CHAPTER 3

CONCEPTS

3.1 Variables and lifetime

In functional PLS, a variable stands for a fixed, but a possibly unknown, value. In imperative and object-oriented PLs, a variable contains a value. The variable may be inspected and updated as often as desired. Such a variable can be used to model a real-world object whose state changes over time.

To understand imperative variables, we assume an abstract storage model. A store is a collection of cells, each of which has a unique address. Each cell is either allocated or unallocated. Each allocated cell contains either a simple value or undefined. An allocated cell can be inspected, unless it contains undefined. An allocated cell can be updated at any time.

There are two classes of variables- simple and composite variables. A simple variable is one that contains a primitive value or a pointer. A simple variable occupies a single allocated cell. A composite variable is one that contains a composite value. A composite variable occupies a group of adjacent allocated cells.

A variable of a composite type has the same structure as a value of the same type. For instance, a tuple variable is a tuple of component variables. Similarly, an array variable is a mapping from an index range to a group of component variables. Depending on the PL, a composite variable can be totally updated (all at once), and/or selectively updated (one component at a time).

We illustrate declaration and updating of composite variables in C:

```
struct Date {int y, m, d;}
struct Date xmas, today;

// selective updating
xmas.d = 25;
xmas.m = 12;
xmas.y = 2022;

// total updating
today = xmas;
```

Every variable is created at some definite time, and destroyed at some later time when it is no longer needed. A variable's lifetime is the interval between its creation and destruction. A variable occupies cells only during its lifetime. When the variable is destroyed, these cells may be de-allocated. Moreover, these cells may subsequently be re-allocated to other variable(s).

A global variable's lifetime is the program's entire runtime. It is created by a global declaration. A local variable's time is an activation of a block. It is created by a declaration within that block, and destroyed on exit from that block. A heap variable's lifetime is arbitrary, but can be bounded by the program's runtime. It can be created at any time (by an allocator), and may be destroyed at any later time. It is accessed through a pointer.

The following is a C program, with different variable types.

```
// global variables
extern int x1, x2;

void main() {
    // local variables
    int m1;
    float m2;
    // ..
    f();
    e();
}

void f() {
    // local variables
    float f1;
    int f2;
    // ..
    e();
}

void e() {
    // local variable
    int e1;
}
```

The following is the lifetime of global and local variables.

Global and local variable's lifetimes are nested. They cannot overlap.

Next, we consider a recursive program.

```
void main() {
    float m;
    // ..
    r(3);
}

void r(int n) {
    int i;
    if (n > 1) {
        // ..
        r(n-1);
    }
}
```

The following is the lifetime of the local variables. A local variable of a recursive procedure/function has several nested lifetimes.

Now, we consider heap variables. Consider the C program below.

```
struct IntNode {int elem; IntList succ;}
typedef struct IntNode * IntList;

IntList c (int h, IntList t) {
    IntList ns = (IntList) malloc (sizeof IntNode);
    ns -> element = h;
    ns -> succ = t;
    return ns;
}

void d (IntList ns) {
    ns -> succ = ns -> succc -> succ;
}

void main() {
    IntList l1, l2;
```

```
17      l1 = c(3, c(5, c(7, NULL)));
18      l2 = c(2, l1);
19      d(l1);
20  }
```

The variables `l1` and `l2` are heap variables since they are initialised using the `malloc` keyword. The following is the heap representation of `l1` and `l2`.

The value `5` is not reachable after calling `d`. In this case, the lifetime of the local and heap variables is given below. The lifetime of heap variables can overlap each other, and other local/global variable's lifetimes.

An allocator is an operation that creates a heap variable, yielding a pointer to the heap variable. For example, in C, the function `malloc` is the allocator, and in Java, an expression of the form `new C(..)` is an allocator. A deallocator is an operation that explicitly destroys a designated heap variable. For example, in C, the function `free` is the deallocator. Java does not have a deallocator.

A heap variable remains reachable as long as it can be accessed by following pointers from a local or a global variable. A heap variable's lifetime extends from its creation until:

- it is destroyed by a deallocator,
- it becomes unreachable, or
- the program terminates.

A pointer is a reference to a particular variable. A pointer's referent is the variable to which it refers. A null pointer is a special pointer value that has no referent. A pointer is essentially the address of its referent in the store. However, each pointer also has a type, and the type of a pointer allows us to infer the type of its referent.

Pointers and heap variables can be used to represent recursive variables such as lists and trees. But, the pointer itself is a low-level concept. Manipulation of pointers is notoriously error-prone and hard to understand. For example, the C assignment `p -> succ = q;` appears to manipulate a list, but which list? Also,

- Does it delete nodes from the list?
- Does it stitch together parts of two different lists?
- Does it introduce a cycle?

A dangling pointer is a pointer to a variable that has been destroyed. Dangling pointers arise when

- a pointer to a heap variable still exists after the heap variable is destroyed by a deallocator
- a pointer to a local variable still exists at exit from the block in which the local variable was declared

A deallocator immediately destroys a heap variable. All existing pointers to that heap variable then become dangling pointers. For this reason, deallocators are inherently unsafe.

This means that C is somewhat unsafe and Java is safe. After a heap variable is destroyed in C, pointers to it might still exist. The same is true for local variables after we have exited their block. Java has no deallocator, so pointers to local variables cannot be obtained.

Now, consider the following C code.

```
struct Date {int y, m, d;}
typedef Date * DatePtr;

// date1 points to a newly-allocated heap node
DatePtr date1 = (DatePtr) malloc (sizeof Date);
date1->y = 2022;
date1->m = 1;
date1->d = 1;

// date2 points to the same heap variable
DatePtr date2 = date1;

// deallocates that heap variable- date1 and date2 now contain
// dangling pointers
free(date2);

// behaves unpredictably
printf("%d4", date1->y);
date2->y = 2009;
```

A command (often called a statement) is a program construct that will be executed to update variables. Commands are characteristic of imperative and object-oriented PLs. Simple commands include:

- A skip command is a command that does nothing.
- An assignment command is a command that uses a value to update a variable.
- A procedure call is a command that calls a proper procedure with argument(s). Its net effect is to update some variables.

Compound commands include:

- A sequential command is a command that executes its sub-commands in sequence.
- A conditional command is a command that chooses one of its sub-commands to execute.
- An iterative command is a command that executes it sub-command repeatedly. This may be definite iteration, where the number of repetitions is known in advance, or an indefinite iteration, where the number is not known.
- A block command is a command that contains declarations of local variables.

Below are some examples in Java.

```
// single assignment
m = n + 1;

// multiple assignment
m = n = 0;

// assignment with a binary operator
m += 7;
n /= b;

// if command
if (x > y) {
    System.out.println(x);
} else {
    System.out.println(y);
}

// switch command
Date today = ...;
switch (today.m) {
    case 1:
        System.out.println("JAN");
        break;
    ...
    case 12:
        System.out.println("DEC");
        break;
}

// while command
Date[] dates;
// ...
int i = 0;
while (i < dates.length) {
    System.out.println(dates[i]);
    i++;
}

// for command
for (int i = 0; i < dates.length; i++) {
    System.out.println(dates[i]);
}

// block command
if (x > y) {
    int z = x;
    x = y;
    y = z;
}
```

The primary purpose of evaluating an expression is to yield a value. In most imperative and object-oriented PLs, evaluating expressions can also update variables. These are side effects. In C and Java, the body of a function is a command. If that command updates a global or a heap variable, calling the function has side effects. In C and Java, assignments are expressions with side effects- `V = E` stores the value of `E` in `V`, as well as yielding that value.

The C function `getchar(fp)` reads a character and updates the file variable `fp` points to. So, the following C code is correct.

```
char ch;
while ((ch = getchar(fp)) != NULL) {
```

```
3     putchar(ch);
4 }
```

On the other hand, the following C code is incorrect.

```
1 enum Gender {FEMALE, MALE};
2 Gender g;
3 if (getchar(fp) == "F") {
4     g = FEMALE;
5 } else if (getchar(fp) = "M") {
6     g = MALE;
7 }
```

This is wrong since the second invocation of `getchar(fp)` has moved past the character.

3.2 Bindings and Scope

The meaning of an expression/command depends on the declarations of any identifiers used. A binding is a fixed association between an identifier and an entity (such as a value, variable or procedure). An environment (or namespace) is a set of bindings.

Each declaration produces some bindings which are added to the surrounding environment. Each expression/command is interpreted in a particular environment. Every identifier used in the expression/command have a binding in that environment.

The following is a C program outline, showing the environments for the two functions.

```
extern int z;
extern const float c = 3.0e6;

void f() {
    // ..

    // ENVIRONMENT:
        // c -> the FLOAT value 3 X 10^6
        // f -> a VOID -> VOID function
        // g -> a FLOAT -> VOID function
        // z -> an INT global variable
}

void g (float x) {
    char c;
    int i;
    // ..

    // ENVIRONMENT:
        // c -> a CHAR local variable
        // f -> a VOID -> VOID function
        // g -> a FLOAT -> VOID function
        // i -> an INT local variable
        // x -> a FLOAT local variable
        // z -> an INT global variable
}
```

The scope of a declaration (or of a binding) is the portion of the program text over which it has effect. In some early PLs, the scope of every declaration was the whole program. In modern PLs, the scope of each declaration is controlled by the program's block structure.

A block is a program construct that delimits the scope of any declarations within it. Each PL has its own forms of blocks:

- In C, block commands (`{ ... }`), function bodies and compilation units are used;
- In Java, block commands (`{ ... }`), method bodies and class declarations are used;
- In Haskell, block expressions (`let ... in ...`), function bodies and modules are used.

A PL's block structure is the way in which blocks are arranged in the program text.

Some PLs such as Cobol have monolithic block structure- the whole program is a single block. The scope of every declaration is the whole program. This is depicted in the figure below.

Some PLs such as Fortran have flat block structure- the program is partitioned into blocks, but these blocks may not contain inner blocks. This is depicted in the figure below.

Modern PLs have nested block structure- blocks may be nested freely within other blocks. This is depicted in the figure below. With nested block structure, the scope of a declaration excludes any inner block whether the same identifier is declared.

For example, C has a flat block structure for functions, but nested block structure for variables. This is depicted in the figure below. Having a flat block structure for functions means that we cannot declare functions within functions, but that they can be accessed from anywhere within the program.

A binding occurrence of identifier I is an occurrence of I where I is bound to some entity e. An applied occurrence of identifier I is an occurrence of I where use is made of the entity e to which I is bound. If the PL is statically scoped, every applied occurrence of I should correspond to exactly one binding occurrence of I. Below we have a C program, with binding occurrences and applied occurrences. For each applied occurrence, there is one binding occurrence.

A PL is statically scoped if the body of a procedure is executed in the environment of the procedure definition. Then, we can decide at compile-time which binding occurrence of an identifier corresponds to a given applied occurrence.

A PL is dynamically scoped if the body of a procedure is executed in the environment of the procedure call site. Then, we cannot decide until runtime which occurrence of an identifier corresponds to a given applied occurrence, since the environment may vary from one call site to another.

The following is a program in C, a statically scoped language. In a hypothetical, dynamically scoped PL, we get the following situation.

Dynamic scoping fits badly with static typing. Nearly all PLs are statically scoped. Only a few PLs, such as Smalltalk and Lisp are dynamically scoped.

A declaration is a program construct that will be elaborated to produce binding(s). A declaration may also have side effects, such as creating a variable. A definition is a declaration whose only effect is to produce binding(s). It has no side effects.

Simple declarations include:

- A type declaration binds an identifier to an existing or new type.
- A constant definition binds an identifier to a value, possible after computation.
- A variable declaration binds an identifier to a newly-created variable.
- A procedure definition binds an identifier to a procedure.

A recursive definition is one that uses the bindings it produces itself. In almost all PLs, recursion is restricted to type (or class) declarations and procedure (or method) declarations.

An example below is given of a recursive Java class, with a recursive method.

```
class IntList {
    int head;
    IntList tail;

    static int length(IntList list) {
        if (list == null) {
            return 0;
        } else {
            return 1 + length(list.tail);
        }
    }
}
```

C struct type declarations may be recursive, but only via pointers. Also, C function definitions may be recursive. Below is the C version of the class `IntList` above.

```
struct IntList {
    int head;
    struct IntList * tail;
}

int length(IntList * list) {
    if (list == NULL) {
        return 0;
    } else {
        return 1 + length(list->tail);
    }
}
```