COMPSCI 630 Systems **Fall 2019**

Lecture 2

Lecturer: Emery Berger *Scribe: Josh Sennett, Vinita Hissaria*

2.1 Overview

In this class we discussed the limitations of assembly language which led to the development of FORTRAN, a brief review of the language and some compiler optimisation techniques which were used in FORTRAN.

2.2 The Evolution of Computers and FORTRAN

2.2.1 Context

The driving force for the development of computers was military applications, such as calculating artillery trajectory and decryption. The only language used was assembly language and the first automatic computers were custom made and purpose-built. Even with the advent of general purpose computers, it was difficult to code because of the following limitations of assembly language:

- **Low Level** : Assembly language is low-level, hence it is cryptic, difficult to read, write and debug.
- **No Portability** : Code written in assembly language can be run only on the computer it was written for. It cannot be ported to other computers. Also, it was backward incompatible, hence difficult to scale.

2.2.2 FORTRAN

FORTRAN and other high-level languages were created to be portable, readable, writable and fast. It is a domain specific language (created for scientists), in the same way as other languages such as COBOL (for managers), SQL (for databases), Matlab (for engineers), CUDA (for GPU programming).

FORTRAN is a compiled language. It was the first high-level language and predates constructs like parsing, compiling, program analysis, object oriented programming, and structured programming which are used in modern programming languages. It was easier to write than assembly language and portable. Most importantly, FORTRAN's optimizing compiler improved computer performance; at the time, computer performance was generally more important than programmer productivity (although the opposite is often true today).

However, it had drawbacks like the use of specific characters for integer and float variables. With the DO statement, wrongly nested loops are also allowed, and the DO statements can allow the control to go to any line in the code, causing "spaghetti" code (programs with complex control flow). It has liberal rules, which can cause problems as demonstrated below: If we have a statement like **DO 10 I = 1.10** which should have been **DO 10 I = 1,10** the translator does not give any error and instead of a loop, the statement becomes an assignment of a float variable. Since spaces are ignored in FORTRAN, the code gets compiled to **DO10I=1.10** which is interpreted as the assignment of value **1.10** to variable **DO10I**.

Despite its drawbacks, FORTRAN is still used today because there are several improved versions of the language available such as FORTRAN 77 and 90, which have features like recursion, dynamic memory allocation and pointers. There is also working legacy code, which does not require change. Moreover, it is fast, because of its compiler optimizations.

2.3 Compiler Optimization Techniques used in FORTRAN

Compiler optimization is a form of static analysis and transformation which is semantic preserving. It takes place at compile time. Some of the techniques used in FORTRAN are:

- **Constant Folding** : It is the process of computing constant expressions at compile time. For example, x = 3+4 is transformed to x = 7, reducing the number of instructions needed.
- **Constant Propagation** : It is the process of substituting the values of known constants in expressions so as to get more constants and reduce computation at runtime. For example, x = 7, z = 12, y = x+3+z is computed at compile time to x = 7, z = 12, y = 22.
- **Strength Reduction** : It is the substitution of an expensive computation with a cheaper one, like changing x = x**3 to x*x*x, since multiplication is cheaper than exponentiation.
- **Vectorization** : Vectorization is applying an operation to an array of values rather than one at a time, reducing the number of instructions. For example, the loop

  `for (i=0, i < 16, i++)` {`a[i] = 2 * a[i]` } can be replaced with a vector operation on array `a` in parallel rather than a loop of scalar operations.
- **Inlining**: Inlining is replacing a function call with the body of the function. This is beneficial to reduce the overhead to make a function call (copying arguments to locals, creating a stack frame, and jumping to call a function, and the reverse to return from a function). However, inlining is not always beneficial; if all code is inline, instructions are too numerous to fit into an instruction cache resulting in worse performance. Inlining is also not possible for recursive functions. A simple example of inlining is:

  Without inlining:

  ```
  def add(a, b):
      return a + b

  x = 1
  y = 2
  z = add(x, y)
  ```

  With inlining

  ```
  x = 1
  y = 2
  z = x + y
  ```

By making the compiler responsible for these optimizations, programmers can write code that is easier to read and write even though many of these optimizations could have been done by hand.

2.3.1 The Halting Problem

The Halting Problem is an example of the limits of static analysis; a simple example shows that it is impossible to determine whether a program will always terminate.

Let `P` be the simple program:

```
if halts(program):
    run forever
else:
    terminate
```

Running `P` on itself shows a contradiction: if `halts(P)` is true, then P will run forever; if `halts(P)` is false, it will halt.