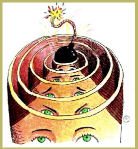
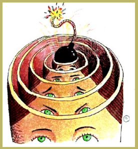


Lecture 9: Programming Languages: Semantics

Aims:

- To look at some subtle issues in the semantics of languages such as Java;
- To discuss the role of compilers and interpreters.



9.1. Semantics

- Grammars define syntactically correct programs. But what do the programs mean?
- Semantic rules define the meaning of syntactically correct programs. These rules must also be precise and unambiguous.

  We shouldn't have to guess what a fragment of program means. If we do, we'll probably get some guesses wrong. Even if we were capable of guessing correctly, the computer cannot make guesses.

  I've seen lots of student programs, for example, where what is needed is an **if** command, e.g. to test whether a button on the screen has been pressed:

```
if (event.getSource() == startButton)
{    ...
}
```

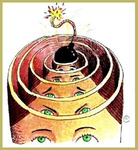
  but what the student has written is a **while** command:

```
while (event.getSource() == startButton)
{    ...
}
```

  These students have the wrong semantics in their heads!

- To illustrate the issues further, we will look at a few fragments of Java and ask what they mean. Specifically, we'll ask what happens when they are executed. The examples I have chosen are again ones where I have found that students have picked up wrong or incomplete semantics.
- **Example 1.**

  What happens here. . .



```
int x = 0;
double y = ...;

x = (int) y;
```

when

- y = 3.2
- y = 3.7
- y = −3.5

- **Example 2.** Assume marks is an array of integers. Would you be happy with this way of computing your average mark?

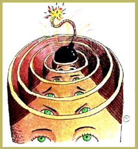
```
public double getAverage(int[] marks)
{    int total = 0;
     for (int i = 0; i < marks.length; i++)
     {   total += marks[i];
     }
     return total / marks.length;
}
```

- **Example 3.** In maths

$$x = \left(\frac{x}{y}\right) \times y$$

In Java, does

x == (x / y) ∗ y



If x and y are variables of type **int**, is it true?

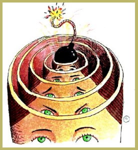
If x and y are variables of type **double**, is it true?

- **Example 4.**
  - What happens when we execute this:
    int x = 4 / 0;
  - What happens when we execute this:
    double x = 4.3 / 0.0;
- **Example 5.**
  - What happens when we execute this:
    int x = Integer.MAX_VALUE ∗ 2;
  - What happens when we execute this:
    double x = Double.MAX_VALUE ∗ 2.0;
- **Example 6.** Given an array a, what are the differences between this

```
for (int i = 0; i < a.length; i++)
{     ...
}
```

and this

```
int len = a.length;
for (int i = 0; i < len; i++)
{     ...
}
```



- To help you decide, suppose we consider this:

```
for (int i = 0; i < a.length; i++)
{    ...
     a = new int[a.length * 2];
     ...
}
```

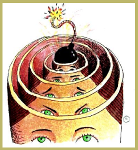
and this

```
int len = a.length;
for (int i = 0; i < len; i++)
{    ...
     a = new int[a.length * 2];
     ...
}
```

- OK, OK. So I've made my point. We need a precise and unambiguous semantics.
- You may be surprised to learn that for most languages, a precise semantics has never been specified!

  Instead, language designers supply language manuals that describe the semantics informally or semi-formally. It's not unusual to find that, when a programming language is extended with new features, the new features interact with the existing ones supported by the language in strange ways.

  In the rare cases where a formal semantics has been defined, it is usually the case that the semantics was defined by some researcher, long after the language was originally designed.



9.1.1. Static vs Dynamic Semantics

- This leads us to the following distinction.

Static Semantics: Semantic rules that can be checked prior to execution.

In many languages, *type checking* is part of the static semantics. For example, according to the BNF grammars we wrote in the previous lecture, the condition in an **if** or **while** command can be any expression. But the semantics of the language might require that these expressions return Booleans. If we check the type of these expressions in advance of execution, then this type checking is part of the static semantics of the language (otherwise, it is part of the dynamic semantics).

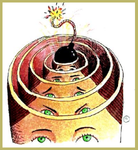
(In Java, in case you're interested, most type-checking is done prior to execution. So people refer to Java as a *statically-typed language*. In fact, for subtle reasons, some type-checks can only be done at run-time in Java. So people sometimes refer to Java as a *mostly* statically-typed language. There are other languages which are *dynamically-typed*, and there are some that don't care much about type checks at all!)

These rules are specified by hanging extra information off the BNF grammar rules.

Dynamic Semantics: Semantic rules that describe the effect of executing programs.

If someone uses the word "semantics" on its own, then you can assume s/he is referring to dynamic semantics: what happens at run-time.

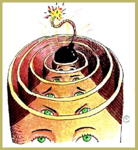
Unlike syntax diagrams or BNF grammars for syntax, there is no widely-accepted metalanguage for dynamic semantics.



9.2. Programs that Eat Programs

- Now that we've discussed how to specify the syntax and semantic of programming languages, we should briefly discuss how the machine makes sense of a program that you have written.
- As you know, you submit your program to another piece of software whose job it is to check its syntax and either execute it or prepare it for execution.
- *Interpreters* are used for the execution of programs.
- *Compilers* are used for the translation of programs.
- Interpreters and compilers are programs which take in other programs!

  There is nothing strange about such programs. One of their parameters is just a piece of text: a program.
- Here's another program that can take in programs.



```
import java.io.*;
public class CharCount
{  public static void main(String[] args)
      throws IOException
   {  String fileName = args[0];
      FileReader reader =
         new FileReader(fileName);
      int count = 0;
      while (reader.read() != -1)
      {  count++;
      }
      System.out.println(count);
   }
}
```

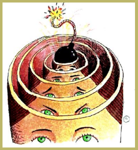
This program reads in a file and counts how many characters it contains. Don't worry if you don't understand all the Java. You may not yet have been taught how to do input from a file. But the algorithm is pretty simple: keep reading and counting characters until you reach the end of the file (when read returns -1).

Of course, we don't really need to write such a program. We can do the same thing in Unix, simply by typing

```
wc -c fileName
```

When we run the program above, we can feed in any file we like. So, if `wombat.txt` contains a joke someone sent you or `me.html` contains your web page, then we can find out how many characters each contains:

```
java CharCount wombat.txt
java CharCount me.html
```



And we can feed it its own source (i.e. the program above):

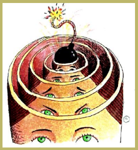
```
java CharCount CharCount.java
```

or its own bytecode:

```
java CharCount CharCount.class
```

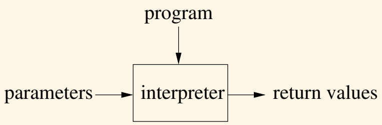
So this is a program that can take in itself or other programs as data. Interpreters and compilers are also programs that take in programs. They could in principle take in themselves, although chances are that this would give an error message (why?).

The point I want to make here is just this: there is nothing strange about one program taking in another as data. And there is nothing strange about a program taking in itself as data.



9.3. Interpreters

- An *interpreter* is a program which executes another program.

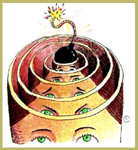


```
while true
{   get next command;
    determine the type of command;
    execute corresponding procedure;
}
```

- An interpreter will have a set of ready-made procedures, e.g. written in machine code, corresponding to each type of high-level command. So the command is examined to see what type it is and this determines which procedure to call.

  (In principle, the procedures could be written in some language other than machine code. In this case, these procedures will themselves have to be given to another interpreter to be executed.)

- The pseudocode above ignores the possibility of error. It may be that the command has a syntax error or a static semantics error; this will be discovered when reading it in and trying to determine its type. Or, it may be that the command gives rise to a dynamic semantics error, when it is executed.

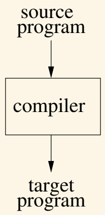


9.4. Compilers

- A compiler translates a program from one programming language, the *source language*, to another programming language, the *target language*.

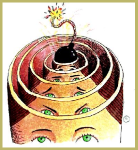
  More usually, the target language is called the *object language*. But, we have already used the phrase 'object language', with a different meaning in our lecture on programming language syntax, so we'll use 'target language' here.

  The source language is usually some high-level language, and the target language is some low-level language (machine code, assembly language or intermediate code), although it is quite possible for the target language to be another high-level language (or even the same high-level language — why might that be useful?).

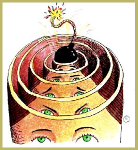
  ```
  while there remain untranslated commands
  {    get next command;
       translate the command;
  }
  ```

- A real compiler is far more complicated than the pseudocode suggests. A compiler may need to read through the program multiple times. For example, it might collect



information on a first *pass*, and then do its translation on a second pass, with the benefit of the information it collected on the first pass. It will also try to carry out as much error-checking as it can, although, of course, it is limited to checking the syntax and the static semantics. It can't check for problems with the dynamic semantics: they will only be discovered when the program is executed.

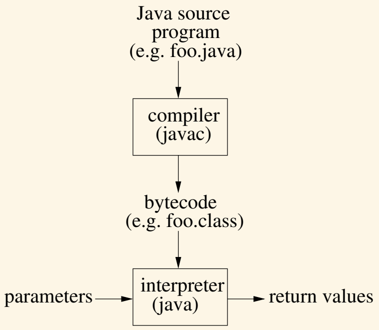
- You can see that, using a compiler, the whole source program gets translated; then we can arrange for execution of the target program. By contrast, when you use an interpreter each command is decoded and immediately executed, one by one.



9.5. Combinations of Compilation and Interpretation

- Our explanation so far has described two ends of a spectrum. At one end is pure interpretation. At the other end is pure compilation (the compiler produces machine code and this is later executed by the hardware). In practice, many languages are handled by various combinations. We'll describe one combination whose popularity has grown because it is the approach used by Java.

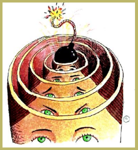
- A Java program is compiled (using the compiler, `javac`) to a program in an intermediate code called *bytecode*. Then the bytecode program is interpreted (by an interpreter, `java`):



Java source program (e.g. foo.java)
compiler (javac)
bytecode (e.g. foo.class)
parameters
interpreter (java)
return values

- This approach has a nice advantage over compiling directly to machine code. If you compile directly to machine code, then your target program can only run on computers with the same machine code.

  But these days portability and mobility are goals. We want to write and compile programs on one machine, but be able to run them on a variety of machines, e.g. machines connected to the Internet. In Java circles, this is summarised by the slogan *"Write once; run anywhere!"*.



The Java solution, as we have seen is to compile to bytecode, an intermediate code, a machine code for an abstract computer. Once compiled, the bytecode can be moved to any machine and executed on that machine, provided a suitable interpreter has been installed on that machine.

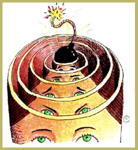
- While this gives portability, the concern of many is that it is an approach that is insufficiently efficient. Interpreting bytecode is much slower than directly executing machine code.

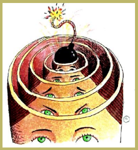
- One option for better performance is to develop hardware that can in fact directly execute the bytecode. Remember, bytecode was designed as the machine code for an abstract machine, a pretend machine, a virtual machine. But why not build these machines for real? Then the bytecode can be executed directly by the hardware. Java chips have been designed and manufactured, but (last time I looked) they are not available commercially.

- Another option is to use what is called a Just-In-Time compiler (a JIT compiler) in place of the interpreter. In other words, we use another compiler which has bytecode as its source language and machine code as its target language. So first Java is compiled to bytecode then, immediately prior to execution, the bytecode is compiled to machine code. The machine code is then directly executed (efficiently) by the hardware. The compiler that translates bytecode to machine code is reasonably simple (much simpler than the Java compiler) and quick, and so this is an approach that can still give portability. Once the Java compiler has produced the bytecode, the bytecode can be moved to any machine and, just prior to execution, the bytecode is compiled to native machine code.

Acknowledgements

This lecture was influenced to a small degree by Section 2.2 of [GJ97].





References

[GJ97] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. Wiley, 3rd. edition, 1997.