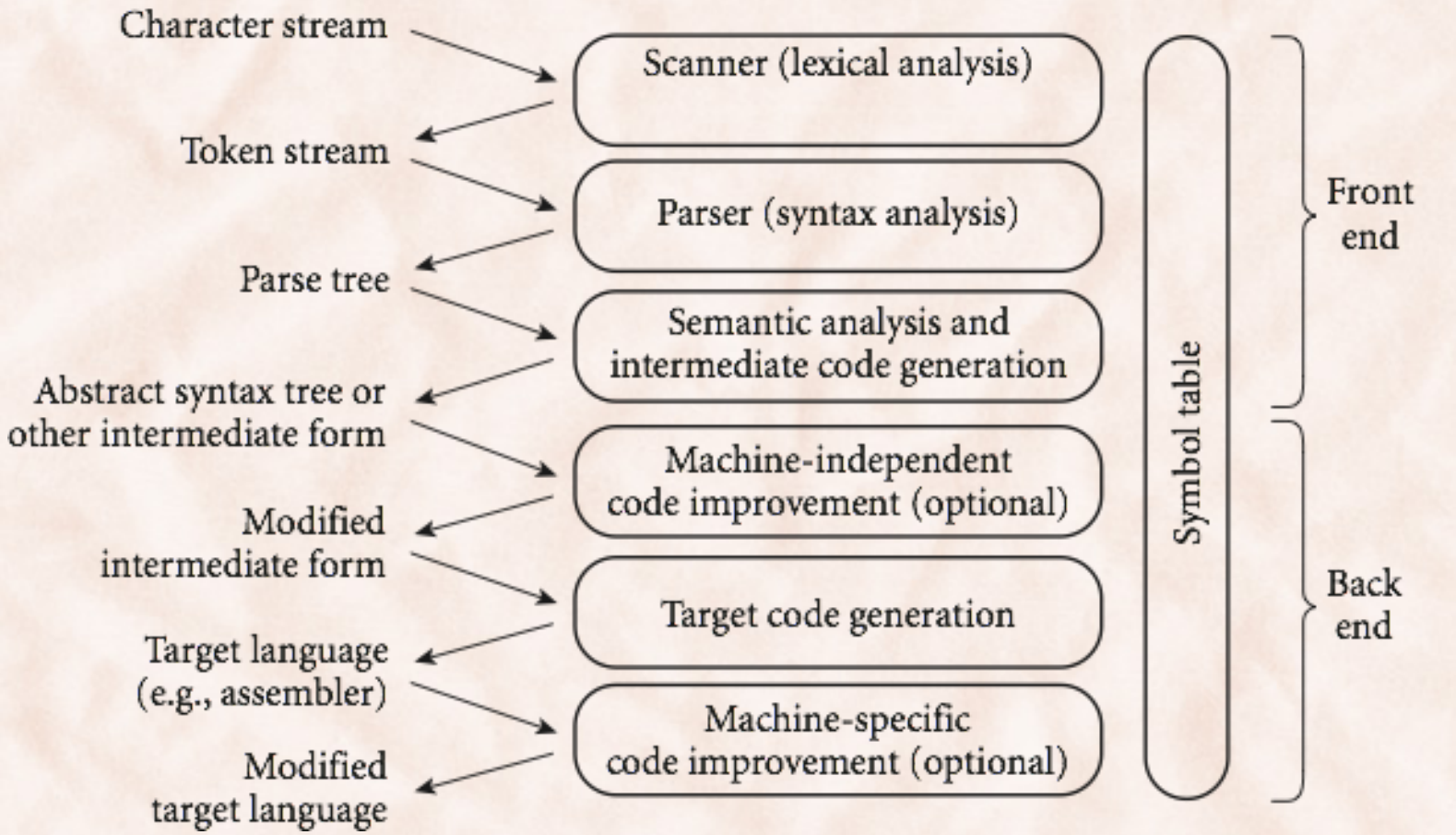
- Enforces rules that pertain to the *meaning* of the program:
  - **Static Semantics**:
    - identifiers declared before used.
    - identifier used in appropriate context.
    - subroutine calls provide correct number and type of arguments.
    - labels in different switch clauses are distinct constants.
  - **Dynamic Semantics**:
    - variables should be given a value before being used.
    - pointers are dereferenced only if they point to valid objects.
    - array subscript expressions are within the bounds of the array.
    - arithmetic operations do not overflow.

Syntax Tree (Abstract Syntax Tree)



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Phases of Compilation



Target Code Generation & Improvement

- Using the annotated syntax tree as intermediate form:
  - traverse the symbol table to assign locations to variables.
  - traverse the syntax tree, generating:
    - loads and stores for variable references.
    - arithmetic operations, tests, and branches.
  - see Figure 1.6 in PLP.

- Code Improvement:
  - keep local variables in registers, instead of locations on the stack.
  - see Example 1.2 in PLP.

Outline

- Reasons for Studying Programming Languages
- Influences on Language Design
- Language Paradigms
- Implementation Methods
- Overview of Compilation

Language Paradigms

- Imperative (Turing Machines – Alan Turing, 1912-1954)
  - Designed around the von Neumann architecture.
  - Computation is performed through statements that change a program's state.
  - Central features are variables, assignment statements, and iteration; sequencing of commands, explicit state update via assignment.
  - May include :
    - OO programming languages,
    - scripting languages,
    - visual languages.
  - Examples: Fortran, Algol, Pascal, C/C++, Java, Perl, JavaScript, Visual BASIC .NET

Language Paradigms

- Functional (Lambda Calculus – Alonzo Church, 1903-1995)
  - Main means of making computations is by applying functions to given parameters.
  - Examples: LISP, Scheme, ML, Haskell
  - May include OO concepts.

- Logic (Predicate Calculus – Gotlob Frege, 1848-1925)
  - Rule-based (rules are specified in no particular order).
  - Computations are made through a logical inference process.
  - Example: Prolog, CLIPS.
  - May include OO concepts.

Summary

- The study of programming languages is valuable for a number of reasons:
  - Increase our capacity to use different constructs.
  - Enable us to choose languages more intelligently.
  - Makes learning new languages easier.
- Major influences on language design:
  - *machine architecture.*
  - *software development methodologies*.
  - *application domains*.
- Major implementation methods:
  - *compilation, pure interpretation, hybrid,* and *just-in-time.*

Criteria for Language Evaluation

- **Readability**: the ease with which programs can be read and understood.
- **Writability**: the ease with which a language can be used to create programs.
- **Reliability**: conformance to specifications (e.g. program correctness).
- **Cost**: the ultimate total cost associated with a PL.

Readibility

- Overall simplicity
  - A manageable set of basic features and constructs.
  - Minimal feature multiplicity.
    - Example: increment operators in C
  - Minimal operator overloading.
    - Example: C++ vs. Java
- Orthogonality
  - A relatively small set of primitive constructs that can be combined in a relatively small number of ways.
    - Example: addition in assembly on IBM mainframe vs. VAX minicomputers.
  - Every possible combination is legal
    - Example: returning arrays & records in C.

Readibility

- Control statements
  - The presence of well-known control structures
    - Example: *goto* in Fortran & Basic vs. *for/while* loops in C.

"Goto" Version:

```
i = 0;
loop1:
  if (i >= 10)
    goto out1;
loop2:
  if (j >= 10)
    goto out2;
  A[i,j] = 1;
  j++;
  goto loop2;
out2:
  i++;
  j = 0;
  goto loop1;
out1:
```

"For" Version:

?

Readibility

- Data types and structures
  - Adequate predefined data types and structures
    - Example: Boolean vs. Integer
  - The presence of adequate facilities for defining data structures
    - Example: array of structs vs. collection of arrays (C).

- Syntactic design:
  - Identifier forms (allow long names).
  - Special words and methods of forming compound statements.
  - Form and meaning:
    - self-descriptive constructs, meaningful keywords.
      - static keyword in C has context dependent meaning.

Criteria for Language Evaluation

- **Readability**: the ease with which programs can be read and understood.
- **Writability**: the ease with which a language can be used to create programs.
- **Reliability**: conformance to specifications (e.g. program correctness).
- **Cost**: the ultimate total cost associated with a PL.

Writability

- Simplicity and orthogonality
  - Few constructs, a small number of primitives, a consistent, small set of rules for combining them (avoid misuse or disuse of features).

- Support for abstraction
  - The ability to define and use complex structures or operations in ways that allow details to be ignored
  - **Process Abstraction** (e.g. sorting algorithm implemented as a subprogram)
  - **Data Abstraction** (e.g. trees & lists in C++/Java vs. Fortran77).

Writability

- Expressivity
  - A set of relatively convenient ways of specifying operations.
  - Strength and number of operators and predefined functions.
  - Examples:
    - Increment operators in C.
    - Short circuit operators in Ada.
    - Counting loops with *for* vs. *while* in Java.

Criteria for Language Evaluation

- **Readability**: the ease with which programs can be read and understood.
- **Writability**: the ease with which a language can be used to create programs.
- **Reliability**: conformance to specifications (e.g. program correctness).
- **Cost**: the ultimate total cost associated with a PL.

Reliability

- Type checking
  - Testing for type errors at compile-time vs. run-time.
  - Examples:
    - (+) Ada, Java, C#, C++.
    - (−) C.

- Exception handling
  - Intercept run-time errors, take corrective measures and continue.
  - Examples:
    - (+) Ada, C++, Java.
    - (−) Fortran, C.

Reliability

- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location:
    - Pointers in C, references in C++.
- Readability and writability
  - A language that does not support “natural” ways of expressing an algorithm will require the use of “unnatural” approaches (less safe), and hence reduced reliability.

Criteria for Language Evaluation

- **Readability**: the ease with which programs can be read and understood.
- **Writability**: the ease with which a language can be used to create programs.
- **Reliability**: conformance to specifications (e.g. program correctness).
- **Cost**: the ultimate total cost associated with a PL.

Cost

- Training programmers to use the language.
- Writing programs (closeness to particular applications).
- Compiling programs:
  - Tradeoff between compile vs. execution time through optimization.
- Executing programs:
  - Example: many run-time type checks slow the execution (Java).
- Language implementation system:
  - Availability of free compilers.
- Reliability: poor reliability leads to high costs.
- Maintaining programs:
  - Corrections & adding new functionality.

Other Criteria

- Portability
  - The ease with which programs can be moved from one implementation to another (helped by standardization).
- Generality
  - The applicability to a wide range of applications.
- Well-definedness
  - The completeness and precision of the language's official definition.

Trade-offs in Language Design

- Reliability vs. cost of execution
  - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs
- Readability vs. writability

  Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability.
- Writability (flexibility) vs. reliability
  - Example: C++ pointers are powerful and very flexible but are unreliable.