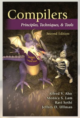
Compilers: Principles, Techniques, and Tools

Chapter 1 Introduction

Dongjie He
University of New South Wales
https://dongjiehe.github.io/teaching/compiler/

29 Jun 2023

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY · AUSTRALIA



Evolution of Programming Languages

- **Machine Languages**: 1940's, sequences of 0's and 1's
  - programming was slow, tedious, and error-prone.
  - programs were hard to understand and modify.
- **Assembly Languages**: early 1950's
  - mnemonics of machine instructions
  - macros: parameterized shorthands for frequently used instructions
- **High-Level Languages**: 1955 to present
  - programming is easier, more natural, and more robust.
  - **General Purpose Languages**: Fortran, Cobol, Lisp, C, C++, Java, ...
  - **Domain-Specific Languages**: NOMAD, SQL, Postscript, ...
  - **Logic- and constraint-based Languages**: Prolog, OPS5, ...
  - Other Classification: 0
    - **Imperative languages**: C, C++, Java, Rust, ...
    - **Functional languages**: ML, Haskell, OCaml, ...
    - **Constraint logic languages**: Prolog, Datalog, ...
  - Von Neumann language/Object-oriented language/Scripting language

Programming Language Basics I

- A language *policy* allows the compiler to decide an issue.
  - **Static policy**: the issue is decided at *compile time*.
  - **Dynamic policy**: the issue is decided at *run time*.
  - e.g., "`static int x;`", "static" in Java enables the compiler to determine the location of 'x' in memory.
- The *scope* of a declaration of x is the region in which uses of x refer to this declaration.
  - **Static scope**: can be determined by looking only at the program.
  - **Dynamic scope**: determined by the program runs.

```
int i; // global i
void f(...) {
   int i; // local i
   i = 3; // use of local i
}
x = i + 1; // use of global i
```

A C/Java Program

```
(defvar x 1) ; global variable 'x' with value 1
(defun foo () (message "The value of 'x' in foo: %s" x))
(defun bar ()
   (let ((x 2)) ; local variable 'x' with value 2
   (foo))) ; Call foo from within bar
(bar) ; Call bar from the global scope
```

An Emacs-Lisp Program

Programming Language Basics II

- A *block* is a grouping of declarations and statements.
  - C family languages use braces { and } to delimit a block.
  - Algol and Pascal use **begin** and **end**.
- Syntax of blocks in C

$$\textit{stmt} := \textit{block} \mid \cdots \qquad \textit{block} := \textit{declarations stmts}$$

- **Block structure**: blocks nested inside each other
- Static-scope of a declaration D (which belongs to block B) of name x
  - B is the most closely nested block containing D
  - the scope of D is all of B, except for any blocks B' nested to any depth within B, in which x is redeclared.

```
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;
}
```

DECLARATION	SCOPE
`int a = 1;`	$B_1 - B_3$
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	$B_2 - B_4$
`int a = 3;`	B_3
`int b = 4;`	B_4

Programming Language Basics III

- Member Scope in Classes/Structures
  - the scope of a member declaration x in a class C extends to any subclass C', except if C' has a local declaration of the same name x
- Explicit Access Control
  - **public/protected/private**
- Parameter Passing Mechanisms
  - Actual Parameters/Formal Parameters
    - e.g., "`A id(A p) {return p;} r = id(a);` "

```
#include <stdio.h>
void incrementByValue(int num) {
   num = num + 1;
   printf("Inside function: %d\n", num);
}
int main() {
   int x = 5;
   printf("Before function call: %d\n", x);
   incrementByValue(x);
   printf("After function call: %d\n", x);
   return 0;
}
```

(a) Call by Value

```
#include <stdio.h>
void incrementByReference(int* num) {
   (*num) = (*num) + 1;
   printf("Inside function: %d\n", *num);
}
int main() {
   int x = 5;
   printf("Before function call: %d\n", x);
   incrementByReference(&x);
   printf("After function call: %d\n", x);
   return 0;
}
```

(b) Call by Reference

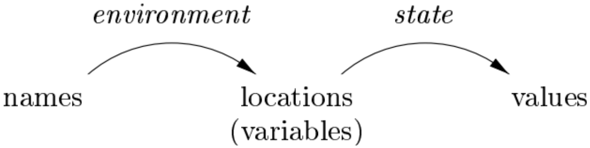
Algol-60

```
procedure swap(a,b);
   integer a, b;
begin
   integer tmp;
   tmp := a; a := b; b := tmp;
end;
integer i, x;
integer array arr[0:9];
x := 2;
for i := 0 step 1 until 9 do arr[i] := 10 - i;
swap(x, arr[x]);
print(x);  printArray(arr);
```

(c) Call by Name

Programming Language Basics III

- Environments and States
  - **environment**: a mapping from names to locations
  - **state**: a mapping from locations to their values



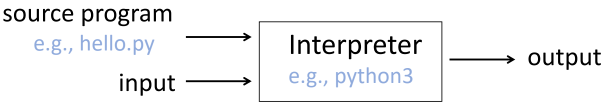
  - e.g., "`x = y + 1`" changes the value in the location denoted by name `x`
- Aliasing
  - *x* and *y* are *aliases* of each other if they can refer to the same location

```
#include <stdio.h>
int main() {
  int x[3] = {1, 2, 3};
  int* y = x;
  y[1] = 4;
  printf("%d", x[1]);
  return 0;
}
```

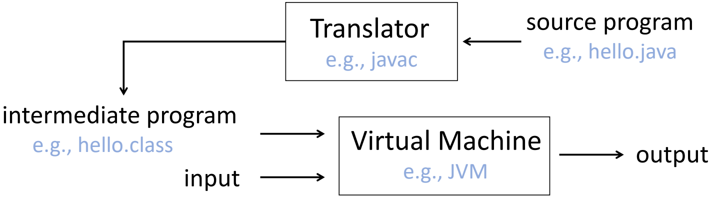
Compilers and Interpreters

- **Compilers**

- **Interpreters**



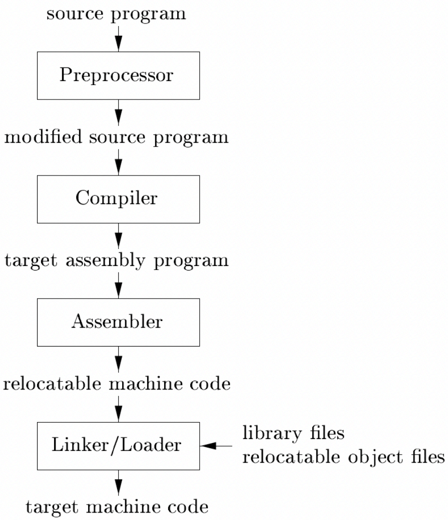
- **Hybrid Compilers**



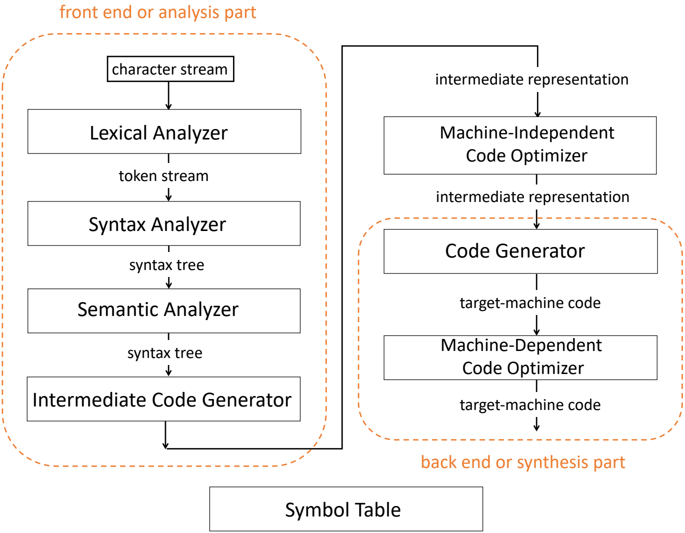
Other Language Processors

- **Preprocessors**: C Preprocessor, "/usr/bin/cpp"
  - modify source code before compilation
  - allow to write code in a more convenient and expressive manner
  - macro expansion, conditional compilation, file inclusion, constant or variable substitution
- **Assemblers**
  - translate assembly language code into relocatable machine code
  - specific to the target architecture or processor
- **Linkers**: GNU Linker, "/usr/bin/ld"
  - combine object files to create an executable program or a library
  - static linking/dynamic linking
- **Loaders**: put all executable object files into memory for execution
- **Debuggers**: GNU Debugger, "/usr/bin/gdb"
  - Breakpoints, Stepping, Variable or Memory inspection, Call stack analysis, ...

A Language-Processing System

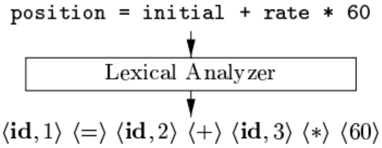


Phases of a Compiler



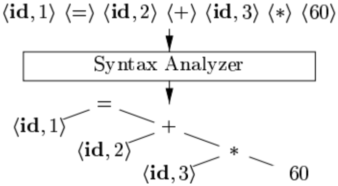
Lexical Analysis

- **lexical analysis/scanning**
  - grouping characters into meaningful *lexemes* sequences
  - *lexeme* $= \langle$*token-name*, *attribute-value*$\rangle$
  - *token-name*: abstract symbol used during syntax analysis
  - *attribute-value*: stored in symbol-table and used in semantic analysis and code generation
- An example



Syntax Analysis

- *syntax*: describes the proper form of programs
- **syntax analysis/parsing**
  - transform tokens into an Intermediate Representation, e.g., *syntax tree*
  - IR depicts the grammatical structure of the token stream
- An example

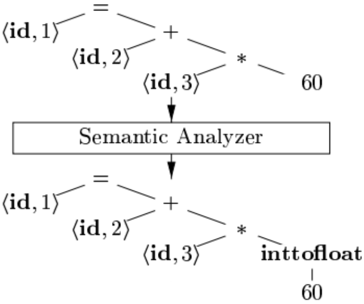


Semantic Analysis

- *semantics*: define what programs mean
- **semantic analysis**
  - check semantic consistency with the language definition
  - gather and save type information in syntax tree or symbol table
  - *type checking* and *type conversions* (*coercions*)
- An example

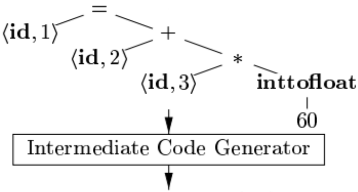
1	`position`	···
2	`initial`	···
3	`rate`	···

SYMBOL TABLE



Intermediate Code Generation

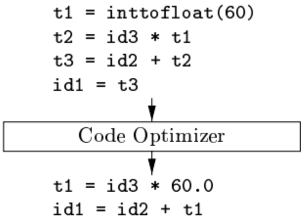
- generate low-level/machine-like intermediate representation
- a program for an abstract machine
- easy to produce and easy to translate into target machine
- An example: *three-address code*




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

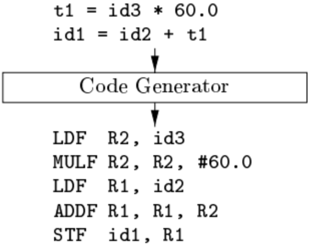
Code Optimization

- machine-independent/machine-dependent code-optimization
- **objectives**: produce better target code
    - faster running time
    - shorter target code
    - consuming less power
    - ...
- An example



Code Generation

- map an intermediate representation into target language
  - instruction selection
  - register allocation



Symbol-Table Management

- **Symbol table**:
  - a data structure recording each variable name and attributes, including storage, type, scope, ...

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

SYMBOL TABLE

  - should be efficient for finding records and storing or retrieving data from the records.

Applications of Compiler Technology

- Compiler design is not only about compilers
- Applications
  - Implementation of High-Level Programming Languages
    - support increased levels of programming abstractions
    - e.g., data abstraction and inheritance in Java, type system in Rust
  - Optimizations for Computer Architectures
    - *parallelism*: instruction level and processor level
    - *memory hierarchy*: closest to the processor being fastest but smallest
  - Design of New Computer Architectures
    - RISC architecture
    - Specialized architecture, e.g., VLIW, SIMD, vector machine, ...
  - Program Translations
    - Binary translation: translate binary code for one machine to another
    - Hardware synthesis: translate RTL descriptions into gates
    - Database Query Interpreters, Compiled Simulation
  - Software Productivity Tools
    - e.g., bug detection, type checking, bounds checking, ...

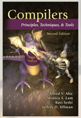
Compilers: Principles, Techniques, and Tools

Chapter 1 Introduction

Dongjie He
University of New South Wales
https://dongjiehe.github.io/teaching/compiler/

29 Jun 2023

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY·AUSTRALIA



Lab 1: Get Familiar with LLVM's Code Structure

- LLVM Project, `https://llvm.org/`, is a collection of modular and reusable compiler and toolchain technologies.
  - **LLVM Core**: a modern source- and target-independent optimizer
  - **Clang**: an LLVM native C/C++/Objective-C compiler
  - **LLD**: a linker
  - ...
- Task 1: install LLVM on your machine
  - `https://llvm.org/docs/GettingStarted.html#getting-the-source-code-and-building-llvm`
- Task 2: getting familiar with LLVM's code structure