Lecture 2: Compilers, Interpreters & Scheme

Adam Hawley

February 20, 2019

Contents

1 Compilers

"Compilation is file conversion that preserves the semantics of the code". The front end analyses the source code (lexical analysis & parsing) and builds an intermediate representaion consisting of a data structure or a stack.

1.1 Back-End of a Compiler

The back-end of a compiler contains several steps:

- Analysis (data flow, alias analysis...)
- Optimisation e.g:
  - Constant folding: `x := 10 + 20 + 30;`
  - Constant propogation: `x := 32; y := x / 2`
  - Dead code elimination (unreachable or only changing variables that are never read again).
- Code generation

1.2 Portability of Compiled Programs

*"If the front-end and the back-end are perfectly separate, we only need M front ends (for M source languages) and N back-ends (for N different hardware platforms) to produce M*N compilers."*

This is a nice idea in theory but such a *pure* approach makes the implementation difficult. A lot of specialised languages compile to the source code which has compilers for most OSs and hardware, e.g C with gcc.

2 Interpreters

2.1 Benefits of Interpreters

Better Portability An interpreter is normally written in a high-level language and is therefore broadly *machine-independent* (unlike compilers which produce *machine-specific* code)/.

Cheaper Writing an interpreter is less work than writing a back-end.

Better Error Checking & Reporting In the strife for efficiency, compilers may throw away valuable info, e.g source code and line numbers.

Increased Security (Idk about this...) Checking compiled binaries vs. examining the interpreter (contributed to Java's success)

Code Generation The interpreter can execute new code generated by the running program itself. Output data can become code!

The main trade-off between interpreters and compilers is compilation time vs. execution time trade-off.

3 Scheme

See the lecture for info on scheme.