Compiler Design Introduction

conf. dr. ing. Ciprian-Bogdan Chirila
chirila@cs.upt.ro
http://www.cs.upt.ro/~chirila

Outline

- Language Processors
- The Structure of a Compiler
- The Evolution of Programming Languages
- The Science of Building a Compiler
- Applications of Compiler Technology
- Programming Language Basics
- Summary

Programming languages (PLs)

- PLs are notations to describe computation to
  - people
  - machines
- all software running on machines
  - is written in some PL
- before running
  - each program must be translated into a form that will be executed by a computer

Lecture topic

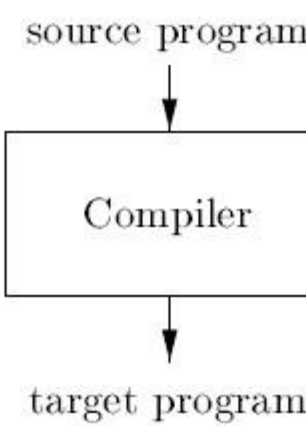
- to design compilers
- to implement compilers
- ideas to be used in the construction of translators
- wide variety of languages and machines
- the principles are applicable for other domains
- to be reused in the computer scientist career

Interacting domains

- programming languages
- machine architectures
- language theory
- data structures and algorithms
- software engineering

Language Processors

- Compiler
  - A compiler is a program that can read a program in one language – **the source language** – and translate it into an equivalent program in another language – **the target language**

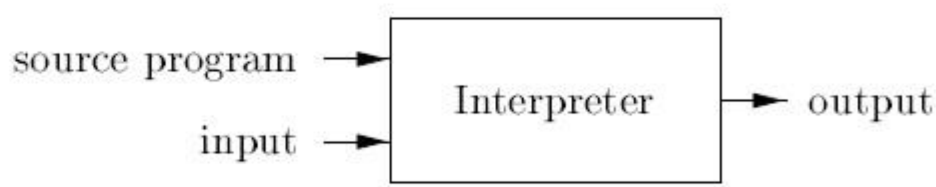


Language Processors

- Compiler
  - Reports errors in the **source program** that it detects during the translation process

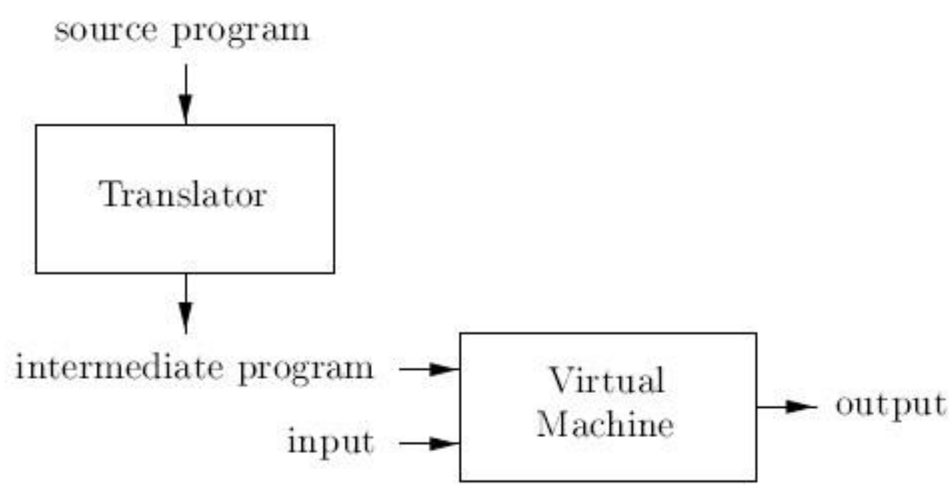
Language Processors

- Interpreter
  - Directly executes the operations specified in the source program on supplied inputs



Language Processors

- Hybrid compiler

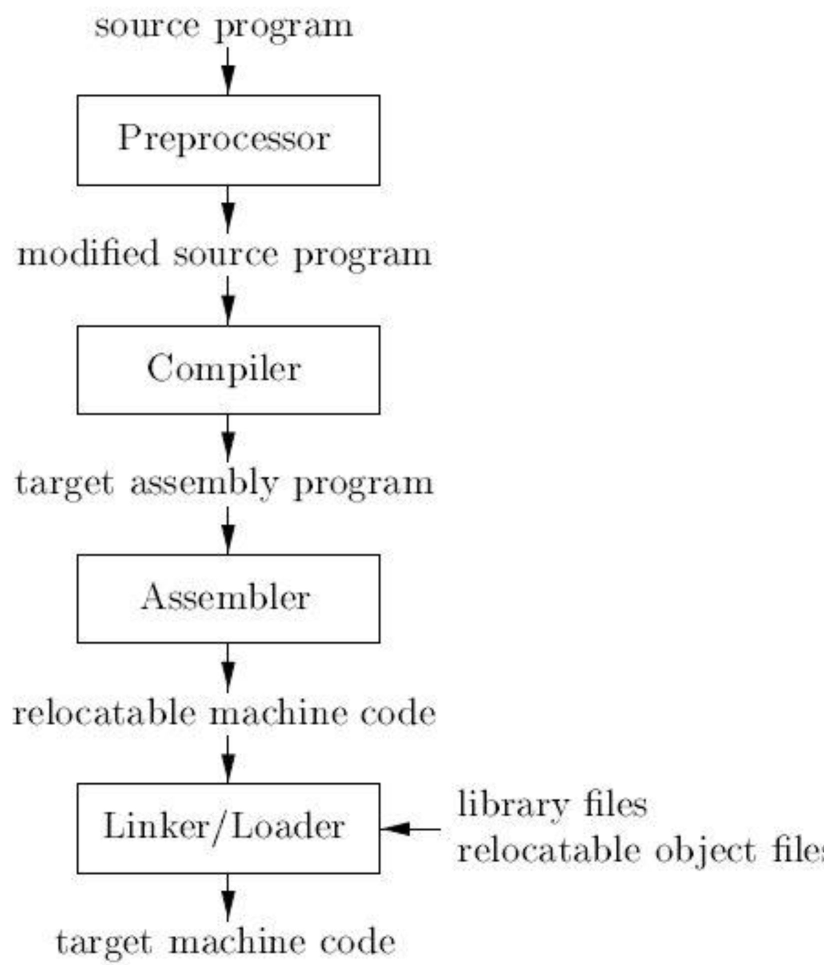


  - Java language processors combine **compilation** and **interpretation**

Language Processors

- Java language processors
  - a java source program is first compiled into *bytecodes*, which are then interpreted by a virtual machine
  - *Bytecodes* compiled on one machine can be interpreted on another machine
    - "Write once, run anywhere"
  - For faster processing, *just-in-time* compilers translate bytecodes into machine language immediately before they run the intermediate program to process the input

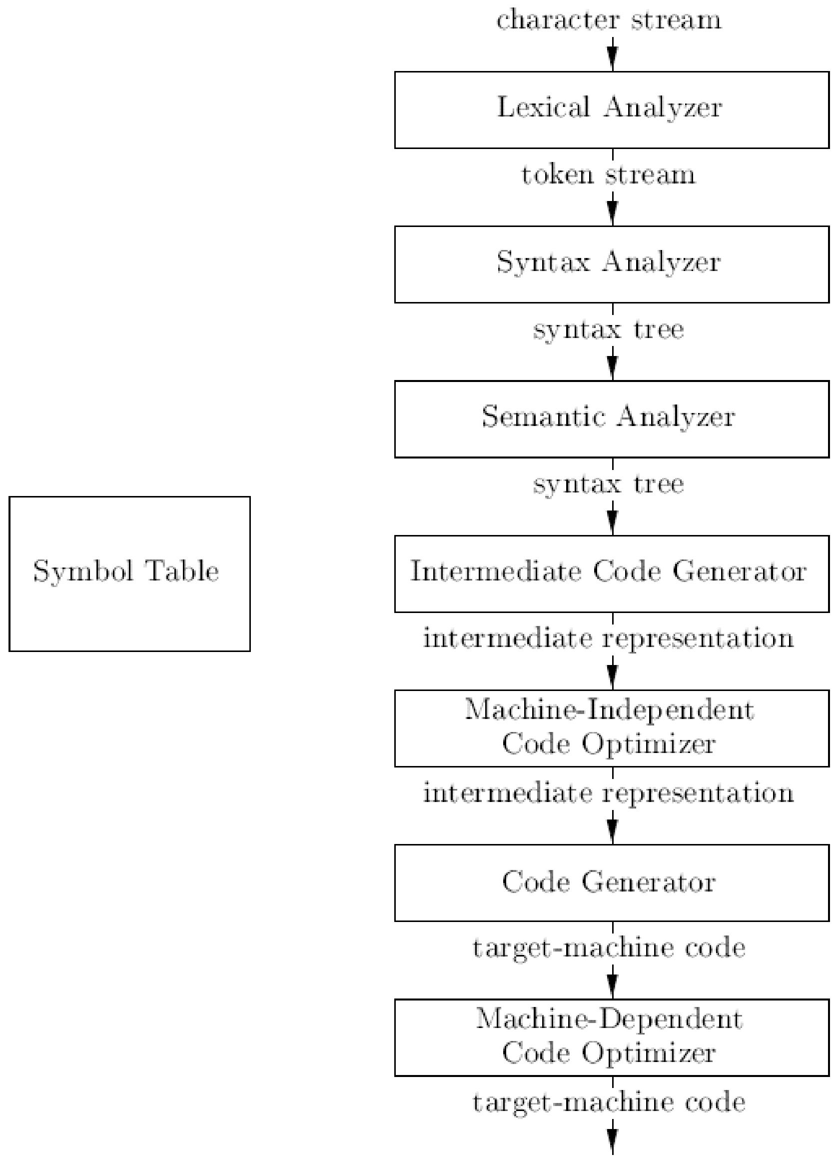
Language Processors



The Structure of a Compiler

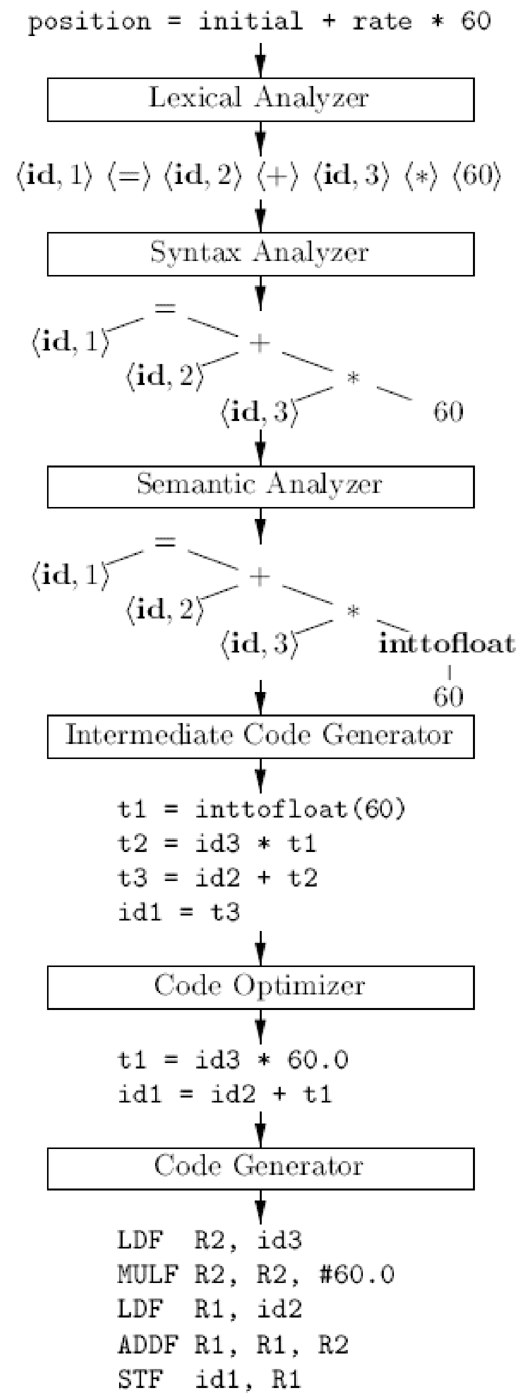
- Compiling is a 2 part process:
  - Analysis
    - Responsible for breaking up the source program into pieces and imposing a grammatical structure on them
    - If it detects errors, it provides informative messages
    - Collects data and stores it in a data structure called a *symbol table*
  - Synthesis
    - Constructs the desired target program from the intermediate representation and the information in the *symbol table*

The Structure of a Compiler



1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE

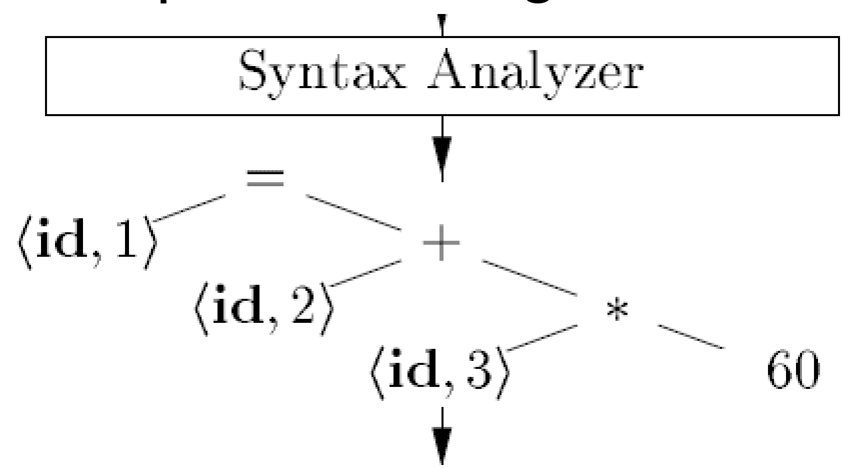


The Structure of a Compiler

- Lexical Analysis
  - Reads the stream of characters making up the source program and groups the characters into meaningful sequences called *lexemes*
  - For each lexeme, the output is a *token* which has the following form:
    - <token-name, attribute-value>

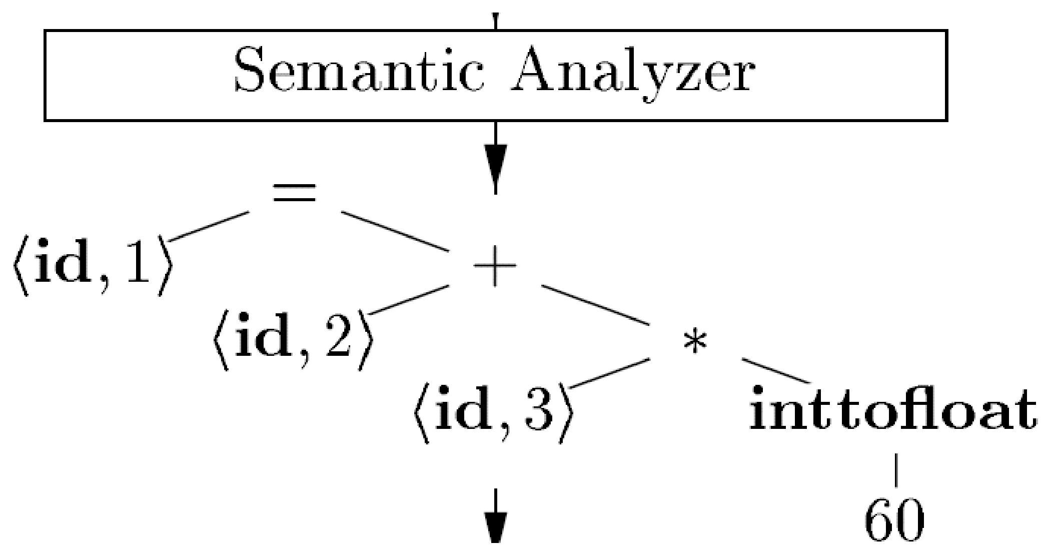
The Structure of a Compiler

- Syntax Analysis
  - It creates a tree-like intermediate representation using the first components of the tokens produced by the lexical analyzer
  - **Syntax tree**
    - each interior node represents an operation and the children of the node represent the arguments of the operation



The Structure of a Compiler

- Semantic Analysis
  - Checks the source program for semantic consistency with the language definition
  - Type checking – checks whether each operator has matching operands
  - Conversions



The Structure of a Compiler

- Intermediate code generation

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

- Code optimization

```
t1 = id3 * 60.0
id1 = id2 + t1
```

- Code generation

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

The Structure of a Compiler

- Symbol-Table Management
  - Recording variable names and collecting information about attributes
    - storage allocated for a name, its scope, its type, number and types of arguments for functions, pass by value or by reference, returned type
- Grouping of Phases into Passes
  - Front-end pass
    - lexical analysis to intermediate code generation
  - Code optimization
    - optional pass
  - Back-end pass
    - code generation

The Evolution of Programming Languages

- 1940's
  - First electronic computers
  - Machine language, sequences of 0 and 1
  - Basic operations
    - move data, add 2 registers, compare 2 values
  - Slow, hard to modify, error prone, tedious
- 1950's
  - Mnemonic assembly languages
  - First step towards higher level languages with Fortran, Cobol, Lisp

The Evolution of Programming Languages

- Classification by
  - Generation
    - First-generation (machine languages)
    - Second-generation (assembly languages)
    - Third-generation (Fortran, Cobol, Lisp, C, C++, C#, Java)
    - Fourth-generation (NOMAD for reports, SQL for queries, Postscript for text formatting)
    - Fifth-generation (logic and constraint based languages like Prolog, OPS5)

The Evolution of Programming Languages

- Classification
  - By programming
    - Imperative (how the computation is to be done)
      - C, C++, C#, Java
    - Declarative (what computation is to be done)
      - ML, Haskell, Prolog
  - von Neumann languages
    - Fortran, C
  - Object-oriented languages
    - Simula67, Smalltalk, C++, C#, Java, Ruby
  - Scripting languages
    - Awk, Javascript, Perl, PHP, Python, Ruby, Tcl

The Science of Building a Compiler

- Fundamental models
  - Finite-state machines
  - Regular expressions
  - Context-free grammars
  - Trees

The Science of Building a Compiler

- Code optimization
  - The result must be code that is more efficient than the obvious code
  - Optimization has become more important and complex because of massively parallel computers, multicore machines
  - Graphs, matrices, linear programs are necessary models to produce optimized code

The Science of Building a Compiler

- Code optimization
  - Design objectives
    - Correct optimization (preserve the meaning)
    - Improved performance
    - Compilation time must be reasonable
    - Manageable required engineering effort
- Compiler development involves both theory and experimentation

Applications of Compiler Technology

- High-Level programming languages implementation
  - **Higher-level** programming languages are easier to program in, but are less efficient
  - **Low-level** programs are harder to write, less portable, harder to maintain, error prone but they do offer more control and produce more efficient code (in principle)

Applications of Compiler Technology

- High-Level programming languages implementation
  - Data-flow optimizations have been developed to analyze the flow of data and remove redundancies from arrays, structures, loops, procedure invocations
  - Object orientation (C++, C#, Java)
    - Makes programs more modular, easier to maintain
    - Main features are:
      - Abstraction
      - Inheritance

Applications of Compiler Technology

- High-Level programming languages implementation
  - Procedure inlining
    - The replacement of a procedure call by the body of the procedure
  - Optimizations to speed up virtual method dispatches

Applications of Compiler Technology

- High-Level programming languages implementation
  - Example Java
    - Type safe, array accesses are checked to be within bounds, no pointers, garbage collector
    - Easier programming, but incur run-time overhead
    - Optimizations to run-time include
      - Eliminating unnecessary range checks
      - Allocating objects not accessible beyond a procedure on stack instead of heap
      - Minimizing overhead of garbage collection
      - Dynamic optimization

Applications of Compiler Technology

- Optimizations for Computer Architectures
  - Parallelism
    - Instruction level
    - Processor level
    - Achieved by programmers writing multithreaded code for multiprocessors or parallel code can be automatically generated by a compiler
    - Great benefits for
      - Scientific computing
      - Engineering applications

Applications of Compiler Technology

- Memory Hierarchies
  - Levels of storage with different speeds and sizes
    - Registers (hundreds of bytes)
    - Caches (KB to MB)
    - Physical memory (MB to GB)
    - Secondary storage (GB, TB)
  - Using registers correctly is the most important issue in optimization

Applications of Compiler Technology

- Design of new Computer Architectures
  - In modern architecture development, compilers are developed in the processor-design stage and compiled code, on simulators is used to evaluate the architecture
  - RISC (Reduced Instruction-Set Computer) architecture
    - Simple instruction sets
    - PowerPC, SPARC, Alpha, MIPS are architectures based on RISC concept
    - x86 is based on CISC(Complex Instruction-Set Computer) but many of the ideas developed for RISC machines are used
  - Specialized Architectures
    - Data flow machines
    - Vector machines
    - VLIW(Very Long Instruction Word)
    - SIMD(Single Instruction, Multiple data)
    - Multiprocessors with shared memory
    - Multiprocessors with distributed memory

Applications of Compiler Technology

- Program Translations
  - Binary translation
    - Used for increasing the availability of software for different machines, with different instruction sets
    - x86 to Alpha or Sparc code
    - x86 to VLIW code (Transmeta Crusoe processor)
  - Hardware Synthesis
    - Hardware designs are described at register transfer level (RTL)
    - RTL descriptions -> gates ->mapped to transistors -> physical layout

Applications of Compiler Technology

- Program Translations
  - Database Query Interpreters (SQL)
  - Compiled Simulation
    - Instead of writing a simulator that interprets the design, it is faster to compile the design to machine code that simulates that design
    - Compiled simulations run orders of magnitude faster than interpreter approaches
    - Used in tools that simulate designs written in Verilog or VHDL

Applications of Compiler Technology

- Software Productivity Tools
  - Data-flow analysis to find errors
    - Type checking
      - Operation applied to wrong type of object
      - Parameters passed do not match signature of method
      - Check for security flaws
    - Bounds checking
      - Buffer overflows can cause security breaches in C
    - Memory-Management tools
      - Garbage collection solves memory management errors

Programming Language Basics

- Static/Dynamic Distinction
  - Static policy
    - Issues decided at compile time
  - Dynamic policy
    - Issues decided at run time

    Example:

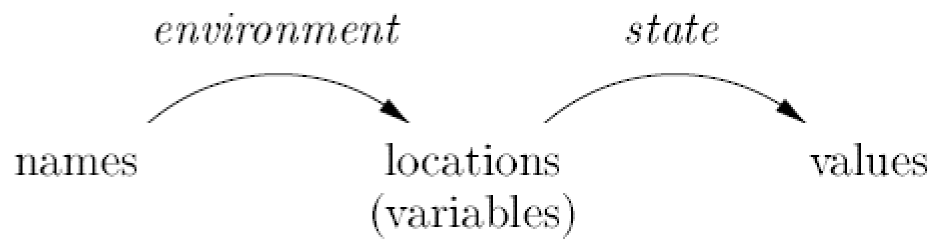
    public static int x; (Java)

    Here x is a class variable (there is only one copy of x, at one location, no matter how many objects of the class are created).

    If it wouldn't have been static, each object would have a different location for x and the compiler would determine them at run time (instance variable).

Programming Language Basics

- Environments and States



  - Binding of names to locations
    - Dynamic generally, but global variables can be given a location once and for all
  - Binding of locations to values
    - Dynamic with the exception of declared constants
      - #define ARRAYSIZE 500 (static)

Programming Language Basics

- Names, identifiers, variables
  - Compile-time names
  - Identifier
    - String of characters, letters or digits that refers to a data object, a procedure, a class, a type
    - All identifiers are names, not all names are identifiers
  - Variable
    - It refers to a particular location of the store
    - Run-time location denoted by names

Programming Language Basics

- Static Scope and Block Structure
  - In C the scope is determined by where the declaration appears
  - In C++, JAVA, C# we have public, private, protected

```
main() {
    int a = 1;                                    B1
    int b = 1;
    {
        int b = 2;                          B2
        {
            int a = 3;              B3
            cout << a << b;
        }
        {
            int b = 4;              B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Programming Language Basics

- Dynamic Scope
  - Macro expansion in C preprocessor

```
#define a (x+1)
int x = 2;
void b() { int x = 1; printf("%d\n", a); }
void c() { printf("%d\n", a); }
void main() { b(); c(); }
```

  - Output: 2 3
  - Another example of dynamic policy would be method resolution in object-oriented programming

Programming Language Basics

- Parameter Passing Mechanisms
  - Call by value
    - The actual parameter is evaluated or copied
    - The value is placed in the location belonging to the corresponding formal parameter of the called procedure
  - Call by reference
    - The address of the actual parameter is passed to the callee as the value of the corresponding formal parameter
    - Necessary when the formal parameter is a large object, array of structure in C/C++

Programming Language Basics

- Aliasing
  - Call by reference
    - 2 formal parameters -> same location (they are aliases of one another)
  - Essential if a compiler is to optimize a program

Summary

- Language Processors
  - Profilers, debuggers, loaders, linkers, assemblers, interpreters, compilers are included in an integrated development environment
- Compiler phases
  - A compiler works as a sequence of phases, each of which modifies the source program from one form to another
- Machine and Assembly Languages
  - 1st machine languages
  - 2nd assembly languages
  - Slow programming, error prone

Summary

- Modeling in Compiler Design
  - Automata, grammars, regular expressions, trees are models found useful
- Code Optimization
  - Important for the study of compilation
- Higher-Level Languages
  - Languages take more and more tasks such as memory management, type-consistency checking, parallel execution of code
- Compilers and Computer Architecture
  - Compilers influence architecture
  - Modern innovations in architecture depend on compilers to use hardware capabilities effectively

Summary

- Software productivity and software security
  - Program-analysis tasks such as detecting bugs, discovering vulnerabilities
- Scope rules
  - Static scope (lexical scope) if it's possible to determine the scope of a declaration by looking only at the program
  - Dynamic scope
- Environments and states
  - Environments
    - Associations of names with locations in memory and then with values
    - map names to locations in store
  - States
    - Map locations to their values

Summary

- Block Structure
  - Nested blocks -> block structure
- Parameter passing
  - Parameters are passed either by value of by reference.
  - When dealing with large objects passed by value, the values passed are references to objects themselves, resulting in an effective call by reference
- Aliasing
  - When passing by reference, 2 formal parameters can refer to the same object, thus allowing a change in one variable to change another

Bibliography

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman – Compilers, Principles, Techniques and Tools, Second Edition, 2007