Compilers

Todd Davies

February 6, 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

2 Parts of the compiler

In this section, we'll give a brief overview of all the bits inside a compiler. Most compilers can be roughly split into two parts; the front end and the back end:

The front end is concerned with analysis of the source language, making sure that the input is legal and producing an intermediate representation.

The bacl end translates the intermediate representation into the target code. This usually involves choosing appropriate instructions for each operation in the intermediate representation.

Usually, the front end can run in linear time, while the back end is `NP-Complete`. We cam automate much of the front end of the compiler.

Having an intermediate representation (IR) of the program is so helpful since it means we can completely seperate the front and back ends of a compiler. Concequently, we can have multiple front ends all compiling different languages into the same IR, and multiple back ends producing instructions for different architectures.

If we have n languages and m architectures, then we'd need to write $n \times m$ monolithic compilers, but can instead write n front ends and m back ends $(n + m)$. This is *almost* as good as it sounds; in practice, you need to choose your IR very carefully, since its hard to make it encapsulate everything all the features of a language (a good example of this is compiling Scala onto code that runs on the JVM, where type erasure is a big limitation for Scala).

In more detail, we code goes through the following steps inside the compiler:

Lexical Analysis (front end):
Here, the source language is read in and tokenized (grouped into tokens for later). If the input was `a = b + x`, you might get `(id,a)(=)(id,b)(+)(id,c)` out of the other end. Whitespace is usually ignored (or, depending on the language, incorperated into the parsing), and a symbol table is generated, containing the words that are not reserved words in the language (e.g. variable names).

Syntax Analysis (front end):
Here, the tokens are given a heirarchical structure, often using recursive rules such as Context Free Grammars. The output might be an **Abstract Syntax Tree** (AST), which is an abstract representation of the program (basically an IR).

Semantic Analysis (front end):
Here, we check for semantic errors (such as type checking, flow control checks etc). The AST is annotated with the results of the checks which can be used for optimisation later.

Intermediate code generation (front end):
The AST is now translated into the IR. If the AST is constructed well, and IR is well chosen, then this should be fairly straightforward. Three address code might be an example of an IR.

Optimisation (back end):
The IR is now optimised to increase how quickly it runs, or decrease how much space it uses etc. Some optimisisitions are easier than others, and some are very complex! Often, the optimisation stage is so complex, that it could be a whole seperate part of the compiler (a middle stage in-between the front and back ends).

Code generation (back end):
This phase is concerned with things such as register allocation (NP-Complete), instruction selection (pattern matching), instruction scheduling (NP-Complete) and more. Architecture specific information may be used for further optimisation here, and machine code is the output.