Compilers

Todd Davies

February 5, 2016

Overview

Any program written in any programming language must be translated before it can be executed on a certain piece of hardware. This translation is typically accomplished by a software system called compiler. This module aims to introduce students to the principles and techniques used to perform this translation and the key issues that arise in the construction of modern compilers.

Syllabus

Introduction What is a compiler? A high-level view of compilation. General structure of a compiler. An overview of compilation technology.

Lexical Analysis (Scanning) Regular languages/expressions, finite state machines, building regular expressions from a finite automaton.

Syntax Analysis (Parsing) Expressing Syntax, Context Free Grammars, Top-Down Parsing, Bottom-Up parsing.

Semantic Analysis Context-sensitive analysis, Attribute Grammars, Symbol Tables, Type Checking.

Intermediate Representations Properties, taxonomy, Graphical IRs, Linear IRs.

Storage Management The Procedure Abstraction, Linkage convention, Run-time storage organisation.

Code Generation Code Shape, Instruction Selection, Register Allocation, Instruction Scheduling.

Topics in Compiler Construction Code Optimisation, JIT Compilation.

Attribution

These notes are based off of both the course notes (`http://studentnet.cs.manchester.ac.uk/ugt/COMP36512/`). Thanks to Rizos Sakellariou for such a good course! If you find any errors, then I'd love to hear about them!

Contribution

Pull requests are very welcome: `https://github.com/Todd-Davies/third-year-notes`

Contents

1 Intro

A compiler is a program that reads another program written in one language and translates it into an equivalent proram written in another language.

An interpreter reads the source code of a program and produces the results of executing the source. Many issues related to compilers are also present in the construction and execution of interpreters.

A good compiler exhibits the following qualities:

- It generates correct code
- It generates code that runs fast
- It confirms to the specification of the input language
- It copes with any input size, number of variables, line length etc
- The time taken to compile source code is linear in its size
- It has good diagnostics
- It has *consistent* optimisations
- It works will with a debugger

An example of a compiler that optimises code is if we had a for-loop such as:

```
A = []
for i = 0 to N:
  A[i] = i
endfor
```

If `N` was always equal to `1`, then the compiler should optimise this to:

```
A = [1]
```

There are two things that any sensible compiler *must* do:

- Preserve the meaning of the program being compiled
- Improve the source code in some way

In addition, it could do some (it's pretty much impossible to do all) of these things:

- Make the output code run fast
- Make the size of the output code small
- Provide good feedback to the programmer; error messages, warnings etc
- Provide good debugging information (this is hard since transforming the program from one language into another often obscures the relationship between an instance of a program at runtime and the source code it was derived from).
- Compile the code quickly