1. Preliminaries

1.1. Reasons for Studying Concepts of Programming Language

- *Increased capacity to express ideas.*

Computer programming language constructs define our ability to deliver products.
Example 1. C++ and Pascal versions of primes

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <cstring>
#include <cstdlib>
#include <cmath>
class Prime {
    int primecount;
    bool isPrime(int c)
    {   int factor=3;
        int maxfactor=(int)ceil(sqrt((double)c));
        while (factor<=maxfactor) {
            if (c%factor==0) return false;
            factor+=2;
        }
        return true;
    }
public:
    Prime(int n=100):primecount(n){}
    void Find(ostream& o)
    {   o << setw(5) << 2;
        int candidate=3;
        int count=1;
        while (count<primecount) {
            if (isPrime(candidate)) {
                count++;
                o << setw(5) << candidate;
                if (count%10==0) o << endl;
            }//end if
            candidate+=2;
        }//end while
    }//end Find
};//end class Prime
int main()
{   Prime P;
    P.Find(cout);
    return 0;
}
```

Figure 1.1.1 C++ version of prime

```
program prime;
    var
        candidate:integer;
        count:integer;
    function isprime(c:integer):boolean;
        var
            factor:integer;
            maxfactor:integer;
    begin{isprime}
        factor:=3;
        maxfactor:=round(sqrt(c));
        while (factor<=maxfactor) and (c mod factor <> 0) do factor:=factor+2;
        isprime:=factor>maxfactor;
    end{isprime};
begin{prime}
    write(2:5);
    count:=1;
    candidate:=3;
    while count<100 do
    begin
        if isprime(candidate) then
        begin
            count:=count+1;
            write(candidate:5);
            if count mod 10 = 0 then writeln
        end{if};
        candidate:=candidate+2
    end{while};
end{prime}.
```

Figure 1.1.2 Pascal version of prime

```
from math import sqrt
def isPrime(c) :
    factor=3
    maxfactor=sqrt(c)
    while factor<=maxfactor :
        if c%factor==0 :
        return False
        factor=factor+2
    return True
primecount=100
print("%5d" % (2),end="")
count=1
candidate=3
while count<primecount :
    if isPrime(candidate) :
        count=count+1
        print("%5d" % candidate,end="")
        if count%10==0 :
            print()
    candidate=candidate+2
```

Figure 1.1.3 Python version of prime

Example 2. C++ and Pascal versions of Stack ADT.

```
#ifndef stack_h
#define stack_h 1
#include <iostream>
struct stackerror {
    stackerror(char* m)
    {   cerr << "\nI am the stack and I am " << m << ".";
    }
};
class stack {
    struct element {
        element* prev;
        char v;
    };
    element* tos;
    void kill(element* e)
    {   if (!e) return;
        kill(e->prev);
        delete e;
    }
public:
    stack():tos(0) {}
    ~stack() {kill(tos);}
    bool full() {return 0;};
    bool empty(){return tos==0;}
    void push(char v)
    {   if (full()) throw stackerror("full");
        element* e=new element;
        e->v=v;
        e->prev=tos;
        tos=e;
    }
    char pop(void)
    {   if (empty()) throw stackerror("empty");
        element* e=tos;
        char v=e->v;
        tos=e->prev;
        delete e;
        return v;
    }
};
#endif
```

Figure 1.1.4 C++ version of Stack ADT: file **stack.h**

```
#include <iostream>
#include "stack.h"
int main()
{    stack s;
     s.push('e'); s.push('l'); s.push('b'); s.push('a');
     while (!s.empty()) cout << s.pop();
     cout << "\n";
     return 0;
}
```

Figure 1.1.5 C++ version of Stack ADT: file **p00.cpp**

```
program stack;
     uses WinCrt;
     type
          stack_p=^stack_e;
          stack_e=record
               prev:stack_p;
               v:char;
          end{stack_e};
     var s:stack_p;
     function stackcreate:stack_p; begin stackcreate:=nil  end;
     procedure stackdestroy(e:stack_p);
          var p:stack_p;
     begin{stackdestroy}
          while e<>nil do
          begin p:=e;
               e:=e^.prev;
               dispose(p)
          end{while}
     end{stackdestroy};
     function stackempty(s:stack_p):boolean; begin stackempty:=s=nil; end;
     procedure stackpush(var s:stack_p;v:char);
          var e:stack_p;
     begin{stackpush} new(e);
          e^.v:=v;
          e^.prev:=s;
          s:=e;
     end{stackpush};
```

Figure 1.1.6 Pascal version of stack ADT

```
    function stackpop(var s:stack_p):char;
        var v:char;  e:stack_p;
    begin{stackpop}
        if stackempty(s) then
            begin
                writeln('The stack is empty');
                stackpop:=chr(0)
            end
        else
            begin e:=s;
                v:=e^.v;
                s:=e^.prev;
                dispose(e);
                stackpop:=v
            end{if-else}
    end{stackpop};
begin{stack}
    s:=stackcreate;
    stackpush(s,'e');  stackpush(s,'l');  stackpush(s,'b');  stackpush(s,'a');
    while not stackempty(s) do write(stackpop(s));
    writeln;
    stackdestroy(s);
end{stack}.
```

Figure 1.1.6 Pascal version of stack ADT (continued)

- *Improved background for choosing appropriate languages.*

  Programming languages are designed for specific areas of application. Programmers without broad experience prefer languages that they know rather than languages tailored for an application.

Language	**Application**	**Notes**
Pascal	Teaching	Niklaus Wirth designed Pascal at Stanford University for the express purpose of teaching students structured programming.
C	Operating Systems	Dennis Ritchie designed C for the purpose of implementing the UNIX operating system.
COBOL	Business	Grace Hopper designed the Common Business Oriented Language to standardize military accounting programs.
FORTRAN	Scientific Programming	John Backus designed FORTRAN (FORmula TRANslation) for the express purpose of efficiently implementing computationally complex scientific programs

- *Increased ability to learn new languages.*

Programming languages have similar constructs. Most programming languages have only two abstractions: data and control. Most programming languages have methods to define data. Standard types are provided from which user defined types can be constructed. Most programming languages have a fixed set of statements that alter the flow of control. Examples of control statements include *assignment-statement*, *if-statement*, *case-statement*, *while-statement*, *for-statement*, and a *procedure-statement*.

- *Better understanding of the significance of implementation.*

  Compiler-writers are sought after by industry because of their broad and deep understanding of operating system internals and machine organization. Compiler-writers are known for their high productivity due to their encyclopedic understanding of the tools of their trade.

- *Better use of languages that are already known.*

- *Overall advancement of computing.*

Language	**Purpose**	**Notes**
Ada	Embedded Systems	Military applications suffered from poor reliability in critical applications. It was though that a strongly typed language would greatly improve reliability. In large systems, many programmers are employed to deliver an application in a reasonable time period. Interfaces must be designed first. Ada supports separate definition and implementation components, thus permitting an interface to evolve while the interface remains constant.
C++	Productivity	It has been recognized that testing occupies the largest part of the development cycle. To reduce the amount of testing on a new product, developers try to reuse existing similar code. Prior to the advent of object-oriented programming, code was copied, altered and applied to the new application. This process did not reduce the amount of testing significantly because small changes in code often dramatically affect the overall function of a product. Object-oriented programming provides a reliable and useful method of reusing existing code without alteration. Testing function built from objects that are already tested improves productivity.

Programming language is at the heart of computer science. Our ability to organize and deliver programs depend on ideas embedded in a programming language.

1.2. Programming Domains

1.2.1. Scientific Applications

- FORTRAN
- many floating point arithmetic operations
- simple data structures, two dimensional arrays representing matrices

- counting loops
- efficiency

1.2.2.Business Applications

- COBOL
- reports
- decimal types to represent currency
- character data to support reports
- spreadsheets, representing automatic ledgers
- database managers for inventory and business modeling

1.2.3.Artificial Intelligence

- LISP
- symbolic rather than numeric computing
- lists support symbolic computing
- dynamic code creation

1.2.4.Systems Programming

- PL/S (IBM proprietary dialect of PL/I)
- C (for UNIX)
- efficient
- low-level constructs to interface with electronic hardware

1.2.5.Web Software

- XHTML
- Java
- JavaScript, PHP

1.3. Language Evaluation Criteria

1.3.1.Readability

- Readability is critical to code maintenance. Programmers must be able to thoroughly understand code to find and fix errors and to add new capabilities. The more clearly an algorithm is represented in code, the easier it is to maintain.
- Programming languages are designed to implement a set of applications. FORTRAN was designed to support scientific applications. COBOL was designed to support business applications. Writing a business application in FORTRAN may make it difficult to read.

1.3.1.1. Overall Simplicity

- A small number of basic components make a programming language easier to understand than a language with a large number of components. A language that has many ways to alter the flow of control, a rich set of operators, many native data types and many ways to construct user-defined types will be harder to understand than a language with the opposite

characteristics. Programmers are usually satisfied with a subset of a large language.

- Several mechanisms to accomplish an operation diminish simplicity. For example, in C a user can increment an integer several ways.

```
count = count + 1;
count += 1;
count++;
++count;
```

- Operator overloading makes a programming language more complex. For example, it is possible in C++ for the adding operator to be defined for the following types, integral types and floating-point types, integral types scalar and vectors of the same type, and conformant vectors. The adding operator can be defined to add an element to a list. The adding operator can be defined to form the union of two sets. A programmer must be extremely careful when so many meanings are assigned to the adding operator.
- Reasonable structure is required to make a language readable. Control and data structures are obscured by the very simplicity of assembly language.

1.3.1.2. Orthogonality

- Orthogonality in a programming language means that primitive constructs are independent and can be combined with other constructs according to the rules defined for the construct. The assignment operator is nearly orthogonal in C. However one array cannot be assigned to another array of the same type and dimension. In a similar way, certain assembly languages lack orthogonality in the many instructions used to add two integer values depending on the location and size of the operands.

1.3.1.3. Data Types and Structures

- Data types that allow programmers to express values symbolically rather than numerically improve readability. For example, the inclusion of a Boolean type makes the second of the two statements more readable.

```
timeout = 1;
timeout = true;
```

Enumerated types are a logical extension of the Boolean type. Enumeration constants red, green and blue are understood more easily than equivalent integer constants 0, 1 and 2.

1.3.1.4. Syntax Design

- *Identifier forms*. Restricting the length of identifiers has been found to reduce readability.
- *Special words*. Concise and lucid control flow reserve words help readability. **while**, **for**, **if**, **then**, **else** and **case** are examples of well-chosen reserve words

that improve readability. In a similar way, well-chosen constructs for defining data help readability.

Nested blocks belonging to iterative or alternative statements that use identical syntax for the block are confusing. How often have we been notified of the syntax error "missing end" or missing "}"

- *Form and meaning*.
  Readability is improved when statements mean what they say. For example a variable that is declared **static** in C means that a single instance of that variable is allocated when the program starts and remains active although perhaps not visible until the program exits. The declaration **static** also directs the compiler to hide the name from the linker. When a function is declared **static** the second meaning is the only meaning that is applicable.

1.3.2.Writability

- Writability is a measure of how easily a language can be used to create an application. For example, COBOL is presumably suitable for writing business applications. FORTRAN has been used to implement many scientific applications.

1.3.2.1. Simplicity and Orthogonality

- Programmers often limit themselves to a subset of a programming language. A programmer will make fewer errors if well-known constructs are used. Fewer errors will be made if constructs are orthogonal. Fewer errors will be made if a construct is not dependent on neighboring constructs for its syntax or meaning.

1.3.2.2. Support for Abstraction

- The degree to which a programming language can be used to implement algorithmic and data abstraction determines its level of writability. Programming languages that make it possible to represent standard mathematical entities like sets and vectors are better than those programming languages where it is more difficult to do so.

1.3.2.3. Expressivity

- Expressivity is a measure of the ease with which common operations are expressed. count++ is easier to code than count = count + 1. A *for-statement* is easier to code than equivalent statements using a *while-statement*.

1.3.3.Reliability

- A programming language supports reliability if it provides mechanisms to detect and correct errors. Errors are less costly if they are removed earlier in the development cycle. An error that is detected and removed at compilation is less costly than one removed at run-time.

1.3.3.1. Type Checking

- Type checking is the process where types are validated during compilation. The number and type of arguments are matched against corresponding

formal parameters. Operations are checked to see if operands are acceptable. Boolean operators in Pascal can accept only Boolean operands. Arrays cannot be assigned to structures.

1.3.3.2. Exception Handling

- Handling errors that occur at run-time often requires significant and difficult design. The design task is reduced if a programming language has facilities to generate and handle exceptions.

1.3.3.3. Aliasing

- Two names that refer to the same storage, perhaps even having different types, is recognized as a source of errors that are difficult to find.

1.3.3.4. Readability and Writability

- Errors are more difficult to find in an application written in a language not suited to the task. For example, a compiler written in COBOL would be most difficult to maintain because COBOL is not naturally suited to the task of writing a compiler. A COBOL compiler would be both difficult to read and difficult to write.

1.3.4.Cost

- Cost of training programmers, simplicity and orthogonality apply. PL/I is difficult. Pascal is easy.
- Cost of writing programs, C++ is less costly than C.
- Cost of compiling programs, Ada was costly because the language was large and because the specification required extensive type checking.
- Cost of executing programs, PL/I is expensive because of the extensive run-time environment required. FORTRAN is cheap because the run-time environment is minimal and procedure prologs and epilogs are minimal.
- Cost of implementation. A programming language that requires a costly compiler will likely fail to achieve wide spread popularity.
- Cost of maintaining programs. Because Pascal is designed around a single compilation unit, it never achieved wide spread industrial applicability. Program maintenance is impossible with a single compilation unit. The program cannot be divided into manageable units. Only one programmer can be used to maintain a single compilation unit.
- Cost of portability. C achieved wide spread acceptability, in part, because of standardization.

1.4. Influences on Language Design

1.4.1.Computer Architecture

- Parallel versus serial hardware

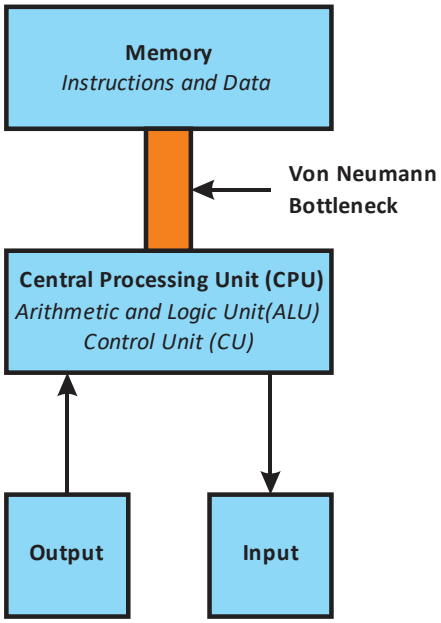

Figure 1.4.1 The von Neumann bottleneck

- Functional versus imperative programming
- Logic versus imperative programming

1.4.2.Programming Design Methodologies

- Structured programming (no gotos)
- Top-down design
- Step-wise refinement (compilation errors still reflect waterfall development strategies)
- Data flow design
- Data abstraction
- Object-oriented design

1.5. Language Categories

- Imperative (Pascal) serial, sequential, detailed specification
- Functional (Lisp) operations are specified by function alone, no side-effects, no data outside local data
- Logic (Prolog), rule-based, resolution engine
- Object-oriented (Smalltalk), classes, inheritance, polymorphism
- Mark up languages (HTML) are not programming languages, they do not specify computation

1.6. Language Design Trade-offs

- reliability and execution cost
- APL: expressivity and readability
- flexibility and safety

1.7. Implementation Methods

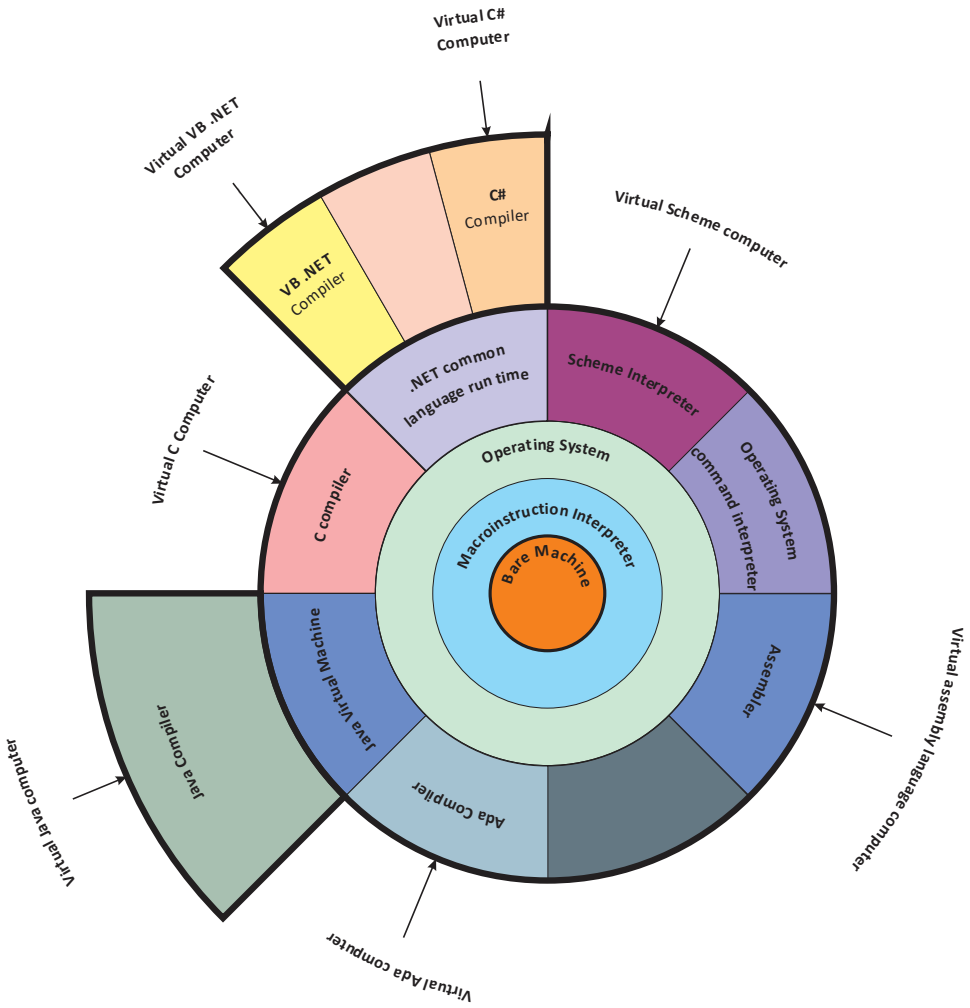

Figure 1.7.1 Layered interfaces

- Layers provide increasingly powerful levels of abstraction.
- At the lowest level the bare machine projects an abstraction consisting of detailed and minute electronic operations
- Macroinstructions allow other programs and users to understand the computer as an instruction set architecture
- The operating system manages all electronic hardware and presents a virtual interface for all facilities under its management
- Programming languages, including assembler, constrain the user to think of the capabilities of a computer through the lens of the language definition.

1.7.1.Compilation

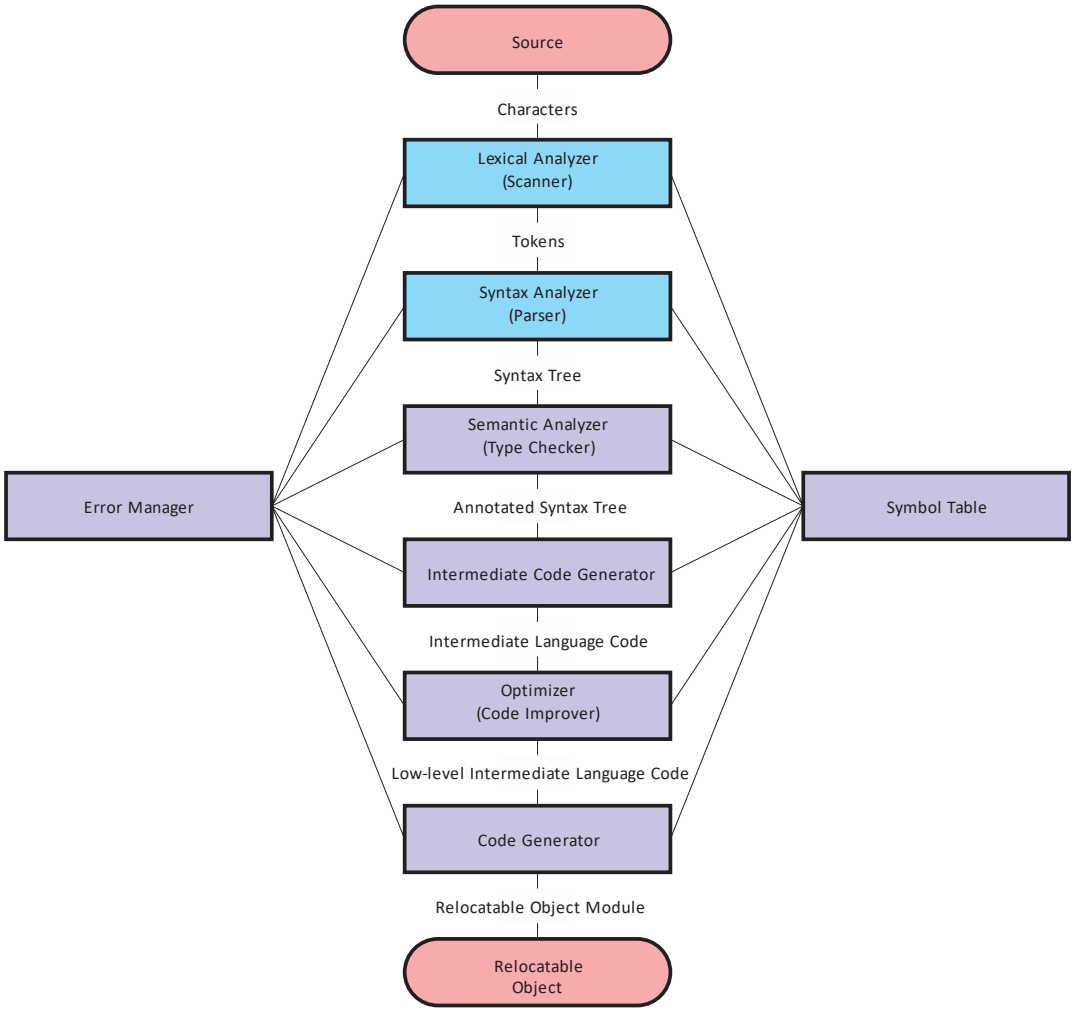

Figure 1.7.1 The compilation process

- The source is seen as a stream of characters. The *Lexical Analyzer* recognizes groups of tokens defined by regular expressions. The Lexical Analyzer in concert with the symbol table replaces each token with an equivalent non-negative integer value. A stream of integer tokens is sent to the *Syntax Analyzer*.
- The Syntax Analyzer determines if the input program is a sentence in the grammar for the programming language. The Syntax Analyzer builds a parse tree of the entire program.
- The *Semantic Analyzer* decorates the parse tree with type annotations to determine if the semantic rules have been observed.
- The *Intermediate Code* Generator produces an intermediate language form of the program. Examples of intermediate languages include P-Code and Java Byte Code.
- Ordinarily the Optimizer is optional. However, most compilers have a significant optimization phase. The Optimizer employs heuristics known to produce code that executes faster. Really, the Optimizer merely improves code: it does not produce optimal code in the mathematical sense of the word.

- The *Code Generator* selects instructions for the target machine. Some entities, functions, variables, are defined in other compilation units. The linkage editor, not shown in this diagram, resolves undefined external references. The Code Generator may also perform one last phase of optimization called peep-hole optimization. The result of compilation is called a relocatable object module. The relocatable means that all references to instructions and data can be relocated because they are relative to a value stored in a register.

1.7.2.Pure Interpretation

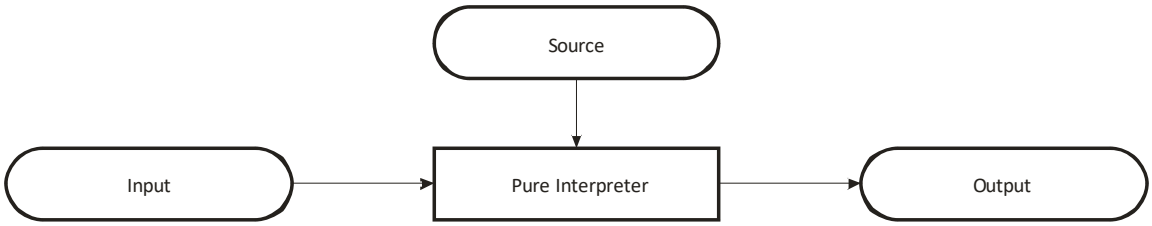

Figure 1.7.2 Pure interpretation

- A Pure Interpreter executes the source program directly. The Pure Interpreter acts as a compiler, operating system, and electronic hardware combined. The Symbol Computer created at Iowa State University is an example of a pure interpreter.

1.7.3.Hybrid Implementation Systems

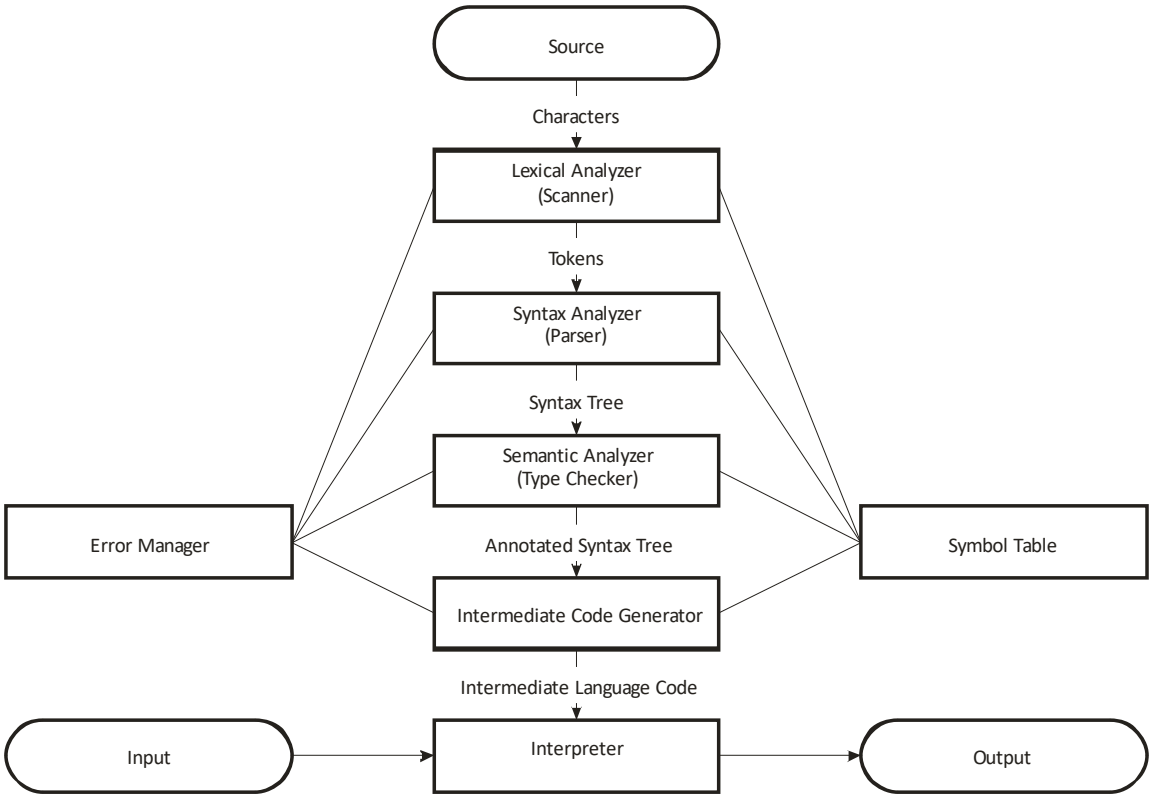

Figure 1.7.3 Hybrid implementation system

- A hybrid interpreter accepts intermediate language code rather than pure source or pure machine code. Examples of hybrid interpreters include the p-code interpreter for Pascal and Basic and the Java Byte Code machine for Java programs.

1.8. Programming Environments

- An integrated development environment (IDE) first made commercially available by Borland International in Turbo Pascal ® is an example of integrating the editor, compiler, and debugger in a single package. Other IDEs include Microsoft's Visual Studio and Common Desktop Environment for UNIX.