Languages and Compilers
(SProg og Oversættere)

Lecture 1
Overview of the course and Language processors

Bent Thomsen

Department of Computer Science

Aalborg University

What is the Most Important Open Problem in Computing?

Increasing Programmer Productivity

- Write programs quickly
- Write programs easily
- Write programs correctly

- Why?
  - Decreases development cost
  - Decreases time to market
  - Decreases support cost

How to increase Programmer Productivity?

3 ways of increasing programmer productivity:

1. Process (software engineering)
   - Controlling programmers
   - Good process can yield up to 20% increase
2. Tools (verification, static analysis, program generation)
   - Good tools can yield up to 10% increase
3. Better designed Languages --- the center of the universe!
   - Core abstractions, mechanisms, services, guarantees
   - Affect how programmers approach a task (C *vs.* Haskell)
   - New languages can yield 700% increase

Quicksort in C and Haskell

```
// To sort array a[] of size n: qsort(a,0,n-1)

void qsort(int a[], int lo, int hi) {
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];

    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    a[hi] = a[l];
    a[l] = p;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

```
qsort [] = []
qsort (x:xs) =
    qsort (filter (< x) xs)
    ++ [x] ++
    qsort (filter (>= x) xs)
```

Programming Languages and Compilers are at the core of Computing

All software is written in a programming language

Learning about compilers will teach you a lot about the programming languages you already know.

Compilers are big – therefore you need to apply all you knowledge of software engineering.

The compiler is the program from which all other programs arise.
Get it wrong and a lot of people will be affected!

What is a Programming Language?

- A set of rules that provides a way of telling a computer what operations to perform.
- A set of rules for communicating an algorithm
- A linguistic framework for describing computations
- Symbols, words, rules of grammar, rules of semantics
  - Syntax and Semantics
  - (Libraries, Frameworks, Patterns and Pragmas)

Why Are There So Many Programming Languages

- Why do some people speak French?
- Programming languages have evolved over time as better ways have been developed to design them.
  - First programming languages were developed in the 1950s
  - Since then thousands of languages have been developed
- Different programming languages are designed for different types of programs.

Levels of Programming Languages

High-level program

```
class Triangle {
  ...
  float surface()
    return b*h/2;
  }
```

Low-level program

```
LOAD r1,b
LOAD r2,h
MUL r1,r2
DIV r1,#2
RET
```

Executable Machine code

```
0001001001000101
0010010011101100
10101101001...
```

Types of Programming Languages

- First Generation Languages

  Machine
  ```
  0000 0001 0110 1110
  0100 0000 0001 0010
  ```

- Second Generation Languages

  Assembly
  ```
  LOAD x
  ADD R1 R2
  ```

- Third Generation Languages

  High-level imperative/object oriented
  ```
  public Token scan ( ) {
  while (currentchar == ' '
  || currentchar == '\n')
  {….} }
  ```

  Fortran, Pascal, Ada, C, C++, Java, C#

- Fourth Generation Languages

  Database
  ```
  select fname, lname
  from employee
  where department='Sales'
  ```

  SQL

- Fifth Generation Languages

  Functional
  ```
  fact n = if n==0 then 1
  else n*(fact n-1)
  ```

  Logic
  ```
  uncle(X,Y) :- parent(Z,Y), brother(X,Z).
  ```

  Lisp, SML, Haskel, Prolog

Beyond Fifth Generation Languages

- Some talk about
  - Aspect Oriented Programming (Not so much ☺)
  - Agent Oriented Programming
  - Intentional Programming
  - Natural language programming
- Maybe you will invent the next big language

The principal paradigms

- Imperative Programming
  - Fortran, Pascal, C
- Object-Oriented Programming
  - Simula, SmallTalk, C++, Java, C#
- Logic/Declarative Programming
  - Prolog, SQL
- Functional/Applicative Programming
  - Lisp, Scheme, Haskell, SML, F#
- (Aspect Oriented Programming)
  - AspectJ, AspectC#, Aspect.Net
- (Reactive Programming)
  - RxJava, Angular, React, Vue, Functional reactive

The Multi-Paradigm Era

Microsoft fellow Anders Hejlsberg, who heads development on C#, said:

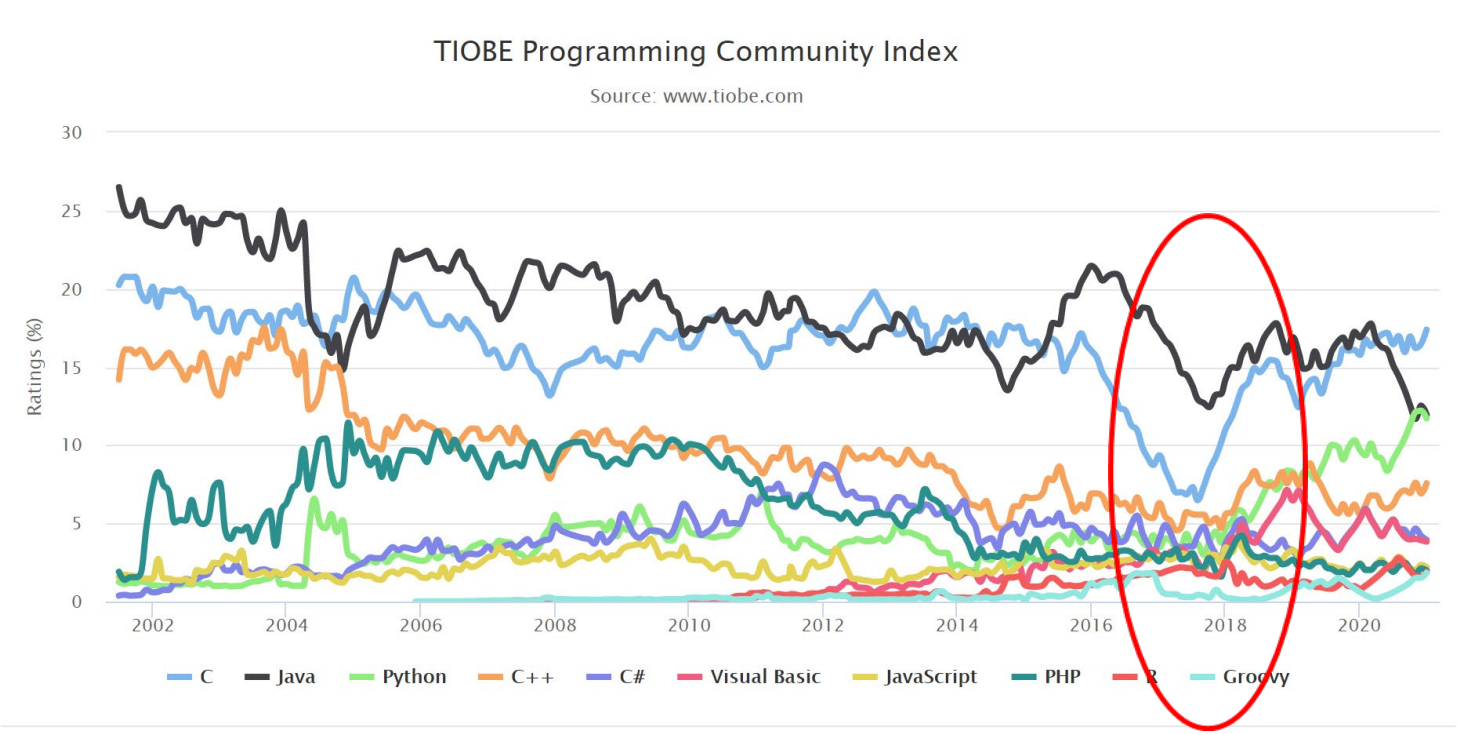
"The taxonomies of programming languages are starting to break down,"

He points to dynamic languages, programming languages, and functional languages.

He said "future languages are going to be an amalgam of all of the above.

If in doubt, take a look at C#

The 10 most popular programming languages



Swift
DART
Erlang
Scala
Lisp
Kotlin
F#
Haskel

https://www.tiobe.com/tiobe-index/

What determines a “good” language

- Formerly: Run-time performance
  - (Computers were more expensive than programmers)
- Now: Life cycle (human) cost is more important
  - Ease of designing, coding
  - Debugging
  - Maintenance
  - Reusability
- FADS
  - A fad is any form of behavior that develops among a large population and is collectively followed enthusiastically for a period of time, generally as a result of the behavior being perceived as popular by one's peers or being deemed "cool" Source Wikipedia

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Evidense Based Programming Language Design

- New direction in PL Resreach (ca. 2005)
  - Use social science techniques
    - Data Mining of repositories or MOC (massive Online Course)
    - Questionaeres
      - E.g. Perl vs. Python (age difference)
      - E.g. ObjectiveC (Most like used in small companies)
  - Use medical science techniques
    - Controlled experiments
      - E.g. Static vs. Dymanic types
    - Placebo effects
      - E.g. Quorum vs. Perl. Vs Randomo
  - Use HCI techniques
    - Eye tracking and Brain Scans
    - (Usability Lab)
    - Discount Method for Programming Language Evaluation
- Actually not that new
  - SmallTalk and Logo designers used observational studies in the 70ies

Programming languages are languages

- But Computer languages lack ambiguity and vagueness
- In English sentences can be ambiguous
  - *I saw the man with a telescope*
    - Who had the telescope?
  - *Take a pinch of salt*
    - How much is a pinch?
- In a programming language a sentence either means one thing or it means nothing

Programming Language Specification

- Why?
  - A communication device between people who need to have a common understanding of the PL:
    - language designer, language implementor, language user
- What to specify?
  - Specify what is a ‘well formed’ program
    - syntax
    - contextual constraints (also called static semantics):
      - scope rules
      - type rules
  - Specify what is the meaning of (well formed) programs
    - semantics (also called runtime semantics)

Programming Language Specification

- Why?
- What to specify?
- How to specify ?
  - Formal specification: use some kind of precisely defined formalism
  - Informal specification: description in English.

  - Usually a mix of both (e.g. Java specification)
    - Syntax => formal specification using CFG
    - Contextual constraints and semantics => informal
    - Formal semantics has been retrofitted though
  - But trend towards more formality (C#, Fortress)
    - fortress.pdf
    - Ecma-334.pdf

Specification of Method invocation in C# according to the ECMA 334 standard

14.5.5 Invocation expressions

An *invocation-expression* is used to invoke a method.

invocation-expression:
primary-expression (*argument-list*$_{opt}$)

The *primary-expression* of an *invocation-expression* shall be a method group or a value of a *delegate-type*. If the *primary-expression* is a method group, the *invocation-expression* is a method invocation (§14.5.5.1). If the *primary-expression* is a value of a *delegate-type*, the *invocation-expression* is a delegate invocation (§14.5.5.2). If the *primary-expression* is neither a method group nor a value of a *delegate-type*, a compile-time error occurs.

The optional *argument-list* (§14.4.1) provides values or variable references for the parameters of the method.

The result of evaluating an *invocation-expression* is classified as follows:

- If the *invocation-expression* invokes a method or delegate that returns `void`, the result is nothing. An expression that is classified as nothing cannot be an operand of any operator, and is permitted only in the context of a *statement-expression* (§15.6).
- Otherwise, the result is a value of the type returned by the method or delegate.

14.5.5.1 Method invocations

For a method invocation, the *primary-expression* of the *invocation-expression* shall be a method group. The method group identifies the one method to invoke or the set of overloaded methods from which to choose a specific method to invoke. In the latter case, determination of the specific method to invoke is based on the context provided by the types of the arguments in the *argument-list*.

The compile-time processing of a method invocation of the form `M(A)`, where `M` is a method group (possibly including a *type-argument-list*), and `A` is an optional *argument-list*, consists of the following steps:

- The set of candidate methods for the method invocation is constructed. For each method `F` associated with the method group `M`:
  - If `F` is non-generic, `F` is a candidate when:
    - `M` has no type argument list, and
    - `F` is applicable with respect to `A` (§14.4.2.1).
  - If `F` is generic and `M` has no type argument list, `F` is a candidate when:
    - Type inference (§25.6.4) succeeds, inferring a list of type arguments for the call, and
    - Once the inferred type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of `F` satisfy their constraints (§25.7.1), and the parameter list of `F` is applicable with respect to `A` (§14.4.2.1), and
  - If `F` is generic and `M` includes a type argument list, `F` is a candidate when:
    - `F` has the same number of method type parameters as were supplied in the type argument list, and
    - Once the type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of `F` satisfy their constraints (§25.7.1), and the parameter list of `F` is applicable with respect to `A` (§14.4.2.1).
- The set of candidate methods is reduced to contain only methods from the most derived types: For each method `C.F` in the set, where `C` is the type in which the method `F` is declared, all methods declared in a base type of `C` are removed from the set. Furthermore, if `C` is a class type other than `object`, all methods declared in an interface type are removed from the set. [*Note*: This latter rule only has affect when the method group was the result of a member lookup on a type parameter having an effective base class other than object and a non-empty effective interface set (§25.7). *end note*]
- If the resulting set of candidate methods is empty, then no applicable methods exist, and a compile-time error occurs.
- The best method of the set of candidate methods is identified using the overload resolution rules of §14.4.2. If a single best method cannot be identified, the method invocation is ambiguous, and a compile-time error occurs. When performing overload resolution, the parameters of a generic method are considered after substituting the type arguments (supplied or inferred) for the corresponding method type parameters.
- Final validation of the chosen best method is performed:
  - The method is validated in the context of the method group: If the best method is a static method, the method group shall have resulted from a *simple-name* or a *member-access* through a type. If the best method is an instance method, the method group shall have resulted from a *simple-name*, a *member-access* through a variable or value, or a *base-access*. If neither of these requirements is true, a compile-time error occurs.
  - If the best method is a generic method, the type arguments (supplied or inferred) are checked against the constraints (§25.7.1) declared on the generic method. If any type argument does not satisfy the corresponding constraint(s) on the type parameter, a compile-time error occurs.

Once a method has been selected and validated at compile-time by the above steps, the actual run-time invocation is processed according to the rules of function member invocation described in §14.4.3.

[*Note*: The intuitive effect of the resolution rules described above is as follows: To locate the particular method invoked by a method invocation, start with the type indicated by the method invocation and proceed up the inheritance chain until at least one applicable, accessible, non-override method declaration is found. Then perform overload resolution on the set of applicable, accessible, non-override methods declared in that type and invoke the method thus selected. *end note*]

Method invocation in Fortress

13.4 Dotted Method Invocations

Syntax:

Primary	::=	*Primary . Id StaticArgs? ParenthesisDelimited*
ParenthesisDelimited	::=	*Parenthesized*
	|	*ArgExpr*
	|	()
Parenthesized	::=	(*Expr*)
ArgExpr	::=	*TupleExpr*
	|	((*Expr* ,)* *Expr* ...)
TupleExpr	::=	((*Expr* ,)+ *Expr*)

A *dotted method invocation* consists of a subexpression (called the receiver expression), followed by '.', followed by an identifier, an optional list of static arguments (described in Chapter 9) and a subexpression (called the *argument expression*). Unlike in function calls (described in Section 13.6), the argument expression must be parenthesized, even if it is not a tuple. There must be no whitespace on the left-hand side of the '.' and the left-hand side of the left parenthesis of the argument expression. The receiver expression evaluates to the receiver of the invocation (bound to the self parameter (discussed in Section 10.2) of the method). A method invocation may include explicit instantiations of static parameters but most method invocations do not include them.

The receiver and arguments of a method invocation are each evaluated in parallel in a separate implicit thread (see Section 5.4). After this thread group completes normally, the body of the method is evaluated with the parameter of the method bound to the value of the argument expression (thus evaluation of the body occurs after evaluation of the receiver and arguments in dynamic program order). The value and the type of a dotted method invocation are the value and the type of the method body.

We say that methods or functions (collectively called as *functionals*) may be *applied to* (also "*invoked on*" or "*called with*") an argument. We use "call", "invocation", and "application" interchangeably.

$$\text{[R-METHOD]}\quad \frac{\texttt{object}\ O\ \_\ (\overrightarrow{x:\_})\ \_\ \texttt{end} \in p \qquad mbody_p(f[\![\overrightarrow{\tau'}]\!], O[\![\overrightarrow{\tau}]\!]) = \{(\overrightarrow{x'}) \to e\}}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).f[\![\overrightarrow{\tau'}]\!](\overrightarrow{v'})] \longrightarrow E[[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x'}]e]}$$

Programming Language Specification

- A Language specification has (at least) three parts
  - Syntax of the language:
    - usually formal in BNF or EBNF + RE for lexems
  - Contextual constraints:
    - scope rules (often written in English, but can be formal)
    - type rules (formal or informal)
  - Semantics:
    - defined by the implementation
    - informal descriptions in English
    - formal using operational or denotational semantics

The Syntax and Semantics course will teach you how to read and write a formal language specification – so pay attention!

Does Syntax matter?

- Syntax is the visible part of a programming language
  - Programming Language designers can waste a lot of time discussing unimportant details of syntax
- The language paradigm is the next most visible part
  - The choice of paradigm, and therefore language, depends on how humans best think about the problem
  - There are no right models of computations – just different models of computations, some more suited for certain classes of problems than others
- The most invisible part is the language semantics
  - Clear semantics usually leads to simple and efficient implementations

- But syntax does matter!
  - Syntax that suggest underlying semantics seems to be important to programmers

Language Processors: What are they?

A programming language processor is any system (software or hardware) that manipulates programs.

Examples:

- Editors
  - Emacs
- Integrated Development Environments
  - Eclipse
  - NetBeans
  - Visual Studio .Net
- Translators (e.g. compiler, assembler, disassembler)
- Interpreters

Interpreters

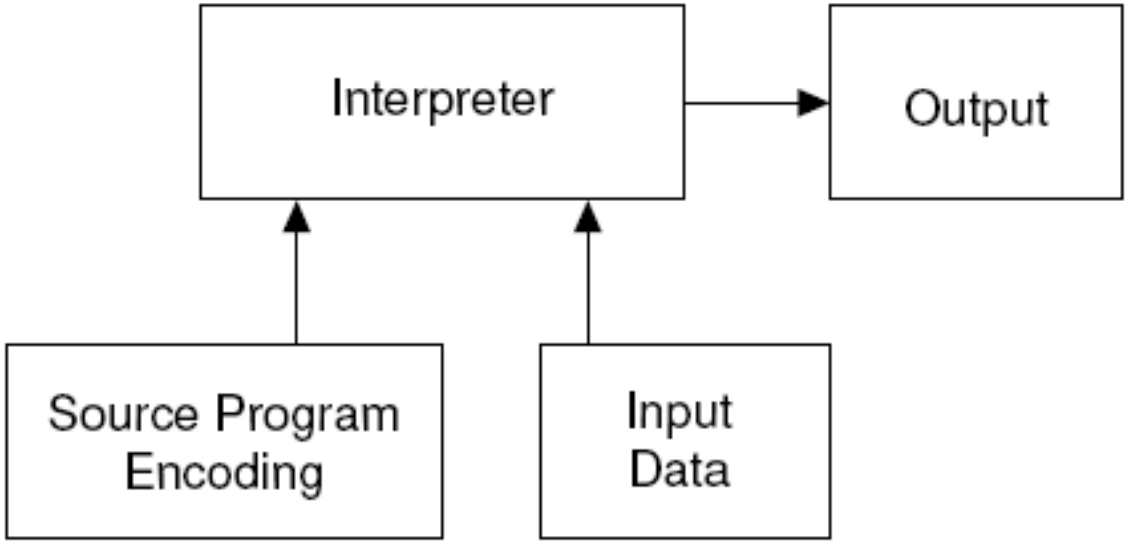
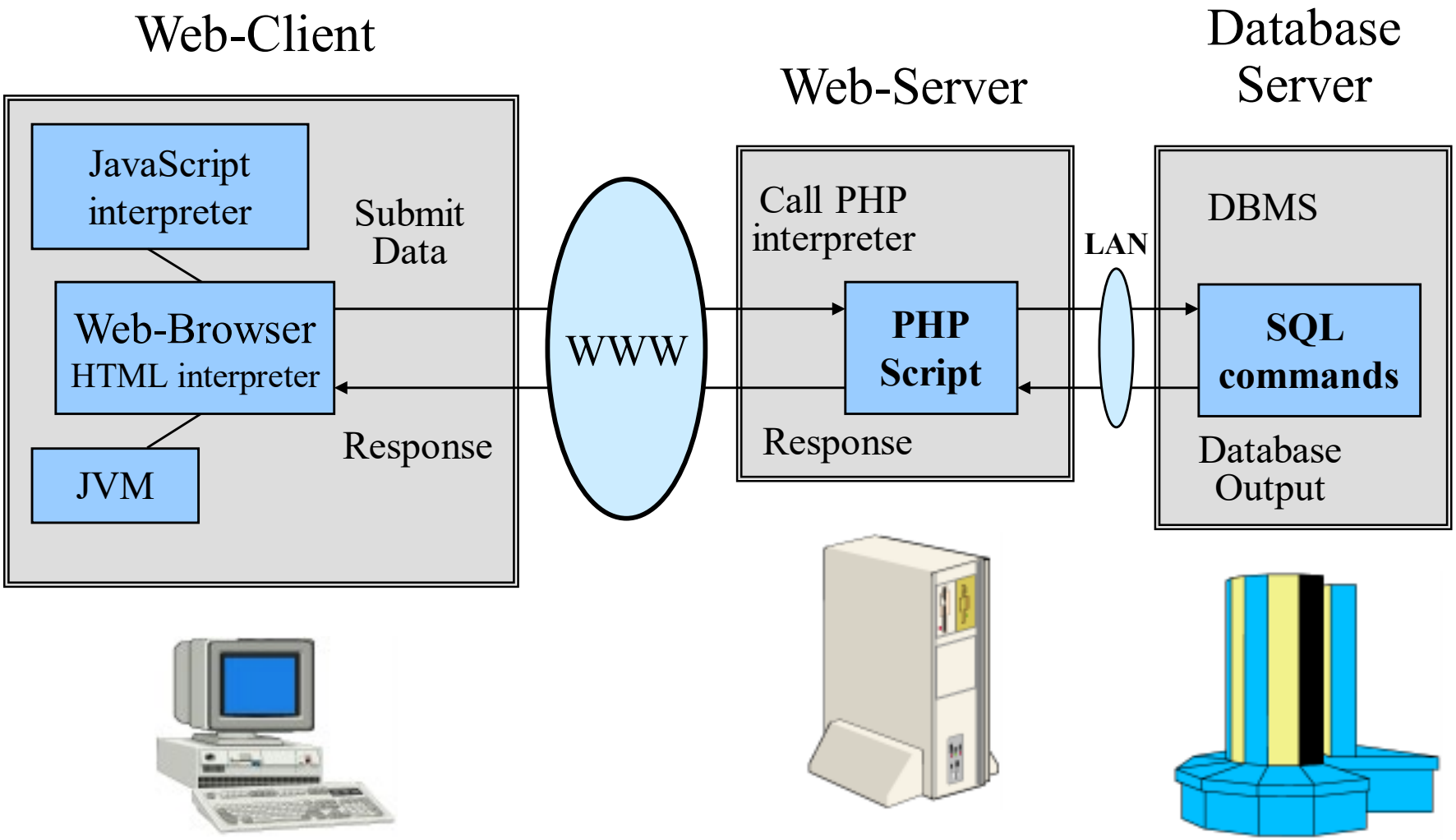


Figure 1.3: An interpreter.

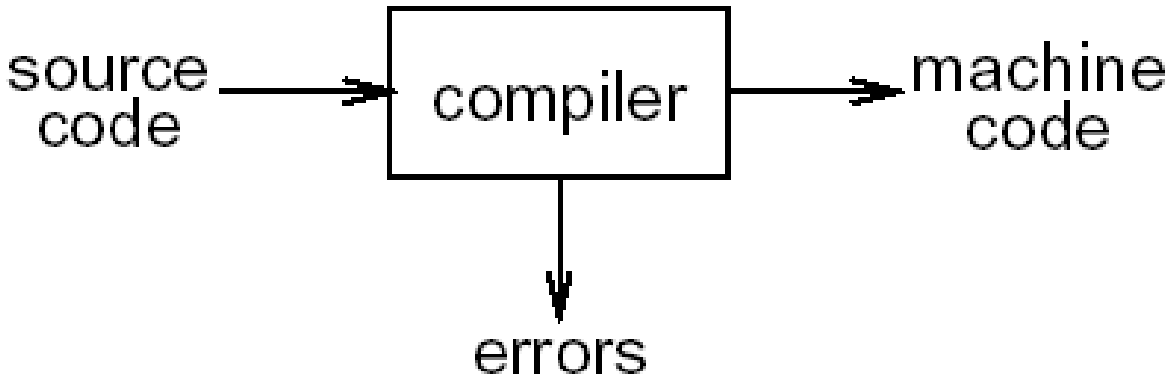
You use lots of interpreters every day!



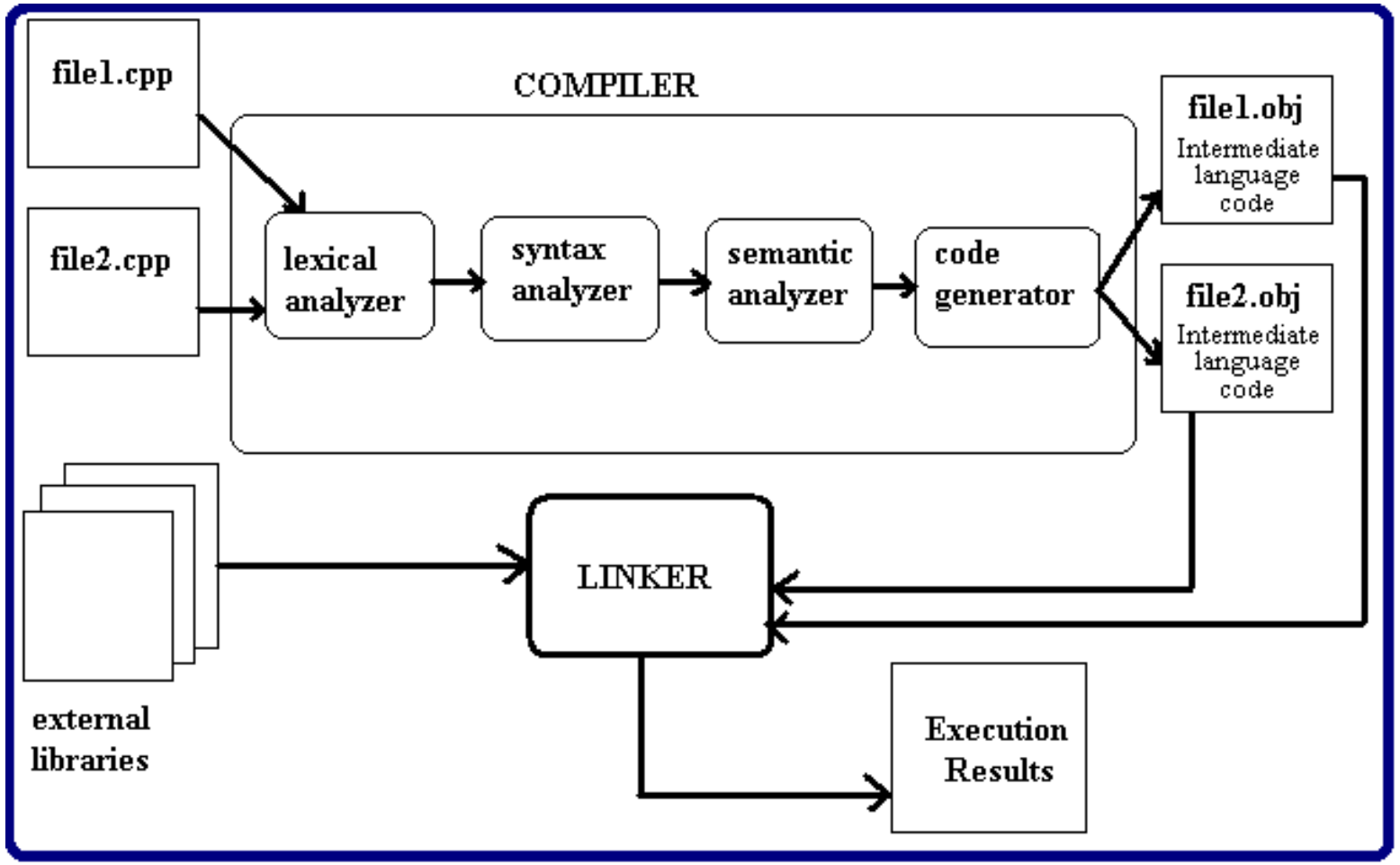
Compilation

- **Compilation** is at least a two-step process, in which the original program (source program) is input to the compiler, and a new program (target program) is output from the compiler. The compilation steps can be visualized as the following.

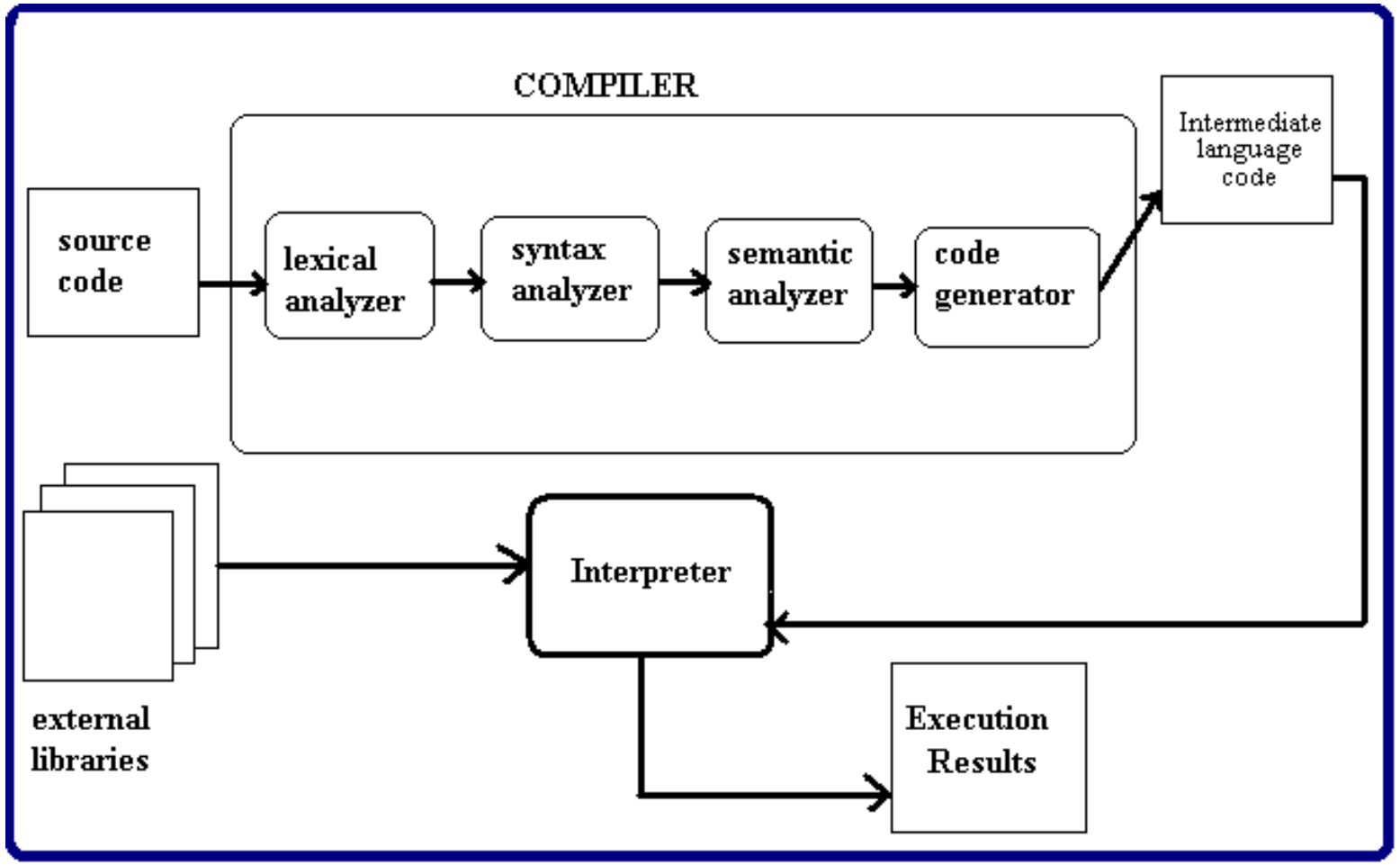
Compiler (simple view)



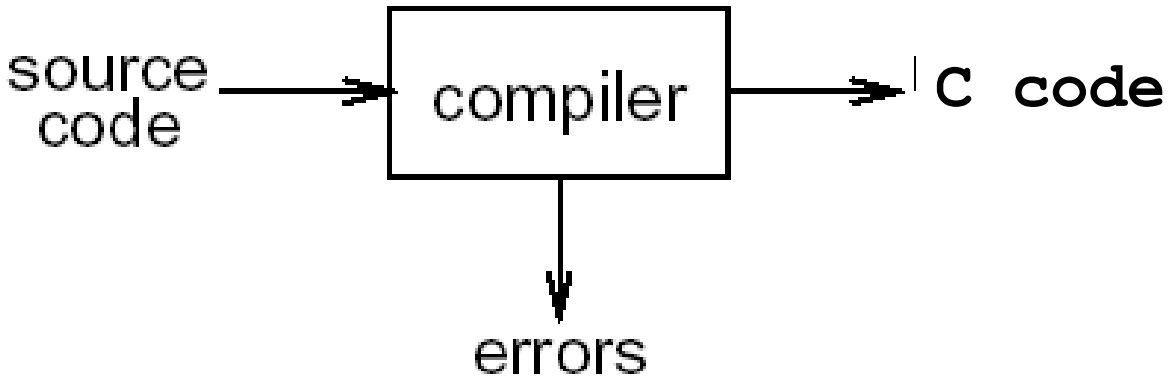
Compiler


file1.cpp
file2.cpp
COMPILER
lexical analyzer
syntax analyzer
semantic analyzer
code generator
file1.obj
Intermediate language code
file2.obj
Intermediate language code
external libraries
LINKER
Execution Results

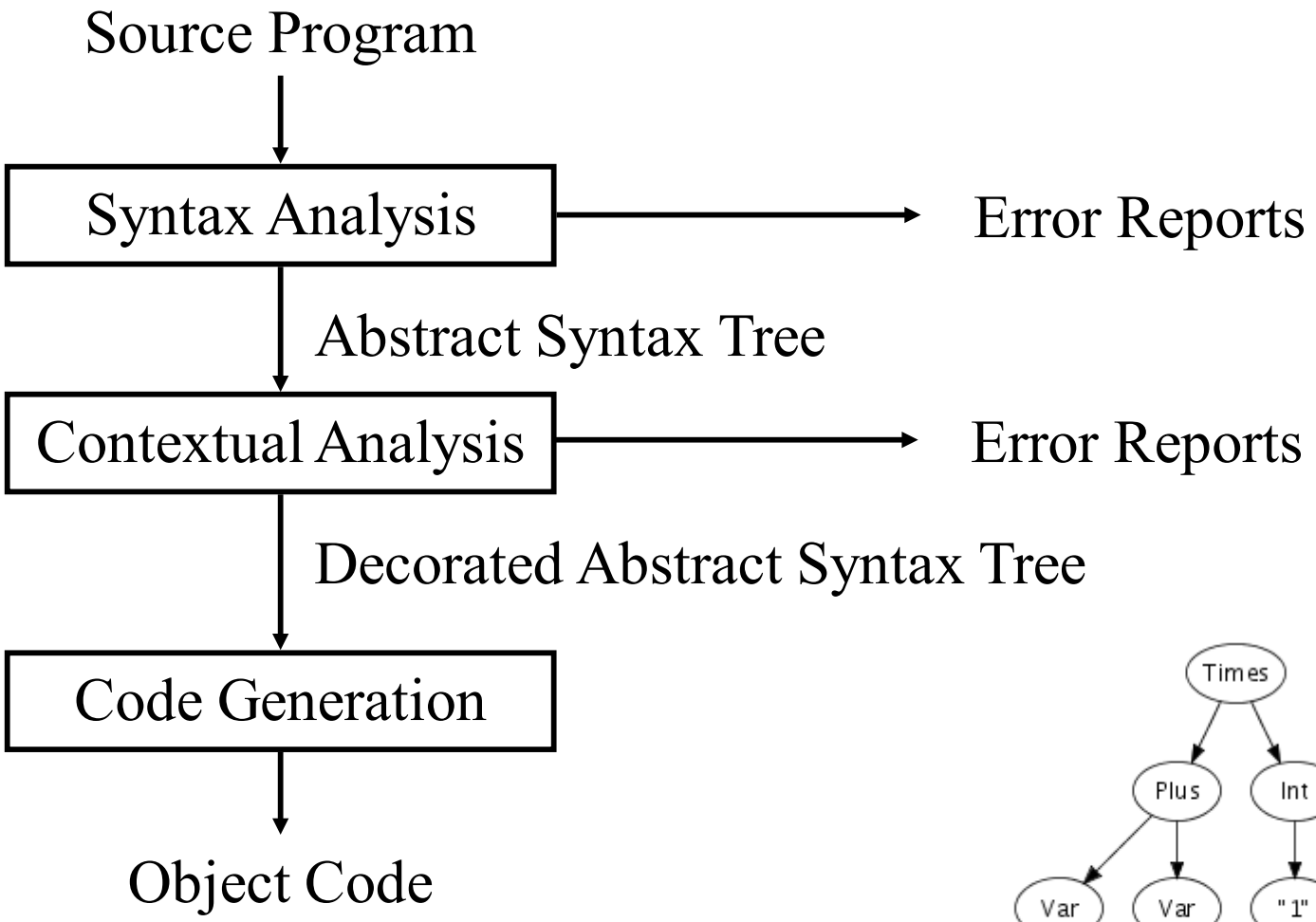
Hybrid compiler / interpreter



Compiler (simple view again)



The Phases of a Compiler



Different Phases of a Compiler

The different phases can be seen as different transformation steps to transform source code into object code.

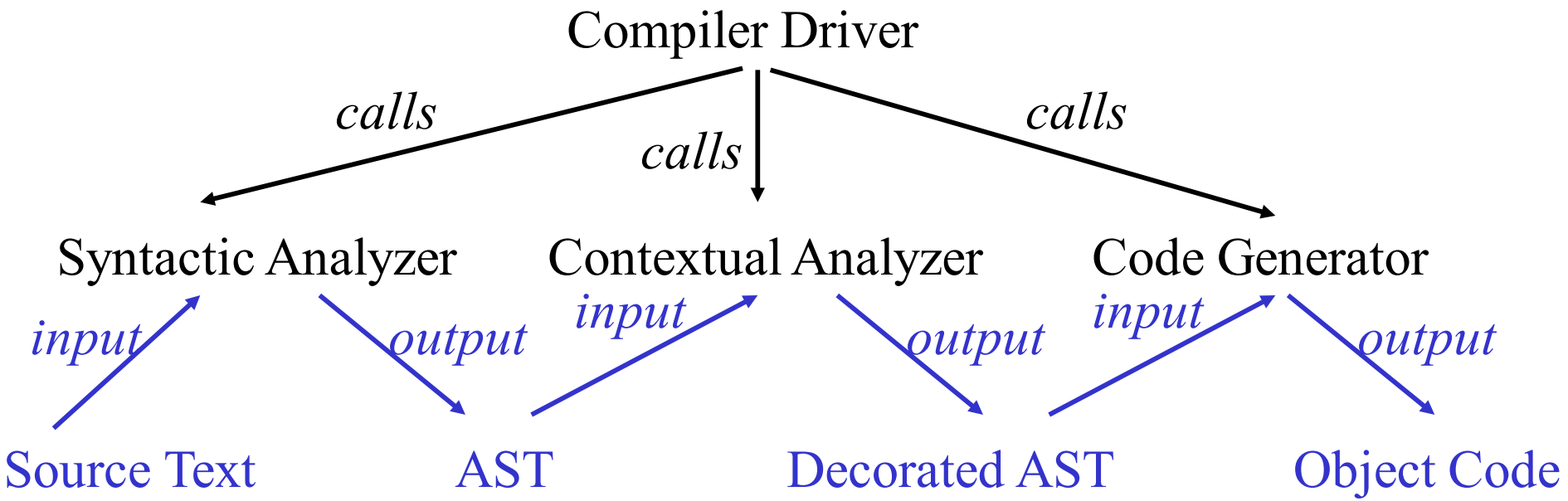
The different phases correspond roughly to the different parts of the language specification:

- Syntax analysis <-> Syntax
- Contextual analysis <-> Contextual constraints
- Code generation <-> Semantics

Multi Pass Compiler

A multi pass compiler makes several passes over the program. The output of a preceding phase is stored in a data structure and used by subsequent phases.

Dependency diagram of a typical Multi Pass Compiler:



Organization of a Compiler

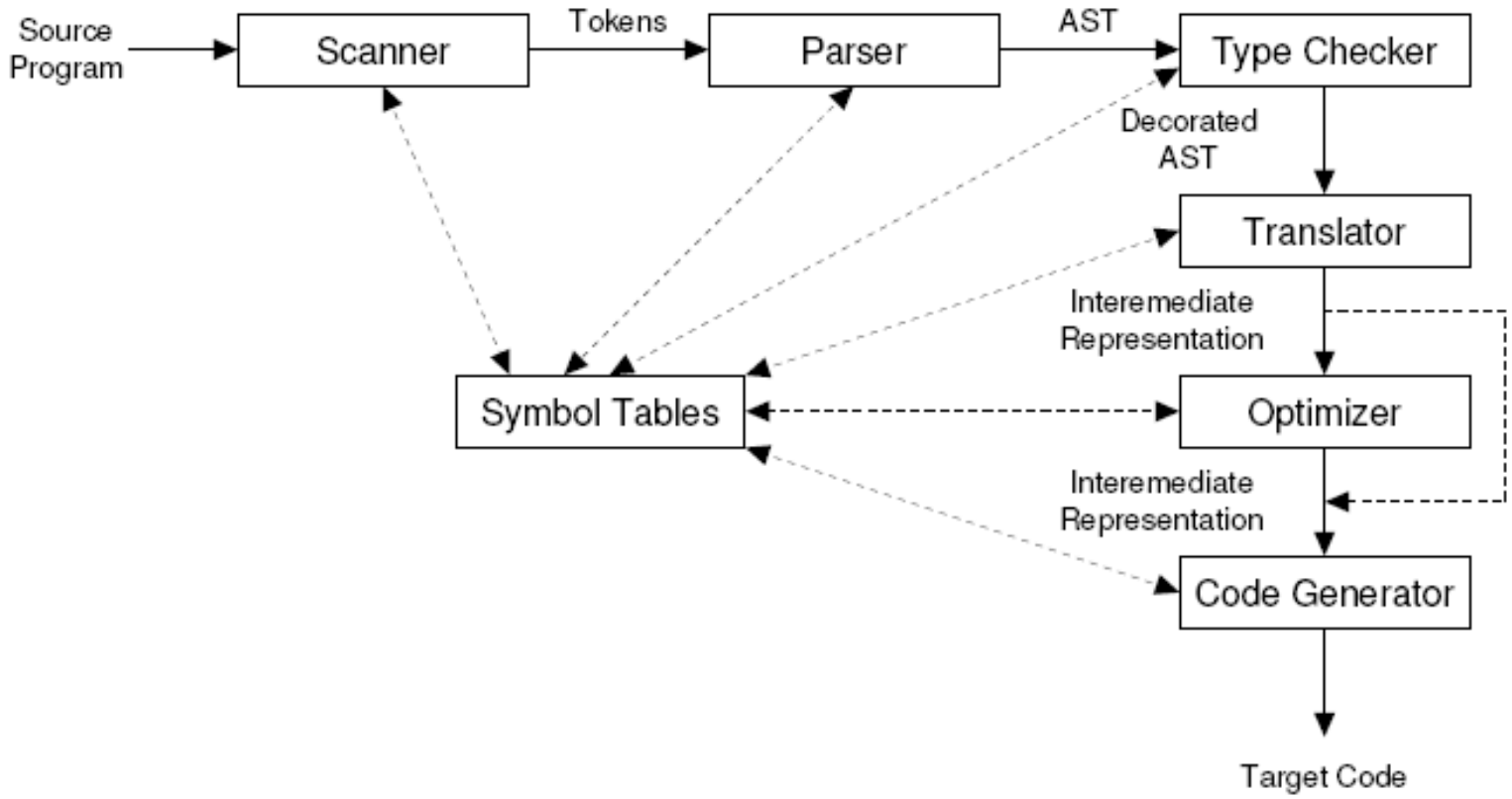
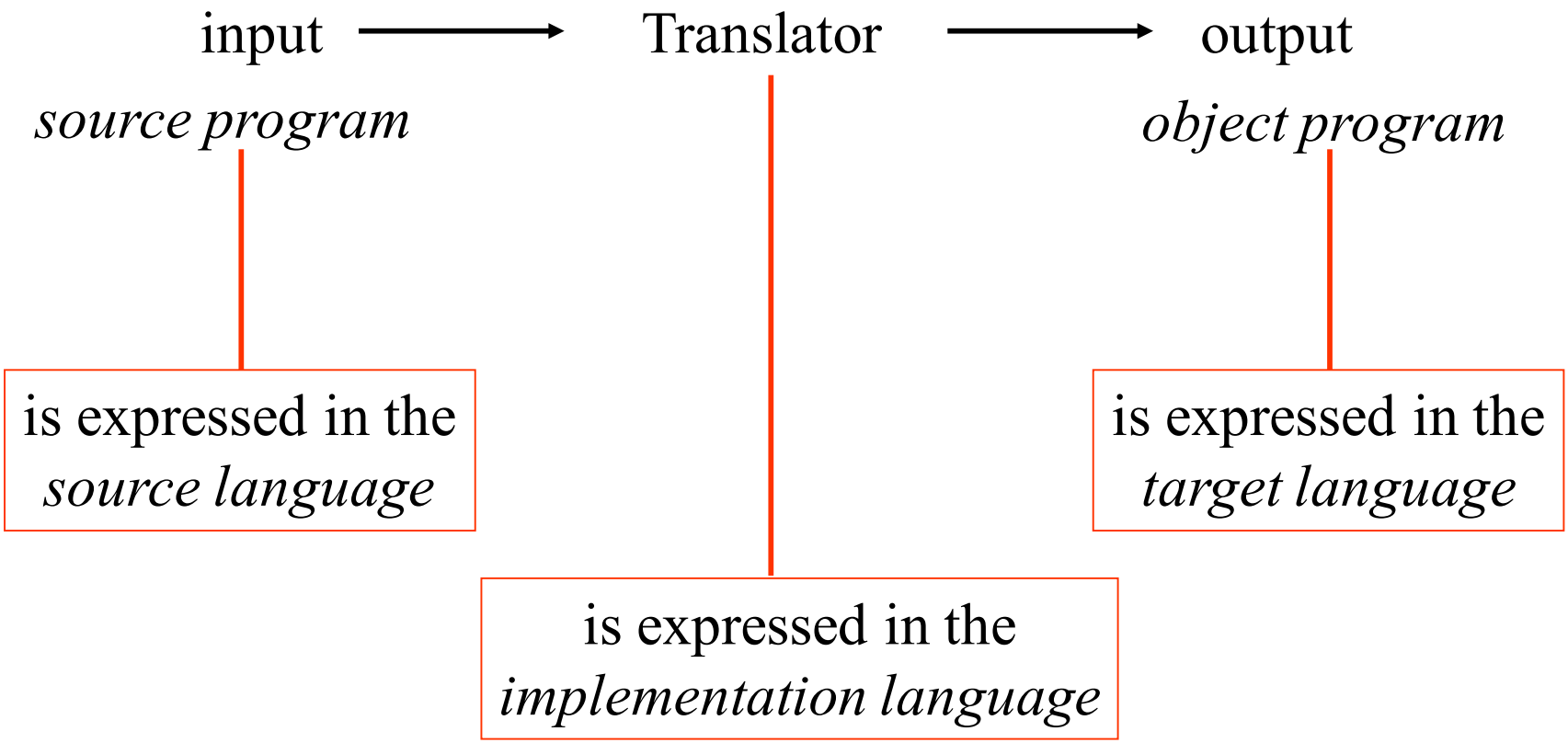


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

Programming Language Implementation

Q: Which programming languages play a role in this picture?

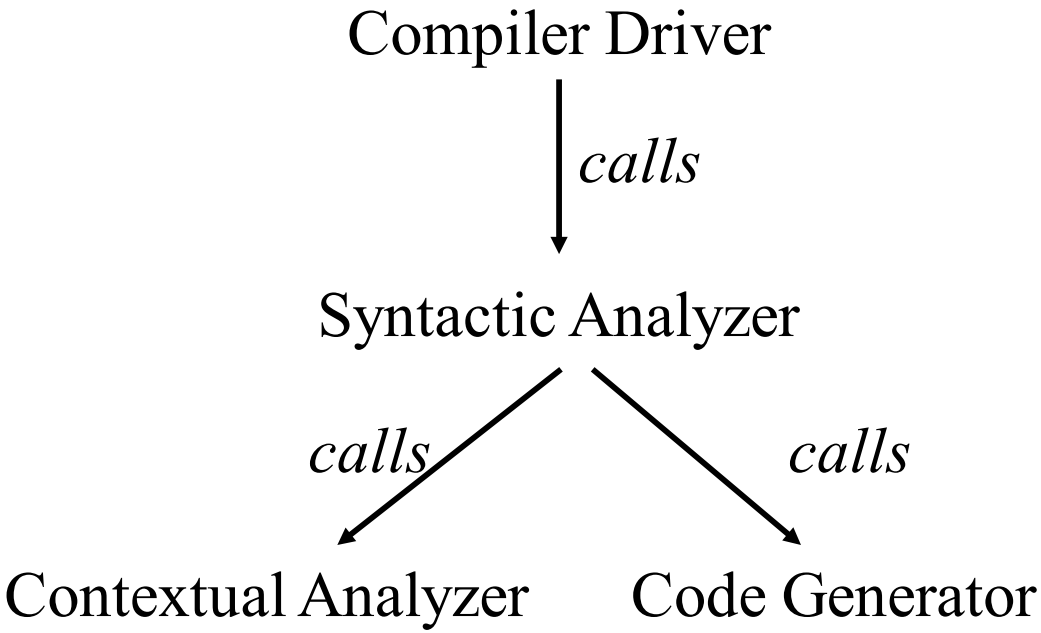


A: All of them!

Single Pass Compiler

A single pass compiler makes a single pass over the source text, parsing, analyzing and generating code all at once.

Dependency diagram of a typical Single Pass Compiler:



Programming Language and Compiler Design

- Many compiler techniques arise from the need to cope with some programming language construct
- The state of the art in compiler design also strongly affects programming language design
- The advantages of a programming language that's easy to compile:
  - Easier to learn, read, understand
  - Have quality compilers on a wide variety of machines
  - Better code will be generated
  - Fewer compiler bugs
  - The compiler will be smaller, cheaper, faster, more reliable, and more widely used
  - Better diagnostic messages and program development tools

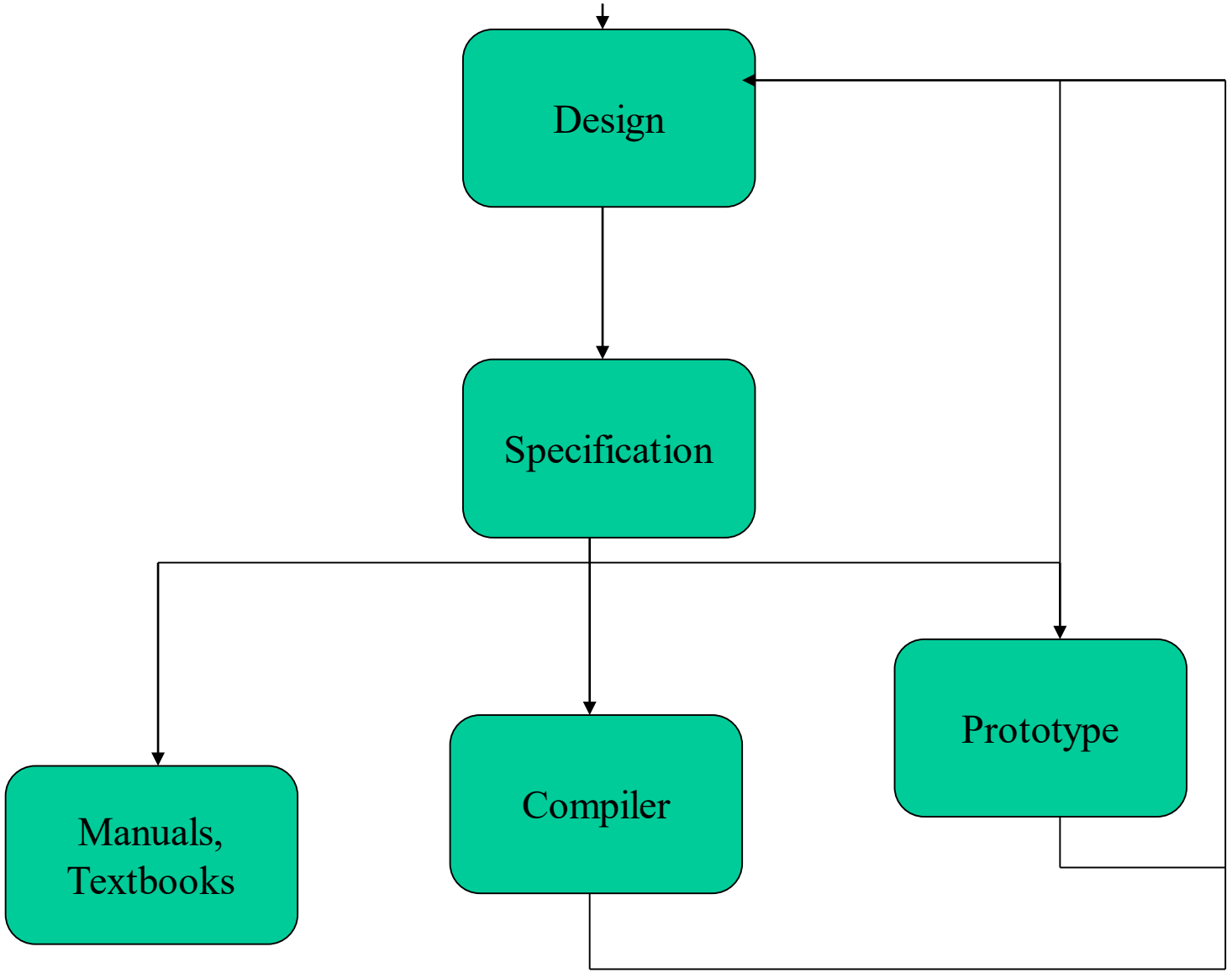
Compiler Writing Tools

- Compiler generators (compiler compilers)
  - Scanner generator
    - JLex (lex, lg)
  - Parser generator
    - JavaCUP (Yacc, pg)
  - Front-end generator
    - SableCC, JavaCC, (COCO/R, ANTLR, ..)
  - Code-generation tools
- Much of the effort in crafting a compiler lies in writing and debugging the semantic phases
  - Usually hand-coded

Programming Language Projects

- A good DAT4/SW4/IT8 project group can
  - Design a language (or language extensions)
  - Define the language syntax using CFG
  - Define the language semantics using SOS
  - Implement a compiler/interpreter
    - in Java (or C/C++, C#, SML, F#, Scala, Kotlin …)
    - Build a recursive decent parser by hand
    - Or using front-end tools such as Lex/Yacc, JavaCC, SableCC, ..
    - Do code generation for abstract machine
      - JVM (PerlVM or .Net CLR) or new VM
    - Or code generation to some high level language
      - C, Java, C#, SQL, XML
    - Or code generation for some hardware platform
      - MIPS, X86, ARM, ATmega, Z80, …
  - (Prove correctness of compiler)
    - Using SOS for Prg. Lang. and VM

Programming Language Life Cycle



Some advice

- A language design and compiler project is easy to structure.
  - Design phase (Lecture 1-5 + 13-14 + 19)
  - Front-end development (Lecture 6-9)
  - Contextual analysis (Lecture 10-12)
  - Code generation or interpretation (Lecture 15-18 + 20)
- You will learn the techniques and tools you need in time for you to apply them in your project

Summary

- Programming Language Design
  - New features
  - Paradigm, Philosophy
- Programming Language Specification
  - Syntax
  - Contextual constraints
  - Meaning (semantics and code generation)
- Programming Language Implementation
  - Compiler
  - Interpreter
  - Hybrid system

Important

- At the end of the course you should …
- Know
  - Which techniques exist
  - Which tools exist
- Be able to choose “the right ones”
  - Objective criteria
  - Subjective criteria
- Be able to argue and justify your choices!

Finally

Keep in mind, the compiler is the program from which all other programs arise. If your compiler is under par, all programs created by the compiler will also be under par. No matter the purpose or use -- your own enlightenment about compilers or commercial applications -- you want to be patient and do a good job with this program; in other words, don't try to throw this together on a weekend.

Asking a computer programmer to tell you how to write a compiler is like saying to Picasso, "Teach me to paint like you."

Sigh Nevertheless, Picasso shall try.