Compiler Construction Tutorial 1

Contents

Lecture Note

Q1.
Explain P18
Q2
Explain P14

tutorial website

Exercise from the Dragon Book

Exercises for Section 1.1

1.1.1

What is the difference between a compiler and an interpreter?

Answer

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 and report any errors in the source program that it detects during the translation process.

Interpreter directly executes the operations specified in the source program on inputs supplied by the user.

1.1.2

What are the advantages of: (a) a compiler over an interpreter (b) an interpreter over a compiler?

Answer

a. The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.

b. An interpreter can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

1.1.3

What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?

Answer

The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.

1.1.4

A compiler that translates a high-level language into another high-level language is called a *source-to-source* translator. What advantages are there to using C as a target language for a compiler?

Answer

For the C language there are many compilers available that compile to almost every hardware.

1.1.5

Describe some of the tasks that an assembler needs to perform.

Answer

It translates from the assembly language to machine code. This machine code is relocatable.

Exercises for Section 1.3

1.3.1

Indicate which of the following terms:

a. imperative b. declarative c. von Neumann d. object-oriented e. functional f. third-generation g. fourth-generation h. scripting

apply to which of the following languages:

1. C
2. C++
3. Cobol
4. Fortran
5. Java
6. Lisp
7. ML
8. Perl
9. Python
10. VB.

Answer

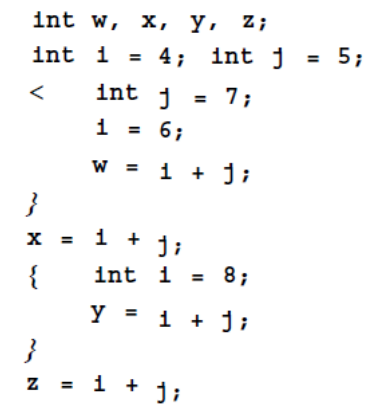
imperative: C, C++

object-oriented: C++, Java

functional: ML

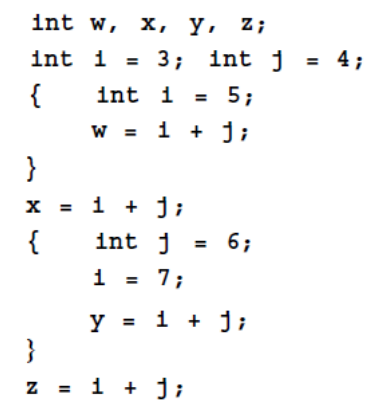
scripting: Perl, Python

Exercises for Section 1.6



```
int w, x, y, z;
int i = 4; int j = 5;
{   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{   int i = 8;
    y = i + j;
}
z = i + j;
```

(a) Code for Exercise 1.6.1



```
int w, x, y, z;
int i = 3; int j = 4;
{   int i = 5;
    w = i + j;
}
x = i + j;
{   int j = 6;
    i = 7;
    y = i + j;
}
z = i + j;
```

(b) Code for Exercise 1.6.2

1.6.1

For the block-structured C code below, indicate the values assigned to w, x, y, and z.

```
int w, x, y, z;
int i = 4; int j = 5;
{
```

```
  int j = 7;
  i = 6;
  w = i + j;
}
x = i + j;
{
  int i = 8;
  y = i + j;
}
z = i + j;
```

Answer

w = 13, x = 11, y = 13, z = 11.

1.6.2

Repeat Exercise 1.6.1 for the code below.

```
int w, x, y, z;
int i = 3; int j = 4;
{
  int i = 5;
  w = i + j;
}
x = i + j;
{
  int j = 6;
  i = 7;
  y = i + j;
}
z = i + j;
```

Answer

w = 9, x = 7, y = 13, z = 11.

1.6.3

Figure 1.13: Block-structured code

```
int w, x, y, z;      /* Block B1 */
{   int x, z;        /* Block B2 */
    {   int w, x;    /* Block B3 */ }
}
{   int w, x;        /* Block B4 */
    {   int y, z;    /* Block B5 */ }
}
```

Figure 1.14: Block structured code for Exercise 1.6.3

For the block-structured code of Fig. 1.14, assuming the usual static scoping of declarations, give the scope for each of the twelve declarations.

Answer

```
Block B1:
        declarations:  ->   scope
                w                               B1-B3-B4
                x                               B1-B2-B4
                y                               B1-B5
                z                               B1-B2-B5
Block B2:
        declarations:  ->   scope
                x                               B2-B3
                z                               B2
Block B3:
        declarations:  ->   scope
                w                               B3
                x                               B3
Block B4:
        declarations:  ->   scope
                w                               B4
                x                               B4
Block B5:
        declarations:  ->   scope
```

y	B5
z	B5

1.6.4

Exercise 1.6.4: What is printed by the following C code?

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

What is printed by the following C code?

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

Answer

3

2

Summary of Chapter 1

- *Language Processors.* An integrated software development environment includes many different kinds of language processors such as compilers, interpreters, assemblers, linkers, loaders, debuggers, profilers.

- *Compiler Phases. A* compiler operates as a sequence of phases, each of which transforms the source program from one intermediate representation to another.

4> *Machine and Assembly Languages.* Machine languages were the first-generation programming languages, followed by assembly languages. Programming in these languages was time consuming and error prone.

4 *Modeling in Compiler Design.* Compiler design is one of the places where theory has had the most impact on practice. Models that have been found useful include automata, grammars, regular expressions, trees, and many others.

4 *Code Optimization.* Although code cannot truly be "optimized," the science of improving the efficiency of code is both complex and very important. It is a major portion of the study of compilation.

4 *Higher-Level Languages.* As time goes on, programming languages take on progressively more of the tasks that formerly were left to the programmer, such as memory management, type-consistency checking, or parallel execution of code.

4 *Compilers and Computer Architecture.* Compiler technology influences computer architecture, as well as being influenced by the advances in architecture. Many modern innovations in architecture depend on compilers being able to extract from source programs the opportunities to use the hardware capabilities effectively.

4 *Software Productivity and Software Security.* The same technology that allows compilers to optimize code can be used for a variety of program-analysis tasks, ranging from detecting common program bugs to discovering that a program is vulnerable to one of the many kinds of intrusions that "hackers" have discovered.

4 *Scope Rules.* The *scope* of a declaration of x is the context in which uses of x refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses *dynamic scope.*

4 *Environments.* The association of names with locations in memory and then with values can be described in terms of *environments,* which map names to locations in store, and *states,* which map locations to their values.

4- *Block Structure.* Languages that allow blocks to be nested are said to have *block structure.* A name a: in a nested block B is in the scope of a declaration D of x in an enclosing block if there is no other declaration of x in an intervening block.

4 *Parameter Passing.* Parameters are passed from a calling procedure to the callee either by value or by reference. When large objects are passed by value, the values passed are really references to the objects themselves, resulting in an effective call-by-reference.

- *Aliasing.* When parameters are (effectively) passed by reference, two formal parameters can refer to the same object. This possibility allows a change in one variable to change another.