prog-theory Documentation

Takashi Hattori

Jan 22, 2024

TABLE OF CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 About this course

- Advanced subjects – for 3rd or 4th year students.
- Aim: Theoretical understanding of programming languages.
- Prerequisite: knowledge about two or more programming languages.

1.2 Evaluation

1.2.1 Allotment of points

- Weelky assignments: 20points
- Mid-term paper and presentation in class: 40points
- End-of-term exam: 40points

1.2.2 Distribution in the past years

- year 2021 — S:6, A:1, B:2, C:2, D:2
- year 2022 — S:13, A:16, B:5, C:3, D:13

1.3 A Fundamental Question

Why do we have so many programming languages?

- List of programming languages - Wikipedia
- New languages such as Go, Swift, and Rust has been developed recently.
- These languages are different from each other, but how different?

1.4 Definition of a language

When you want to define a new language, it is important that the definition is simple, rigorous and understandable.

- Scheme gives a mathematically precise definition which is hard to understand.
- BASIC and C had no rigid definitions at first. As a result, we had several slightly different language processors.

We need two things to define a language.

- syntax — how you write a program.
- semantics — what will happen when executiing a program

1.5 Differences in syntax

1.5.1 Character

- Unique symbols (cf. APL)
- Japanese characters (cf. Mind)
- Graphics instead of characters (cf. ToonTalk , Viscuit)

1.5.2 Grammar

There are languages called Esoteric progarmming language.

- Whitespace
- Befunge

1.6 Differences in semantics

Philosophically speaking, programming paradigm means “What is computation?”.

Imperative (Procedural)
: Computation is a sequence of changing state.

Functional
: Computation is a reduction of expression.

Logical
: Computation is a proof of proposition.

1.7 Criteria for good programming languages

- Readablility and writability are not compatible in general.
- Efficiency
  - fast execution
  - fast compilation
  - fast development process
- Reliability
  - Strong typing, forcing execption handling, etc. (However, some people don't like them)
- Other points
  - low learning cost
  - low development cost of language processors
  - abundant libraries

1.8 Reasons to study the theory of programming language

- You can select a language suitable for a task in your hand. — There is a saying 'If all you have is a hammer, everything looks like a nail'.
- You can learn a new language easily. — You must continue to learn since progress of IT is very fast.
- It is an important base of the computer science.

CHAPTER

TWO

SYNTAX AND SEMANTICS

2.1 Brief introduction to the formal language theory

2.1.1 Definition of formal languages

alphabet
: a set of characters

sentence
: a sequence of characters

grammar
: a set of rules to determine if a sentence is correct or not

language
: a set of correct sentences

We often use *tokens* instead of alphabets because of simplicity. A token is something like a word composed of several letters.

An example of tokens

Sentence `total = total + x;` can be decomposed into a sequence of characters `t, o, t, a, l, space, =, space, t, o, ...`, which is cumbersome. Instead, we usually use a sequence of tokens `total, =, total, +, x, ;`.

Generative Grammar defines a language by generating correct sentences in some manner.

terminal symbols
: Tokens used in languages

non-terminal symbols
: Symbols representing structures of sentences. One of them is designated as the initial symbol.

production rules
: Rules to rewrite some non-terminal symbols to other symbols.

Starting from the initial symbol, we can generate a correct sentence by repeatedly applying rules.

Simple grammar

- Terminal: I, you, love
- Non-terminal: S, N, V

- Initial: S
- Rule: S→NVN, N→I, N→you, V→love

There are several classes of grammars such as *context-free grammar* and *regular grammar*. Most of programming languages can be defined using context-free grammar.

2.1.2 BNF Notation

BNF (Backus-Naur Form) is a convenient notation to write a context-free grammar for practical use.

- Terminal symbols are written as they are.
- Non-terminal symbols are enclosed by `< >`.
- Production rules are in the form *non-terminal* `::=` *symbols* where we can write multiple alternatives by separating with `|` in the right side.

Simple arithmetic expression

Terminal: `A, B, C, +, *, (, )`

Non-terminal: `<var>, <term>, <expr>`

Initial: `<expr>`

Rule:

```
<var> ::= A | B | C
<term> ::= <var> | ( <expr> )
<expr> ::= <term> | <expr> + <term> | <expr> * <term>
```

2.1.3 Parse tree

To *parse* is to find how a sentence is generated by production rules. A *parse tree* is a tree structure that represents the result of parsing.

Generation of `A*(B+C)`

`<expr>` → `<expr>*<term>` → `<term>*<term>` → `<var>*<term>` → `A*<term>` → `A*(<expr>)` → `A*(<expr>+<term>)` → `A*(<term>+<term>)` → `A*(<var>+<term>)` → `A*(B+<term>)` → `A*(B+<var>)` → `A*(B+C)`

Exercise 1

Draw a parse tree of `A+B*C` with the grammar shown above.

[Answer]

Exercise 2

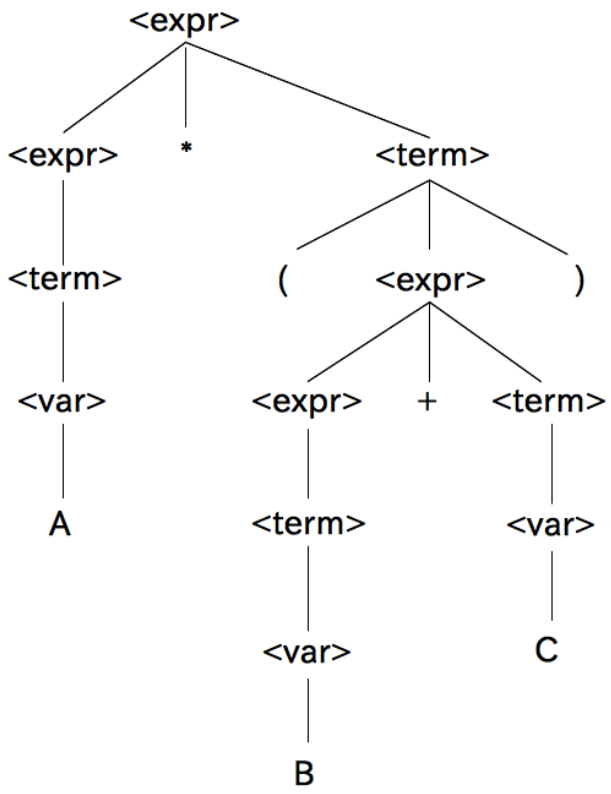


Fig. 1: Parse tree of `A*(B+C)`

In the grammar shown above, there is no precedence between + (add) and * (multiply). Change the grammar so that * (multiply) takes precedence over + (add). For example, when parsing `A+B*C`, make `B*C` be a subtree instead of `A+B`. Hint: introduce a new non-terminal symbol and a new rule so that + (add) and * (multiply) are parsed in separated levels.

[Answer]

2.1.4 Ambiguous grammar

We say a grammar is ambiguous if more than one parsing is possible for one sentence.

In a natural language

Structural ambiguity in English word-formation

An example of ambiguous grammar

We have two different parse trees for `A*B+C` with the following grammar.

```
<var> ::= A | B | C
<expr> ::= <var> | <expr> + <expr> | <expr> * <expr>
```

Dangling-else

Languages such as C and Java have the *dangling-else problem* since `else` is optional in `if` statements.

```
<if-statement> ::= if ( <expr> ) <statement> |
                   if ( <expr> ) <statement> else <statement>
```

Exercise 3

Explain the dangling-else problem in the following statement.

```
if ( x > 0) if ( y > 0 ) z = 1; else z = 2;
```

[answer]

Exercise 4

Explain that the following grammar is ambiguous.

```
<a> ::= <a> <a> | A
```

[answer]

2.2 Glance at the semantics theory

It is very difficult to define semantics of languages precisely. In C and Java, we usually say the meaning of ++ operator is "first evaluate the variable, then increment it". However, we cannot figure out the result of `x = 1; x = x++;` with this definition. As long as we use a natural language, ambiguity is inevitable.

Semantics of languages are largely depend on their paradigms. Functional languages have lambda calculus, and logical languages have first-order predicate logic as the theoretical foundation of their semantics.

We have no obvious theoretical foundation for imperative (procedural) languages. However, there are several formal semantics theories that can give rigorous definition of semantics of imperative languages.

2.2.1 Operational semantics

The operational semantics describes how a program is executed as a sequence of computational steps.

- A computational step is a change of state in the computer.
- The state in the computer is, for example, a mapping from variables to values (a model of the memory device).
- Using logical inference rules, we can define what kind of computational steps are possible.

2.2.2 Axiomatic semantics

The axiomatic semantics is based on Hoare logic. The meaning of a sentence is determined by a pair of precondition (which is true before the execution of the statement) and postcondition (which is true after the execution).

It is closely related to verification of a program. We can say a program is *correct* if it satisfies its specification. A specification is a pair of the initial condition and the final condition. Using Hoare Logic, we can prove a program satisfies its specification by chaining preconditions and postconditions of sentences.

2.2.3 Denotational semantics

The denotational semantics is based on recursive functions and domain theory.

It constructs mathematical objects that describe the meaning of a program. The interesting part is it includes an object that denotes an infinite loop.

Unfortunately, the constructed objects are often very hard to read (cf. Revised^5 Report on the Algorithmic Language Scheme).

CHAPTER

THREE

NAMES AND BINDINGS

3.1 Names in a program

We need *names* (or *identifiers*) in order to denote various entities in a program.

- Syntax of names differs depending on the language. In most languages, a name is a sequence of alphabets, numbers and some symbols.
- In some languages, there is a limit on the length of names.
- There are case-sensitive languages and case-insensitive languages.

A *Keyword* is a name which plays some designated role in syntax. A *reserved word* is a keyword which cannot be used with other meaning.

Fortran has no reserved word

The following Fortran program is syntactically correct.

```
INTEGER REAL
REAL INTEGER
REAL = 3
INTEGER = 1.2
```

Most languages provide standard libraries. Names defined in those libraries are neither keywords nor reserved words.

Names defined in libraries

In Javascript, `alert` function is provided by browsers. We can use the name `alert` for other purpose.

```
var alert;
alert = 1;
```

3.2 Namespace

When we use a lot of names, it is necessary to use long names in order to avoid conflicts. However, long names are not convenient to read and write. A *namespace* is a name given to a set of names, which allows us to make the hierarchy of names. In other words, it enables us to use concatenated short names instead of a single long name.

Namespace in C++

`::` is the operator to concatenate names.

```
#include <iostream>

main() {
  std::cout << "Hello, world!\n";
}
```

We can set the default namespace by `using namespace`.

```
#include <iostream>
using namespace std;

main() {
  cout << "Hello, world!\n";
}
```

Importing a module in Python

If you import a module, names defined in the module are available in the form *module.function*.

```
import foo
x = foo.bar()
```

If you import a function, the function name is available without the module name.

```
from foo import bar
x = bar()
```

3.3 Bindings

3.3.1 Definition of bindings

Because a name is a syntactic object, we must make a connection between a name and a semantic entity (e.g. variable, function, etc.).

- A *binding* is a link between a name (more precisely, an occurence of a name) and a certain semantic entity.
  - We call a binding *static* when it can be determined before execution (usually compile-time).
  - We call a binding *dynamic* when it is determined during execution.
- An *environment* is a set of bindings currently used during execution.

Note: A static binding does not necessarily mean a certain name always bound to the same entity. Different occurences of a certain name may have different bindings.

Variables in TinyBASIC

TinyBASIC (very small BASIC interpreter) has only 26 variables: `A` to `Z`. In this case, we always use the same environment.

Variables in Java

In Java, a variable declaration `int i;` means we will bind a name `i` to a newly created integer variable.

Exercise 5

There are two types of languages as to varibale declarations. Languages such as C and Java require declarations before variables are used, while languages such as Perl and Ruby do not. What are merits and demerits of these two types of languages?

[Answer]

3.3.2 Anonymous and alias

Bindings are not necessarily one-to-one association.

- We call an entity *anonymous* if it is not bound to any name.
- One entity can be bound to more than one names. In such case, names that bound to the same entity are called *alias* of each other.

Anonymous function in JavaScript

```
function(x) { return x+1; }
```

Anonymous function in Lisp

```
(lambda (x) (+ x 1))
```

Reference in PHP

- Reference Explained (PHP Manual)

Alias in Ruby

```
alias $b $a
$a = 2
puts $b
```

3.4 Scope

A *scope* is a part of program where a certain binding is used (in other words, we can see an entity through its name).

3.4.1 Static Scoping

In Algol-based languages, the scope of a name is a block in which the name is declared. If the same name is declared more than once, the most inner one takes precedence over outer ones. This type of scoping is also called lexical scoping because it is determined by the lexical stuructres.

Static scoping

Both `procedure main` and `procedure sub2` have variable definitions of `x`. Occurences of `x` in `procedure sub1` use the variable defined in `procedure main`.

```
procedure main;
  var x : integer;    { declaration 1 }

  procedure sub1;
    begin
      ... x ...       { refers declaration 1 }
    end;

  procedure sub2;
    var x : integer;  { declaration 2 }
    begin
      ... x ...       { refers declaration 2 }
    end;

  { body of main }
  begin
    ... x ...         { refers declaration 1 }
  end;
```

3.4.2 Dynamic scoping

In the original Lisp and some of its successor, if a name is declared in a function, its scope extends to other functions called from it. If the same name is declared more than once, the most recent one takes precedence over older one.

Dynamic scoping

Occurrences of `x` in `procedure sub1` have different bindings depending on which procedure calls `sub1`.

```
procedure main;
  var x : integer;  { declaration 1 }

  procedure sub1;
    begin
      ... x ...     { refers declaration 1 when called from B
                      refers declaration 2 when called from A }
    end;

  procedure sub2;
    var x : integer;  { declaration 2 }
    begin
      sub1          { call A }
    end;

  { body of main }
  begin
    sub1;           { call B }
    sub2
  end;
```

Exercise 6

In the following pseudo code, what is the output when using staic scoping and dynamic scoping, respectively?

```
procedure main;
  var x : integer;

  procedure sub1;
    begin
      writeln(x)  { output the value of x }
    end;

  procedure sub2;
    var x : integer;
    begin
      x := 10;
      sub1
    end;

  begin
    x := 5;
    sub2
```

(continues on next page)

(continued from previous page)

```
end.
```

[Answer]

CHAPTER

FOUR

VARIABLES

4.1 What is a variable?

A variable in mathematics represents a possibility of a value. If a range is designated to a variable, the variable may take one of values in the range. The selection of the value must be consistent throughout a proof.

A variable in functional and logical programming languages is similar to that in mathematics. A variable name is bound to a value. Once bound, it never changes. Note that the same name may be bound to a different value in the different environment.

A variable in imperative (procedural) programming languages is a named box to store a value. A variable name is bound to a certain memory area. The value in the memory area may change during execution of the program.

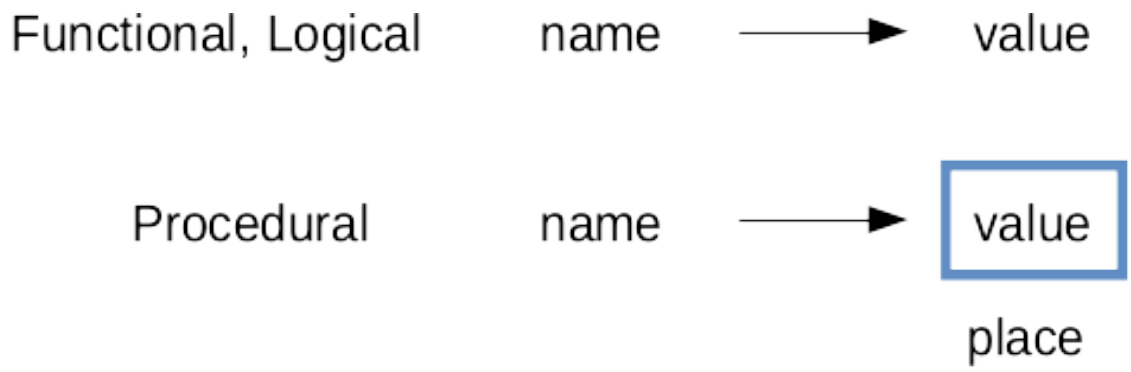


Fig. 1: Difference of variable binding

In the remaining part of this section, we mainly deal with variables in imperative (procedural) languages.

4.2 Type

Many languages such as Fortran, Pascal, C and Java have a *type* as a property of a variable. In these languages, a type must be declared for a variable before its use, and the variable can only store values of that type.

On the other hand, languages such as Lisp, JavaScript, Python, PHP and Ruby have no types for variables. A variable can store any value.

We will learn about types in detail next week.

4.3 Left value

A memory area is the essential part of a variable. A value is stored there during program execution.

If a variable name occurs in a program, it is evaluated as follows:

1. Get an address of a memory area which is bound to the name.
2. Get a value from the memory area.

If a variable name occurs in the left side of an assignment statement, however, the result of evaluation should be an address to store a new value. Sometimes we call an address as a *left value*.

4.4 Memory management

Compilers (or interpreters) allocate some memory area for each of the variables. There are several memory management methods as described below.

4.4.1 Static data area

A static variable is a variable that exists across the entire execution of the program. It is usually allocated in the *static data area* at compile time.

One of the merits of type declaration is that the required amount of memory can be calculated at compile time based on the type declared. If we have no type declaration, the memory area for a variable usually contains a pointer that refers to a heap area (see below) where we can allocate required amount of memory at run time.

4.4.2 Stack

A *stack* is a first-in-last-out data structure. It is often used for the area to allocate local variables. When a fuction is called, parameters and local variables are allocated at the top of the stack. When that function returns, the allocated area is discarded.

- Recursive call can be easily implemented.
- When using dynamic scoping, we can find the valid bindings by searching downwards from the top of the stack. It is easier to implement than static scoping.
- A variable which occurs in a closure should not be allocated in the stack because the variable will be used when the closure is called afterwards (see extent).

4.4.3 Heap

A *heap* is a data structure in which we can allocate some amount of memory when required and release it when not necessary any more. The released area will be reused for other purposes.

Some languages require the programmers to write a code for releasing unnecessary area. They often (very often) forget to write the code, resulting a situation where we have no new area to allocate even if the actually used area is small. We call this situation as a *memory leak*.

In order to avoid the memory leak, some languages provide *garbage collection* which automatically releases unnecessary area. We can determine a certain area is unnecessary if there is no refrence to the area.

Followings are the typical garbage collection methods:

Reference count
: A counter is attached to each memory area. When someone starts to refer the area, increment the counter. When stopping to refer, decrement it. If the counter becomes 0, release that area.

Mark and sweep
: First, we put a mark on every area reachable by following references from areas obviously necessary (e.g. an area bound to a variable name). Next, we sweep the entire heap from end to end, and release areas that are not marked.

Copying
: We divide the heap into two, and use only half. When there is no available area to allocate, we copy the necessary area to the other half by following references starting from obviously necessary areas. This method achieves *compaction* as well as garbage collection.

Garbage collection usually requires to stop the execution of the program during it takes place. Since it is inconvenient for real-time systems, there are devised garbage collection methods that can be done in parallel with the execution of the program.

Exercise 7

Suppose we use reference count garbage collection. In some cases, the counter of unnecessary area will not become 0. Give an example.

[Answer]

Exercise 8

Explain why we should stop the execution of the program when using mark-and-sweep garbage collection.

[Answer]

4.5 Extent

An *extent* is a time period during which a variable holds its value. Note that a scope and an extent are different in general.

In most languages, variables have the following extent:

global variables
: during the entire program execution.

local variables
: during the function (or subprogram unit) execution.

instance variables
: during the instance exists.

Static local variables

In languages such as C and PHP, a local variable declared as `static` has a scope of that function, but its extent is the entire execution of the program.

```
function test()
{
```

(continues on next page)

(continued from previous page)

```
    static $a = 0;
    echo $a;
    $a++;
}
```

Exercise 9

Give an example where a static local variable is effectively used.

[Answer]

Note: When a function is excuted as a closure (we will study it later), the extent of local variables declared in it will be extended. In addition, they must be allocated on the heap instead of the stack.

CHAPTER

FIVE

TYPES

5.1 What is a type?

A *type* consists of a set of values and designated operations on the values.

Some languages provide sophisticated types while others do not. Major reasons why we use types are as follows:

- More informative for human readers.
- Compilers can detect errors when types mismatch.
- Compilers can generate more efficient code.

Exercise 10

An integer is used for conditional branch (0 is false, otherwise true) in C, while `boolean` and `int` are different types in Java. What are merits and demerits of these two methods.

[Answer]

5.2 Type declaration

Dynamic typing
: Syntactic entities have no type declarations whereas semantic entities (data) have types. Types of expressions are determined when their values are calculated. Any data can be assigned to any variable.

Static typing
: Types of expressions are determined before execution, based on type declarations.

- In Fortran, explicit and implicit variable declarations are possible. Using implicit declaration, a variable is typed according to the initial letter of its name: integer if beginning with I, ... , N, and real if beginning with other letters.
- In Algol-like languages, variables, function parameters and return values need type declarations.
- In ML and Haskell, a type can be declared for any expression in a program.

5.3 Various types

Primitive types
: Hardware-defined or indecomposable types. (e.g. integer, floating point, character)

Structured types
: Types composed of some other types. Many languages allow programmers to define new types.

5.3.1 Enumerated type

We can define a new enumerated type by writing all the possible values for that type.

- Usually implemented by integer.
- only comparison is allowed.
- More readable. Easier to detect errors.

5.3.2 Subrange type

A subrange type is an extraction of a certain ordinal type by specifying its highest and lowest values.

- Same operations are allowed as the original type.
- We can detect errors when values are out of range. (For efficiency, some languages provide choices to detect errors or not.)

An enumerate type and a subrange type in Pascal

```
type
  suit = (club, diamond, heart, spade);
  card = record
           s: suit;
           v: 1..13;
         end;

var myhand: array[1..5] of card;
begin
  myhand[1].s := club;
  myhand[1].v := 1;
  myhand[2].s := club + heart;   { Error }
  myhand[2].v := 0; { Error }
end
```

5.3.3 Union type

A union type is a type whose value may be one of several different types.

- In C, `union` allows programmers to access the same memory location with more than one types.
- It is up to a programmer which type is used to read and write the content of the memory.
- Some programmers intentionally use it for evil purposes.
- In Pascal, `record` with variant parts provides more sophisticated way.
- A tag field indicates the type currently stored in a variant part.
- An error occurs when an operation mismatches with the type in the tag.
- Still, some programmers may intentionally change the tag value to avoid type checking.

A record with variant parts in Pacal

```
{ `shape' is an enumerated type whose value is `circle' or `rectangle' }
type shape = (circle, rectangle);

{ `figure' is a record type to store data of rectangles and circles }
type figure = record
                x : real, y : real;
                case form : shape of      { variant parts whose tag is `form' }
                  circle:    (radius : real);
                  rectangle: (width: real; height: real);
                end;

var myfigure : figure;
begin
  myfigure.x := 2.0;
  myfigure.y := 3.0;
  myfigure.form := circle;
  myfigure.radius := 1.0;
  myfigure.width := 1.0;  {Error}
end
```

- In object oriented languages, class inheritance is used for that puropose.

Class inheritance in Java

```
class Main {
  public static void main(String[] args) {
     Figure myfigure;
     myfigure = new Circle();
     myfigure.x = 2.0;
     myfigure.y = 3.0;
     ((Circle)myfigure).radius = 1.0;
  }
}

class Figure {
```

(continues on next page)

(continued from previous page)

```
  double x, y;
}

class Circle extends Figure {
  double radius;
}

class Rectangle extends Figure {
  double width, height;
}
```

- In TypeScript, the union operator combines any two types.
  - An operation is allowed if it is valid for all of original types.

Union class and narrowing in TypeScript

```
"use strict"

interface circle { kind: "circle", x:number, y:number, radius:number }
interface rectangle { kind: "rectangle", x:number, y:number, width: number, height:␣
↪number }

type figure = circle | rectangle

function foo(f: figure) {
    console.log(f.x)       // OK
    console.log(f.radius)  // Error - Property 'radius' does not exist on type 'figure'.
}

function bar(f: figure) {
    if (f.kind == "circle") {   // Narrowing - Type of f is now circle
        console.log(f.radius)   // OK
    }
}
```

5.3.4 Pointer type

A pointer type is a type whose value is a reference to another variable. By applying dereference operation, we can get a value of the referred variable.

By using pointers, we can dynamically allocate variables that are not bound to any names. It may cause the following two problems:

Dangling Pointer
: If a memory area is released during a pointer is referring to it, the pointer will refer to a garbage or a reused area.

Lost Heap-dynamic Variable
: If values of pointers are changed, the memory area referred by the pointers may lose all the references, which means we can never access that area.

- In case of Pascal, a pointer can only refer to an area allocated by `new`. We must explicitly release the allocated area by `dispose`. Pascal has the above two problems.
- In case of C, pointers are virtually same as the hardware memory addresses. C has many problems including the above two.
  - For example, some programmers relied on the fact that address 0 always contains value 0 in VAX machines. As a result, there were plenty of C programs that run only on VAX machines.
- In case of Java, objects are implicitly referred to by pointers. Automatic garbage collection prevents the above two problems.

5.3.5 Function type

A function type is composed of types of parameters and return values. It will be very complex for higher-order functions (i.e. parameters or return values are functions).

Function type in Haskell

```
inc :: Int -> Int
inc x = x + 1

add :: Int -> Int -> Int
add x y = x + y

twice :: (Int -> Int) -> Int
twice f = f(f(1))

twice(inc)   -- returns 3
```

5.4 Type systems

A *type system* is a set of rules that associates syntactic entities (usually expressions) with their types.

The process of verifying consistency of types in a program is called *type checking*. Type checking at compile time is very useful for debugging.

5.5 Type inference

In languages such as ML and Haskell, every expression in a program should have its type. It is too painful to write types for all expressions, the compiler makes an inference about proper typing if we give minimum declarations.

Type inference in Standard ML

We need no type declaration for the function `inc`.

```
fun inc x = x + 1;
  val inc = fn : int -> int
```

We need at least one type declaration for the function `add`.

```
fun add (x, y) = x + y;
  Error: overloaded variable '+' cannot be resolved

fun add (x, y) = x + (y : int);
  val add = fn : int * int -> int

fun add (x, y) : int = x + y;
  val add = fn : int * int -> int
```

Note: In SML ver0.5 or after, the + operator is an overloaded function that has default type `int`, which means `fun add(x, y) = x + y;` has the type `fn: int * int -> int`.

5.6 Polymorphism

When a single operator or function can deal with values of more than one type, we call it polymorphism.

ad hoc polymorphism
: Definitions for each type are independent. Also called overloading.

parametric polymorphism
: Definitions include formal parameters that receive types of data currently processed.

+ operator in Java

In Java, + operator has two meanings: addition of numbers and concatenation of strings.

Template in C++

We can write a function `max` in C++ as follows:

```
template <class Type>
Type max(Type first, Type second) {
    return first > second ? first : second;
}
```

This template can be instantiated for `int` and `char` as follows:

```
int a, b, c;
char d, e, f;

c = max(a, b);
f = max(d, e);
```

Polymorphic function in Haskell

In Haskell, an array type is denoted by a bracket `[ ]` and its element type. For example, the type of `[1, 2, 3]` is `[Int]`, and the type of `["abc", "def"]` is `[[Char]]` (because a string is an array of character `[Char]`).

Although `[1, 2, 3]` and `["abc", "def"]` have different types, the function `length` can be applied to both of them. For example, `length [1, 2, 3]` returns 3 and `length ["abc", "def"]` returns 2.

The type of `length` is `[a]->Int` where `a` is a type variable.

Exercise 11

In Haskell, a function call `map f x` applies the function `f` to each element of the array `x`, and then make an array collecting their results. For example, `map abs [-5, 3]` returns `[5, 3]` and `map length ["abc", "de"]` returns `[3, 2]`. Write the type of `map`, using type variables `a` and `b`.

[Answer]

CHAPTER

SIX

EXPRESSIONS

6.1 Operator Precedence Grammar

When given an expression, we can investigate its structure by making its parse tree. We have learned BNF(Section 2.1.2) to write a grammar, but we prefer *operator precedence grammar* for expressions because it is much simpler.

- Each operator has its precedence. The higher the precedence of a certain operator is, the smaller sub-tree it makes (in other words, it connects its arguments earlier than other operators).
- Each binary operator is either left-associative or right-associative. When there are more than one operator of the same precedence in an expression, if they are left-associative, the leftmost operator connects its arguments first. If right-associative, the rightmost connects first.

Note: Instead of drawing parse trees, we usually use parentheses to indicate which part is connected earlier than other parts. Preceisely speaking, parse trees of expressions with extra parentheses are different from original ones.

Abstract syntax tree (AST) is a tree that can be made from a parse tree by removing non-essential nodes (e.g. parentheses).

Abstract syntax tree - Wikipedia

In the following explanation, we add parentheses to indicate precedences among operators, keeping the AST unchanged.

Precedences of arithmetic operators

In most languages, arithmetic operators have the same precedences as mathematics. For example, `a + b * c` is equivalent to `a + ( b * c )`.

Associativities of arithemtic operators

In most languages, arithmetic operators are left-associative. For example, `a - b + c - d` is equivalent to `(( a - b ) + c ) - d`.

Precedence and associativity in APL

In APL, all operatos have the same precedence, and are right-associative. For example, `A × B + C` is equivalent to `A × ( B + C )`.

Exercise 13

Suppose precedences and associativities of operators are given by the following table. Put parentheses at proper positions in expressions below so that their structures are explicit.

precedences	associativities	
high	left-associative	#, $
low	right-associative	!, ?

1. `a # b $ c`
2. `a ! b ? c`
3. `a # b ! c $ d`
4. `a # b ! c ? d`
5. `a # b $ c ! d ? e`

[Answer]

6.2 Order of evaluation

Evaluation is the process to calculate values of expressions.

Note that the order of evaluation is a different concept from precedence and associativity because evaluation is a semantic concept while precedence and associativity is a syntactic concept(Section 1.4).

When an expression is composed of several sub-expressions, the result depends on the order of evaluation if some of sub-expressions have side-effects.

Side effect and evaluation order

In the following C program, the result is different depending on that either variable `a` or function `fun1()` in the statement (1) is evaluated first.

```
int a = 1;

int fun1() {
  a = 2;
  return 3;
}

void fun2() {
  a = a + fun1();  /* (1) */
}

void main() {
  fun2();
}
```

As shown above, side-effect causes a problem if the order of evaluation is undefined. We may think of several solutions as follows:

1. Do nothing. — The result is compiler dependent.

2. Prohibit side effect. — It is not practical for imperative languages.
3. Define the order of evaluation. — It obstructs optimization.

- Old C language adopts (1).
- Java adopts (3). It evaluates an expression from left to right (See What are the rules for evaluation order in Java? for detail).
- C/C++ introduces a sequence point, which is a mixture of (1) and (3).
- Functional lanuages such as ML and Haskell adopts (2) with no problem because they have no side-effects.

6.3 Short-circuit evaluation

Short-circuit evaluation is an evaluation strategy where some sub-expressions may not be evaluated if the value of the entire expression can be determined without them.

Logical operators

When the value of `a` is less than 0, the following expression can be evaluated to `false` without evaluating `b < 10`.

```
( a >= 0 ) && ( b < 10 )
```

Exercise 14

Java has two types of logical operators: `&&` and `||` are short-circuting and `&` and `|` are not. Explain why we need both types of operators.

[Answer]

6.4 Lazy evaluation

Lazy evaluation is an evaluation strategy where an expression is not evaluated until its value is necessary in another part later.

Lazy evaluation implies short-circuit evaluation because sub-expressions are not required to be evaluated after the value of entire expression is calculated.

`range` in Python3

```
for i in range(10000):
   if i > 10: break
   print(i*2)
```

In Python2, `range(10000)` generates a list of 10000 elements and returns them. In Python3, `range(10000)` returns a list whose elements have not been evaluated yet. The elements will be generated one by one when each time `print(i*2)` is executed.

Enumerable#lazy in Ruby

```
x = (1..Float::INFINITY).lazy.map{|x| x*2}
puts x.first(10)
```

Haskell

Haskell uses lazy evaluation for every expression.

CHAPTER

SEVEN

CONTROL STRUCTURES

7.1 Structured programming

Languages in early days such as Fortran and COBOL use `goto` statement extensively, which results in obscure program structure. In 1968, Dijkstra insisted a program structure is more readable if it is a hierarchical composition of sequences, iterations and conditional branches.

Loop in early Basic and Pascal

```
10 X=1
20 IF X>=1000 THEN GOTO 60
30 PRINT X
40 X=X*2
50 GOTO 20
60 END
```

```
x = 1;
while x<1000 do
  begin
    writeln(x);
    x = x*2
  end;
```

7.2 Non-local exits

According to the idea of structured programming, it is not a good code to jump over block structures, but we sometimes want to write such code. For example, a `return` statement immediately exits from the function block even if it is in nested blocks.

Non-local exits is an enhanced version of `return` (and tamed version of `goto`) that terminates nested function calls at once and goes back to the outer level function.

Catch and throw in Ruby

```
def bar
  throw :a, 3
end
```

(continues on next page)

(continued from previous page)

```
def foo
  bar
end

catch(:a) do
  foo
end
```

7.3 Exceptions

7.3.1 What is exception?

In early languages, if something unusual occurs during execution, the program has no choice but to stop with an error message. Recent languages can deal with such situation more gracefully.

An *exception* is an unusual event in a program for which we need some extra processing.

- Detected by hardware: division by zero, to read or write protected memory, end of file, etc.
- Detected by software: index of an array is out of range, etc.
- Defined by programmers: anything you wish.

7.3.2 Exception handling

Even if a language has no exception handling mechanism, it is possible to write a program that deal with unusual events. However, exception handling is more convenient in the following way:

- No need to write codes for detecting exceptions. Imagine if you need to check that the divisor is not equal to 0 before every division.
- Cleaner code for dealing with an unusual event, especially when it requires non-local exits.
- Managing several exceptions in one place.

An *exception handler* is a part of program that is invoked when an exception occurs.

Exception handling in PL/I

An exception handler can be written in anywhere in the program with the `ON` statement. For example, the following is a handler for an exception `SUBSCRIPTRANGE`, which occurs when an index of an array is out of range.

```
ON SUBSCRIPTRANGE
  BEGIN;
    PUT LIST ('ERROR') SKIP;
  END;
```

Note that the `ON` statement is an executable statement (not a declaration), which means bindings between exceptions and exception handlers are dynamic. An exception handler is valid after an `ON` statement is executed, and until another `ON` statement with the same exception name is executed.

In addition, in order to make an execption handler valid, we need to write an exception name before a part of the program in which the exception may occur. For example, the following code checks the index `I` if it is out of range or not.

```
(SUBSCRIPTRANGE):
  BEGIN;
    A = B(I);
  END;
```

Exception handling in Java

- An exception is represented by an instance of an exception class, which is passed to a handler as an argument.
- Exception classes are hierarchical. An execption handler can deal with many kinds of exceptions by specifying a high level exception class in the hierarchy.
- Bindings between exceptions and exception handlers are lexical. `try-catch-finally` statement makes the scope of handlers.

```
try {
    FileWriter x = new FileWriter("sample.txt");  // IOException may occur
} catch (IOException e) {
    e.printStackTrace();
}
```

- By declaring `throws` at the head of a method, an exception can be propagated to the calling method, which allows dynamic bindings between exceptions and exception handlers.

```
static void bar() {
    try {
        foo();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

static void foo() throws IOException {
    FileWriter x = new FileWriter("sample.txt");
}
```

- If there is a possibility of an exception, programmers must either write an exception handlers or propagate it. The aim of this rule is to make programs robust in unexpected situations.

Excercise 15

Java forces programmers to handle exceptions, but that is not the case for `Error` and `RunTimeException`. Guess the reason.

[Answer]

CHAPTER

EIGHT

SUBPROGRAMS

8.1 What is a subprogram?

A *subprogram* is a chunk of code that is independent from other parts to some extent. In most languages, subprograms have the following properties.

- There is a single entry point. (But Fortran provides multiple entry points.)
- When a subprogram *A* calls another subprogram *B*, *A* will pause until *B* terminates. Namely, there is only one subprogram that is executed at a time. (This is not the case for concurrent languages.)

In some languages, a subprogram that returns a value is called a *function* and one that does not is called a *procedure* or a *subroutine*, but both types of subprograms are called *functions* in other languages such as C. In addition, object-oriented languages usually use the term *methods* for them.

8.2 Parameter

A *parameter* is used in order to apply a single subprogram to various data.

Actual parameters ::
Data specified when a subprogram is called.

Formal parameters ::
Variable names used in a subprogram to store actual parameters.

- In many languages, acutal parameters are associated to formal parameters by their positions (*positional parameters*). It is error-prone when a subprogram has many parameters. In some languages, it is possible to specify a formal parameter for each actual parameter when calling (*keyword parameters*).
- In many lanuages, the number of actual parameters and formal parameters must be the same. In some languages, it is possible to declare default values for formal parameters so that we can omit some actual parameters.
- In script lanuages such as Ruby, more than one actual parameters can be combined into an array to be stored in a single formal parameter, and vice versa.

Parameters in Ada

Suppose we have the following declaration in Ada.

```
procedure Draw_Rect( X , Y : Integer;
                     Width : Integer := 10;
                     Height : Integer := 10;
                     Color : Integer := 0 )
```

We may write any of the following procedure calls.

```
Draw_Rect( 100, 50, 20, 10, 1 )
Draw_Rect( Width => 20, X => 100, Y => 50, Color => 1 )
Draw_Rect( 100, 50, Width => 20, Color => 1 )
```

Parameters in Ruby

When a formal parameter has an asterisk, any number of actual parameters are combined into an array.

```
def foo(*a)
  puts a  # output [1, 2, 3]
end

foo(1, 2, 3)
```

When an actual parameter is an array and formal parameters are enclosed by a parenthes, elements of the array will be stored in the formal parameters one by one. Surplus elements are ignored.

```
def bar((a,b))
  puts a  # output 1
  puts b  # output 2
end

bar([1,2,3])
```

8.3 Direction of parameter passing

Input
: Data are passed from actual parameters to formal parameters when a subprogram is called.

Output
: Data are passed from formal parameters to actual parameters when a subprograum finishes its execution.

In case of output, the actual parameter must be assignable, i.e. an expression that has a left value (Section 4.3), which is typically a variable.

Passing parameters in Ada

```
procedure Foo(A : in Integer;
              B : in out Integer;
              C : out Integer) is
begin
  B := B + 1;
  C := A + B;
end Foo;
```

8.4 Evaluation strategy of parameters

8.4.1 Pass by value (call by value)

First, all the actual parameters are evaluated. Then, the values are copied to formal parameters.

- It is usually used for input direction. In case it is used for both input and output, we specifically call it *pass-by-copy-restore*.

8.4.2 Pass by reference (call by reference)

Formal parameters are bound to entities referred by associated actual parameters. In other words, formal parameters become aliases (Section 3.3.2) of actual parameters.

- Its direction is inherently both input and output.

Passing parameters in Pascal

In Pascal, a foraml parameter declared with `var` is pass-by-reference, otherwise pass-by-value (input only).

For example, the following procedure does nothing at all.

```
procedure swap(a, b : integer)
  temp : integer;
  begin
    temp := a;
    a := b;
    b := temp
  end;
```

If we change the first line as follows, it will swap values of two parameters.

```
procedure swap(var a, b : integer)
```

8.4.3 Pass by name (call by name)

Actual parameters are not evaluated when a subprogram is called. They are evaluated each time associated formal parameters are used in the body of subprograms. The environment used for evaluation is the one at the place where the subprogram is called.

- In Algol, an assignment to a formal parameter implies an assignment to the associated actual parameter (provided it is assignable), which means the direction is input and output. In Scala, an assignment to a formal parameter is prohibited, so the direction is input only.
- Precisely speaking, there are two methods to implement pass-by-name.
  - Actual parameters are evaluated each time formal parameters are used.
  - Actual parameters are evaluated at the first time formal parameters are used, and the values are memoized. After that, the memoized values are used, and actual parameters are not evaluated. This method is also called *pass-by-need* to distinguish it from the previous method.

  Results of the two methods are the same as long as there is no side-effect, but they may be different if actual parameters or subprograms have side-effect.

Passing parameters in Scala

```
var x = 0
def f(a: => Int) = {
    print(a)  // output 1
    x = 2
    print(a)  // output 3
}
f(x+1)
```

Exercise 16

Suppose we have the following C-like program.

```
void main() {
  int x = 2, list[5] = {1, 3, 5, 7, 9};
  swap(x, list[0]);
  swap(list[0], list[1]);
  swap(x, list[x]);
}

void swap(int a, int b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Answer the values of variables `x` and `list` after execution when we use the following evaluation strategy of parameters, respectively.

1. pass-by-value (input only)
2. pass-by-reference
3. pass-by-name (input and output, evaluate each time)

NOTE: When using pass-by-name, assume that an assignment to a formal parameter implies an assignment to the associated actual parameter, thus the direction is input and output.

[Answer]

8.5 Closure

In languages such as Lisp, Ruby, JavaScript, etc., a function is a first-class object (i.e. it can be assigned into a variable, passed as a parameter, etc.).

Suppose we put a function into a variable and call it later.

In case the language has static scope (Section 3.4), variables in the function should be evaluated in the environment at the place the function is defined (not the place the function is called).

In order to do that, we need to prolong extent (Section 4.5) of variables of outer blocks so that the stored function can use those variables. This feature is called a *closure*.

closure in JavaScript

```
function get_increment_function() {
    var x = 1;
    return function() { return x += 1; };
}

var inc1 = get_increment_function();
console.log(inc1());  // 2
console.log(inc1());  // 3

var inc2 = get_increment_function();
console.log(inc2());  // 2

console.log(inc1());  // 4
```

Exercise 17

Suppose we want to create 10 buttons in a web page, each of which will show its id number when clicked. But the following JavaScript code does not work properly. Explain why.

```
var b = document.getElementsByTagName("input");
        // Returns an array of 10 button objects.
let i;
for (i = 1; i <= 10; i++) {
    b[i-1].onclick = function(){ alert(i); };
}
```

[Answer]

CHAPTER

NINE

OBJECT ORIENTED PROGRAMMING

9.1 What is "Object Oriented"?

9.1.1 "Object oriented" as a paradigm

We usually make a model when solving a real-world problem. The model is called *object oriented* if the model consists of something called *objects* and these objects interact with each other.

- By using objects, we can divide a task into several encapsulated sub-tasks.
- The object oriented paradigm is not only used in programming but also applicable to the whole software development process such as the object oriented design.

9.1.2 Object oriented languages

Many people use the term *object oriented languages* in various different ways. One of the broadest definitions is the following two conditions:

- There is a unit called *object* that performs some calculation and stores some data.
- A *message* is passed from one object to another.

This definition requires neither class nor dynamic creation of objects.

Other (narrower) definitions include more conditions, some of which are:

- Objects are created dynamically.
- Objects are abstract data types.
- Objects can inherit functions of other objects.

9.2 Abstract data type

An abstarct data type is a type that has following properties:

- From outside, we cannot say how data are actually represented and stored.
- The only available operations from outside are subprograms given by the abstract data type.

The two properties above provide abstarction of data structures; modificatin inside an abstract data type does not affect outside. This is also called *encapsulation* or *information hiding*.

Package in Ada

At the time Ada was designed, there was no such thing as Internet. It was very important to support development of a large program by dividing it to several modules which were developed in different places.

- *package* is the unit of encapsulation.
- A package consists of a *specification package* and a *body package* of the same name.
- A specification package contains interface information. When compiling, only specification package is required to use that package.

Though a specification package is basically a public information, a private part can be declared in a specification package so that type information is available for only compilers, not programmers.

```
package Stack_Pack is
  -- public part
  type Stack_Type is limited private;
  function Empty(S : in Stack_Type) return Boolean;
  procedure Push(S : in out Stack_Type; Element : in Integer);
  procedure Pop(S : in out Stack_Type);
  function Top(S : in Stack_Type) return Integer;
  -- private part
  private
    Max_Size : constant := 100;
    type Stack_Type is
      record
      List : array ( 1 .. Max_Size ) of Integer;
      Top_Index : Integer range 0 .. Max_Size := 0;
      end record;
  end Stack_Pack
```

9.3 Designing object oriented languages

9.3.1 How pure the language is object oriented?

It is possible to represent every data as an object and every operation as a message. However, it may be inefficient to implement operations such as 1+2 with objects and messages.

- In C++ and Java, primitive data are not objects.
- In Ruby, all the data are objects, but control structures such as `while` and `if` are not messages.
- In Smalltalk, everything is an object, and every operation (except assignment and return) is a message.

New methods for integers in Ruby

```
class Integer
  def foo
    self + 100
  end
end

1.foo  # => 101
```

Conditional branch in Smalltalk

```
"Push an element into the stack"
push: anElement
  self isFull ifTrue: [ self error: 'stack full' ]
              ifFalse: [ top <- top + 1.
                         elements at: top put: anElement ]
```

metaclass

Since everything is an object in Ruby and Smalltalk, a class is also an object. For example, sending a message *new* to a class creates a new instance.

Since every object has a class that defines the behavior of the object, a class has also a class called *metaclass* that defines the behavior of the class.

Since everything is an object, a metaclass is an object.

Since every object has a class that defines the behavior of the object, a metaclass has also a class that defines the behavior of the metaclass.

Since everything is an object, a class of metaclass is an object.

Since every object has ...

9.3.2 Class-based or prototype-based?

class-based

- There is a class that defines attributes or behavior of objects.
- It is possible to create more than one instances from a class where all instances share the definition of the class.
- *Inheritance* is a mechanism to share attributes and behavior among more than one classes.
- Many languages such as Smalltalk, C++, Java, Ruby, etc.

prototype-based

- Each object has a reference to another object called *prototype*.
- When a object is requested some attributes the object does not posses, the object asks for its prototype. If the prototype does not posess those attributes either, it further asks for its prototype. This mechanism is called *delegation*.
- Languages such as Self, JavaScript.

9.3.3 What kind of information hiding is possible?

It is usually a good idea to encapsulate data stored in an object. However, it is sometimes convenient to allow access from outside.

Controlling Access in Smalltalk

- No Instance variables can be accessed by other objects.
- All methods are accessible from any other object.

Controlling Access in C++

- `public` : accessible from any class.
- `protected` : accessible from subclasses and friends.
- `private` : accessible from friends only.

Controlling Access in Java

Package is another unit of information hiding.

- `public` : accessible from any class.
- No modifiers : accessible from classes in the same package.
- `protected` : accessible from subclasses.
- `private` : not accessible from any other class.

9.3.4 Single inheritance or multiple inheritance?

single inheritance

Each class has only one superclass.

multiple inheritance

A class may have more than one superclasses.

Obviously, multiple inheritance is more powerful than single. However, some people think multiple inheritance is hard to use because its semantics is very complicated.

9.3.5 Specification inheritance or implementation inheritance?

Specification inheritance

Only a public part of superclass is accessible from its subclasses.

Implementation inheritance

All parts of superclass (including internal structure) is accessible from its subclasses.

Implementation inheritance allows programmers to reuse code in a superclass. However, it is not good for encapsulation because modificaton of superclass may affect subclasses.

Inheritance in Java

- `extends` : single implementation inheritance
- `implements` : multiple specification inheritance

9.3.6 How types are checked?

In typed object oriented languages, a class is usually a type. Since a subclass represents a subset of its superclass, it is natural to define a subclass as a subtype of its superclass (i.e. the subclass type can be substituted for the superclass type).

Type checking in Java

```
public class A {
    public static void main(String args[]) {
        A x = new B();    //OK
        B y = new A();    //Error at compile time
        B y = (B)new A(); //Error at runtime
    }
}

class B extends A {
}
```

Exercise 18

In the example above, `B y = new A();` is prohibited while `A x = new B();` is permitted.

Explain what kind of inconvenience would occur if `B y = new A();` was permitted.

[Answer]

9.3.7 Early binding or late binding?

When a method in a superclass is overridden by a method in a subclass, we need to decide which method definition will be used.

- Early binding (or static binding): methods to be called are fixed before execution by using type information. Faster execution.
- Late binding (or dynamic binding): methods to be called are determined at runtime. More flexible.

Virtual functions in C++

- Default is early binding.
- A function with `virtual` uses late binding.

```
class Bird {
public:
  bool fly() { return true; }
};

class Penguin : public Bird {
public:
  bool fly() { return false; }
};

int main() {
  Bird *x;
  Penguin *y;

  y = new Penguin();
  x = y;
  x->fly() // true
  y->fly() // false
}
```

In case of Java

- Default is late binding.
- A method with `final` uses early binding because it cannot be overridden by subclasses.

CHAPTER

TEN

FUNCTIONAL PROGRAMMING

10.1 What is functional programming language?

Common features of functional programming languages are as follows:

- A program is a set of functions.
- A function is a first class object.
- There is no such thing as statements or commands. Evaluating expressions is the only way to execute a program.
- There is referential transparency.

10.2 referential transparency

A language has *referential transparency* if literally same expressions always result in the same value as long as the environment is the same.

10.2.1 Variable

Since a single variable is an expression, referential transparency implies that the value of the variable is always the same.

In other words, a variable name is directly bound to a value (cf. *Bindings*). Note that the same name may be bound to another value in another environment.

You may think switching environments is virtually same as assigning new values. However, there are differences as follows:

- Environments are usually associated with syntactic structures while assignments are non-structured and hard to predict their influence.
- Since creation of a new environment does not affect existing environments, there is no side-effect.

10.2.2 Loop

With referential transparency, it is impossible to have a loop structure such as `while` because the value of its condition never changes.

Instead, recursive functions are usually used for iterations in functional languages. Each time a function is called, a new environment is created so that a condition may have another value.

10.2.3 Data structure

When we have a complex data, referential transparency implies that its structure never changes. It is also called *persistent data structure* or *immutable data structure*.

Although it is called persistent, it may be removed by the garbage collector.

Note: Whereas the most prominent feature of functional languages is referential transparency, not so many languages strictly follow the policy. For example, Lisp has `setq` (assignment to variables) which breaks referential transparency. However, many people say Lisp is a functional language because it is possible to write a program with only features that keep referential transparency.

When we want to emphasize that every feature of a language keeps referential transparency, we say the language is pure functional.

10.3 Theoretical basis

Lambda calculus is a formal system used for research of theoretical computer science. See Wikipedia for detail.

10.3.1 Reduction

In mathematics, the right side of $1 + 2 =$ may be $4 - 1$ or $\sqrt{9}$. However, we expect the value of `1+2` is `3` in computer programs. Reduction is, intuitively speaking, a procedure to convert an expression into a simpler equivalent expression.

- β-reduction in lambda calculus corresponds to applying a function to its arguments in programming languages.
- Given a lambda expression, we can repeatedly apply β-reduction until it is irreducible any more. The final expression is called the *normal form* of the given expression.
- Normal form may not exist because β-reduction may be applicable infinitely many times.

10.3.2 Reduction strategy

A lambda expression may have more than one part for which β-reduction is applicable. A *reduction strategy* is a rule to choose the next reduction step.

applicative order

> Select the innermost part for which β-reduction is applicable. There is a possibility that reduction continues forever even if the expression has a normal form.

normal order

> Select the outermost part for which β-reduction is applicable. Reduction always reach the normal form if it exists.

Reduction strategies correspond to evaluation strategies in programming languages (cf. *Evaluation strategy of parameters*, *Lazy evaluation*).

applicative order	normal order
strict evaluation	non-strict evaluation
eager evaluation	lazy evaluation
pass by value	pass by name

Reduction steps that do not reach the normal form corresponds to an infinite loop in programming languages.

Strictness and infinite loop

JavaScript is a strict language. `bar(foo("a"))` goes into an infinite loop.

```
function foo(x) {
  return foo(x)
}

function bar(x) {
  return "b"
}

console.log(bar(foo("a")))
```

Haskell is a non-strict language. `bar(foo("a"))` returns `"b"`.

```
foo :: String -> String
foo x = foo x

bar :: String -> String
bar x = "b"

main = putStrLn (bar (foo "a"))
```

10.4 Monad

There are several cases where referential transparency obstacles practical use of the language.

- Random number: It is nonsense that the function `random` returns the same value every time.
- Input/Output: The function `getLine` should return strings typed by humans.

Haskell, a pure functional language, has introduced *monad* to solve the problems above.

- Monad is a simple mechanism that generates a new type from an existing type by adding auxiliary information.
- IO monad is somewhat special.
  - It encapsulates input/output operation as auxiliary information.
  - For example, `getLine` always returns the same value whose type is `IO string`.
  - Actual input/output operations are done by Haskell runtime.
- cf. You Could Have Invented Monads! (And Maybe You Already Have.)

CHAPTER

ELEVEN

MID-TERM PAPER

Exercise 12

Pick one programming language, and answer the following question.

1. Explain features of the language, comparing with other languages.
2. Give a sample program that fully shows the features of the language. (It is not necessary to write it by yourself. It should be a complete program without any omission.)
3. Explain why you choose the sample program above.

Note that:

- The less popular language you pick, the higer score you will get.
- "Features of a language" means syntactic or semantic properties of the language. It does not mean, for example, "there are plenty of libraries", or "there is a very good development environment".
- You should clearly state where you can find the features of the language in the sample program you chose.
- It is not necessary to run the sample program. It is all right if a compiler (or an interpreter) is not available as long as specification of the language is clear enough.
- You are strongly requested to cite references. Otherwise you are considered to commit plagiarism.
- You may use generative AI if you attach an annotation to state so.
- There is no restriction about the length of the paper. Normally 3 or 4 pages long in A4 paper, excluding the example code.

Schedule:

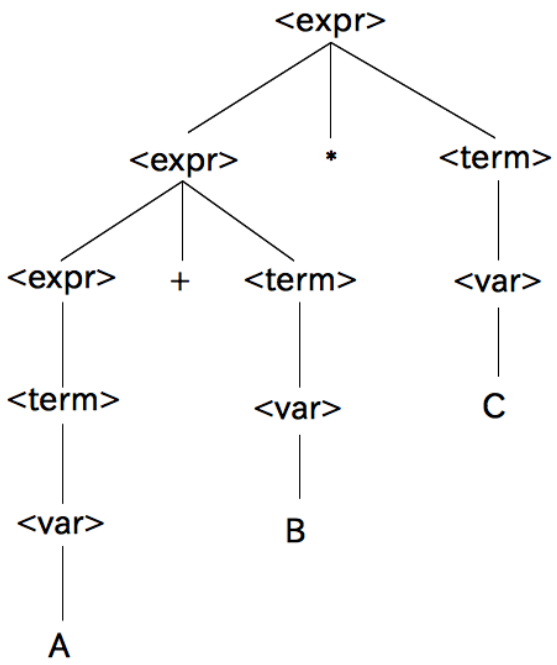
- Submit a draft version by Dec 9th.
- Give a presentation to each other in the class on Dec 11th. Get feedback from other students.
- Revise the paper and submit final version by Dec 27th.

CHAPTER

TWELVE

ANSWER

Exercise 1

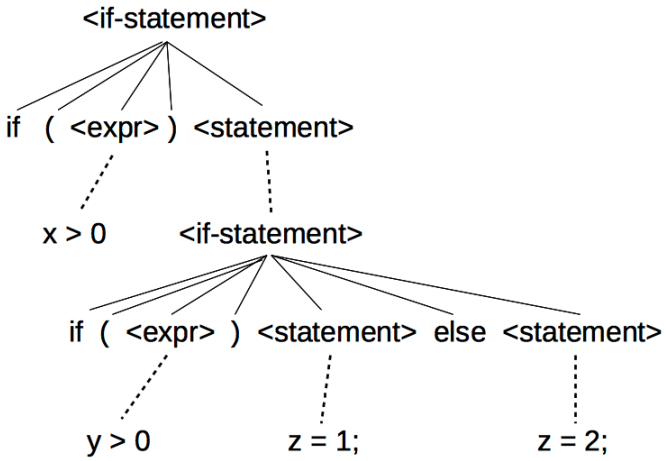


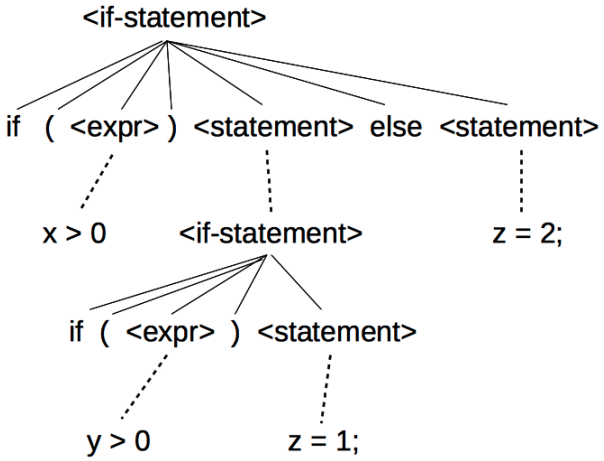
Exercise 2

```
<var> ::= A | B | C
<factor> ::= <var> | ( <expr> )
<term> ::= <factor> | <term> * <factor>
<expr> ::= <term> | <expr> + <term>
```

Exercise 3

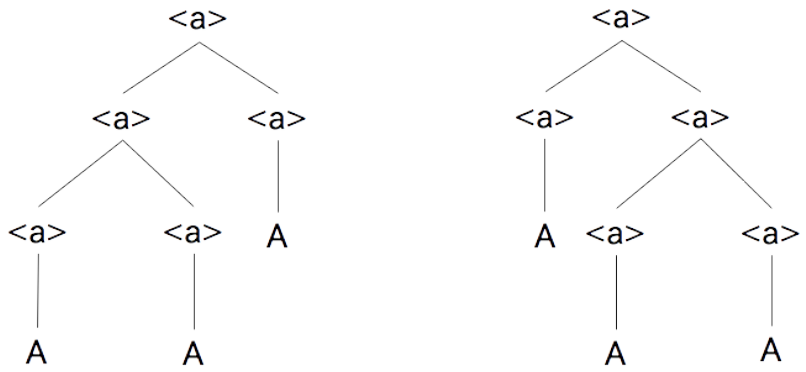
The `else` part can be associated with both of two `if` statements, resulting the following two interpretation:





Exercise 4

In order to prove a certain grammar is ambiguous, we need a sentence which can be parsed in more than one way. In this case, `AAA` is such a sentence.



Exercise 5

	without declarations	with declarations
code length	short	long
misspelling	difficult to find	easy to find
maintainability	easy to write, hard to read	hard to write, easy to read

In many languages, variable declarations are accompanied by information such as types (e.g. int, double), scopes (e.g. global, local), and access modifiers (e.g. public, private):

- Compilers can take advantage of such information (e.g. type checking, memory allocation)
- Code is less flexible because more features are fixed at compile time.

Exercise 6

Static scoping: 5

Dynamic scoping: 10

Exercise 7

Suppose data A and B refer to each other. When all references from outside are lost, A and B are unreachable, but their reference counts are 1.

Exercise 8

1. Suppose there are two variables A and B, and B refers to a heap area C.
2. Data that can be followed from A are marked by GC.

3. Some part of the program is executed where the values of A and B are changed. Now A refers to C, and B refers to somewhere else.
4. Data that can be followed from B are marked by GC.
5. Though A refers to C, C is discarded since it is not marked.

Exercise 9

The library function `strtok` breaks a string into a series of tokens using some delimeters. On its first call we should provide a string for the first parameter, but we should give NULL when we retrieve the succeeding tokens. This is because `strtok` has a static local variable to store the string.

- It should not use global variables because it is a library function.
- It would requre memory allocation of an array of strings if you want to return all tokens at once.

cf. source code (Apple Open Source) , sample usage (TutorialsPoint)

Exercise 10

- **Merits of C**
  - The language specification can be kept small. In other words, it is easier to implement a compiler.
  - It is possible to write concise code if 0 has a special meaning (e.g. end of a character string, absence of data, etc.).

```
while (*q++ = *p++);  /* copy a string from *p to *q */
```

- **Merits of Java**
  - The code may be more human-readable.
  - Some kind of bugs can be detected at the compile time. For example, it is a common mistake in C language to write `if (x = 1)` when we want to compare `x` and `1`. In Java, such code makes a type mismatch error.
  - It may be possible to optimize code using type information (e.g. a boolean value consumes 1byte in the memory while an integer needs 4bytes).

Exercise 11

`(a->b)->[a]->[b]`

Exercise 13

1. `(a#b)$c`
2. `a!(b?c)`
3. `(a#b)!(c$d)`
4. `(a#b)!(c?d)`
5. `((a#b)$c)!(d?e)`

Exercise 14

If the second argument has a side-effect, we may want to use the non-short-circuiting operators so that the side-effect will occur regardless of the value of the first argument.

Otherwise short-circuiting operators are more efficient.

Exercise 15

- `Error` corresponds to a severe incident, so it is difficult to recover by the user program.
- `RuntimeException` includes:
  - exceptions caused by mistakes of programmers (e.g. `ArrayStoreException`)
  - exceptions that have chances to occur at almost everywhere in the program, so it is too cumbersome to write handlers (e.g. `ArithmeticException`)

Exercise 16

1. x: 2, list: {1, 3, 5, 7, 9}
2. x: 2, list: {3, 1, 5, 7, 9}
3. x: 2, list: {3, 2, 1, 7, 9}

Exercise 17

Variable `i` is declared in the surrounding environment of `function() { alert(i); };`, being shared by all generated closures. The value of `i` is 11 after execution of the loop, hence every button shows number 11.

Note that if we used `for (let i = 1; i <= 10; i++)`, it would work fine because the variable name `i` is bound to a newly created variable in each iteration.

Exercise 18

- Subclass may have new methods in addition to methods inherited from its superclass.
- Static type checking should guarantee that no type-related error will happen at run time.

The reason why `A x = new B()` is allowed

> '`x` has type `A`' means we may call methods defined in class `A` for an object stored in `x`. If the object is actually an instance of class B, no error will happen at run time because the object inherits all methods of `A`.

The reason why `B y = new A()` is not allowed

> '`y` has type `B`' means we may call methods defined in class `B` for an object stored in `y`. If the object is actually an instance of class A, an error will happen when we call a method defined in `B` but not `A`.

- genindex
- search

INDEX

P

R

S

T