Notes for Dragon Book

Declarative vs. Imperative

Broadly speaking, computer programming languages have been divided into two categories — imperative languages and declarative languages. Examples of imperative languages are Pascal, C, Java, etc. Examples of declarative languages are ML, pure Lisp and pure Prolog.

The programming model in imperative languages is based on a statement-at-a-time paradigm where each statement has some effect on a memory store. Imperative programming is centered around the assignment statement, which allows one to change the content of cells in the memory store. Programs written in imperative languages are generally harder to write, debug, and maintain compared to those written in declarative languages. Imperative programming lays more stress on "how" a solution procedure is specified. Programs written in imperative languages are generally larger in terms of code size and run faster compared to programs written in declarative languages. Imperative languages do not have a solid mathematical basis.

The programming model in declarative languages is based on stating the relationship between inputs and outputs. The actual computation procedure adopted is left to the runtime system. A declarative program can be viewed as a high level specification. Declarative programs are shorter, supposedly easier to write, debug, and maintain. Declarative programs are generally slower than imperative programs in execution speed. There are two major programming paradigms that are declarative: functional programming (with its formal basis in mathematical functions and the lambda calculus) and logic programming (with its formal basis in first order logic).

Static vs. Dynamic scoping

Consider the following pseudocode with dynamic scoping.

```
int x = 10;

int f() {
    return x;
}

int g() {
    int x = 20;
    return f();
}

int main() {
    printf(g());
}
```

Dynamic scope output:

```
20
```

Static scope output:

```
10
```

Dynamic scope resolution is also essential for polymorphic procedures, those that have two or more definitions with the same name, depending only on types of the arguments.

Names, Identifiers and Variables

An *identifier* is a string of characters, typically letters or digits, that refer to (identifies) an entity, such as a data object, a procedure, a class, or a type. All identifiers are names, but not names are identifiers. Names can also be expressions. For example, the name `x.y` might denote the field `y` of a structure denoted by `x`. Here, `x` and `y` are identifiers, while `x.y` is a name, but not an identifier. Composite names like `x.y` are called qualified names.

A *variable* refers to a particular location of the store. It is common for the same identifier to be declared more than once. Each such declaration introduces a new variable. Even if each identifier declared just once, an identifier local to a recursive procedure will refer to different locations of the store at different times.

Aliasing

```c
#include <stdlib.h>
#include <stdio.h>

void p(int x, int y) {
    x = 9890;
    printf("p: x=%d  y=%d\n", x, y);
}

void t(int *x, int *y) {
    x[1] = 377;
    printf("t: x[1]=%d   y[1]=%d\n", x[1], y[1]);
}

void q(int x[], int y[]) {
    x[1] = 42;
    printf("q: x[1]=%d   y[1]=%d\n", x[1], y[1]);
}

void main() {
    int a[] = {1, 2, 3};
    int *b = malloc(sizeof(int) * 3);
    int c = 1;
    b[0] = 1; b[1] = 2; b[2] = 3;

    q(a, a); t(b, b); p(c, c);
}
```

Output:

```
q: x[1]=42   y[1]=42
t: x[1]=377  y[1]=377
p: x=9890    y=1
```