Introduction - “this is a program”

Contents

Motivations for the art of programming

There exist two main reasons why people write programs:

- to automate tasks, so the tasks can be performed by a computer or robot instead of a person; this is called *software engineering*;
- to create structure in human thoughts, so the thoughts can be checked more thoroughly by other people or by computers, this is called *formalization*.

A programmer writing programs can be motivated by either of these reasons or both simultaneously. Meanwhile, *different programming languages exist that are better suited to one or the other* of these activities.

Historically, most programmers have first been interested in automation, so the largest number of courses, tutorials, programming languages and software utilities is dedicated to support learning and practicing software engineering. For example, *Java is a programming language that was designed expressely for software engineering*, and the ecosystem of Java programming tools helps in this direction.

However, there exists also a world of programming where practitioners are primarily interested in writing equations that externalize how people think, so they can write down their abstract ideas formally and share them precisely with others. *Only few people practice this formalization activity, but a lot of their ideas have proven very powerful to software engineers: they often make the task*

of software engineering simpler and more efficient, and they provide concepts that can be reused directly across many programming languages. Although this course focuses primarily on software engineering and Java, I will sometime refer to such formal concepts, especially those that can be reused across multiple programming languages.

This is a program

Let us consider the following Python program (`Max.py`):

```
a = [1.55556, 3.2123, 3.11, 1, 0, 2]
m = 0
i = 0
while i < len(a):
   if m < a[i]:
      m = a[i]
   i = i + 1
print("Maximum is ", m)
```

A program in a language like Java or Python is like a recipe book: any part of it describes a *sequence of steps* to perform. In the case of this example, there is one such step per line in the first 3 lines; then a line with the special word `while` which indicates to *repeat* the next 3 lines multiple times. When the repetition is completed, the step described in the last line is performed.

Such a high-level explanation is sufficient to get a general intuition, but a more careful analysis is needed to get a *precise* understanding of what this program does.

Syntax and semantics

The text of a program, also called its *code*, is composed of words and punctuation. There are three types of words in every language: *keywords* are words reserved by the language that have always the same meaning; *identifiers* or *names* are words that are freely chosen by a programmer to *designate* abstract values or things; and *literal strings* are words that represent themselves.

How does one distinguish keywords from identifiers? Text editors will often automatically recognize keywords and give them a different color, like in the example above. Usually, a programming language only has very few keywords so it is possible to learn them by heart.

For example, in the program above, "`while`" is a keyword, "`len`" and "`print`" are identifiers chosen by programmers whom you likely haven't met, "`a`", "`m`" and "`i`" are identifiers chosen by myself, and "`"Maximum is"`" is a literal string that represents the text "Maximum is".

The meaning of a program, also called its *semantics*, is given by rules in the programming language. The *semantic rules* in a programming language are rules that *associate a general meaning to every program fragment that use a particular syntax form*.

Example semantic rules

To start, the language Python has a semantic rule that looks like this:

1) *Assignment*:

   Syntax: <identifier> = <expression>

Semantics: a statement of this form, when encountered during execution, will assign the value of the expression on the right side of "`=`" to the *variable* designated by the name on the left side. If the variable does not exist yet, it is created automatically.

Using this rule, we can give meaning to a construct like "`m = 0`": when this line is reached during execution, the value 0 is assigned to the variable designated by `m`. Since no such variable exists yet, it is created at that point. Likewise, for "`i = i + 1`": when this line is reached during execution, the value computed by `i + 1` is stored in the variable designated by `i`, which was created by `i = 0` a few steps earlier.

What is a variable? *A variable is a location in the computer's memory where you can store values.* In most cases, variables can be modified: the value stored "in the variable" can be changed by the program over time. Changing the value of a variable is what is called an "assignment".

> **Note**
> Most languages designed for software engineering have at least one construct to perform assignments to variables, ie. change their value. The family of all programming languages that provide variables and assignments is collectively called "imperative languages". Python and Java are such imperative languages.
> In Java, the syntax for assignments is the same as in Python: a name followed by a single "`=`" followed by an expression. It has a slightly different semantics however: the variable is not created automatically by an assignment in Java; it must be defined separately, using a separate construct which will be described later.

The other semantic rules that are key to understand the example program above are:

2) *Addition*:

   Syntax: <expression> `+` <expression>

   Semantics: the expressions on both sides are first evaluated; the value of the combined expression is the result of adding the value of the expression on the left side to the value of the expression on the right side.

3) *Comparison*:

   Syntax: <expression> `<` <expression>

   Semantics: the expressions on both sides are first evaluated; the value of the combined expression is true if and only if the value of the expression on the left side is strictly lower than the value of the expression on the right side.

4) *Repetition*

   Syntax:

   `while` <expression> :
       <block>

Semantics: the block is executed repeatedly as long as the expression on the first line evaluates to true.

5) *Indexing*

   Syntax: <identifier> `[` <expression> `]`

   Semantics: the expression between the square brackets is evaluated, then its value is used as index to one element of the array (table) identified by the name on the left.

6) *Function call*

   Syntax: <identifier> `(` <expression> `[ ,` <expression> `]*` `)`

   Semantics: when evaluating an expression of this form, first the expressions between parentheses are evaluated; then the function designated by the name on the right is invoked using the value of the expressions between parentheses as input arguments; when it completes execution, its return value becomes the value of the entire expression.

> **Note**
> The syntax for this last rule uses the notation `[ ]*` to indicate zero or more repetitions of the syntax between `[]`. Thanks to this, the rule applies to both `len(a)` and `print("Maximum is", m)`.

7) *Conditional execution*

   Syntax:

   `if` <expression> :
       <block>

   Semantics: the expression is evaluated; then the block is executed if and only if the resulting value is true.

> **Note**
> Rules #2, #3, #5, #6 are the same in Java. Java also has repetition and conditional execution, however it has a different syntax for it. This will be described later.

Program decomposition

Using these rules, we can *decompose* the meaning of a program by *recognizing which semantic rule apply to each syntax form*:

- the 1st line, by rule #1, is an assignment; it creates a variable called "`a`" and assigns the table of 6 values `[1.55556, 3.2123, 3.11, 1, 0, 2]` to it.
- the 2nd line, by rule #1, is also an assignment; it creates a variable called "`m`" and assigns the value 0 to it.

- the 3rd line, by rule #1, is also an assignment; it creates a variable called "`i`" and assigns the value 0 to it.
- the 4th line to the 7th line, as a whole, are a repetition due to rule #4. The block formed by lines 5-7 is repeated as long as the expression `i < len(a)` evaluates to true.
- the form `i < len(a)`, is a comparison due to rule #3; it compares the current value of the variable designed by `i` to the value of `len(a)`.
- the form `len(a)` is a function call due to rule #6; it invokes the function `len` on the value of the variable named `a`, which is the table of 6 elements. As the Python documentation then reveals, `len` returns the number of elements in the array, so it is always 6 in this program.
- the 5th and 6th lines, together, are a conditional execution due to rule #7. It evaluates the condition `m < a[i]` then executes `m = a[i]` if and only if the condition is true.
- `m < a[i]` is a comparison as per rule #3. See above.
- `a[i]` is an indexing as per rule #5; it retrieves the element in the array/table named `a` at the position indicated by `i`.
- `m = a[i]` is an assignment as per rule #1. See above.
- `i = i + 1` is an assignment as per rule #1. See above. In turn it uses `i + 1` which is an addition as per rule #2.
- `print("Maximum is ", m)` is a function call as per rule #6.

Once the program is decomposed, we can reconstruct its overall meaning by reading aloud the meaning of the individual lines:

1. create a variable `a` and set it to the table of 6 values;
2. create a variable `m` and set it to 0;
3. create a variable `i` and set it to 0;
4. repeat steps 5-7 below as long as `i` is strictly smaller than 6:
5. if `m` is strictly lower than the item in `a` at position `i`, then do step 6; otherwise go directly to 7;
6. set `m` to the item in `a` at position `i`;
7. set `i` to the sum of the (previous) value of `i` and 1;
8. print the text "Maximum is " followed by the value of `m`.

Do you recognize it? This example program computes the maximum value in the table of 6 values.

This intellectual step where you recognize the overall meaning of a program fragment from a step-wise, piecemeal decomposition is called *reverse-engineering*. This task is not easy initially, but it becomes easier with practice. Also, in many cases the programs you will read will contain comments and annotations by their original writer, to help you understand the program code.

Important concepts

You must be able to explain in your own words:

- *software engineering* and *formalization*;
- *sequence of steps*
- program text = *code*;
- *keywords*, *identifiers/names*;
- *syntax*, *semantics*, *semantic rules*, and how to read semantic rules;
- *variables* and *assignment*;
- *addition*, *comparison*, and *indexing* for expressions;
- *function calls*;
- *conditional execution*, *repetition* (note there are differences here between Java and Python)
- *program decomposition*
- *reverse-engineering*

Further reading

- Absolute Java, section 1.2 (pp. 13-20)
- Think Java, chapter 1 (pp. 1-12), sections 2.2, 2.3, 2.5-2.7, 2.9-2.11 (pp. 15-24)
- Introduction to Programming, section 1.4 (pp 8-10), section 2.2.1 (pp 23-25), section 3.3.2 (pp. 82-84), section 3.5 (pp 96-104), 4.1 (pp. 135-137)

Copyright and licensing