Compiler Overview

Compiler Input and Output

**Text File** Common on Unix.

**Syntax Tree** A structure editor uses its that knowledge of the source language syntax to help the user edit & run the program. It can send a syntax tree to the compiler, relieving it of lexing & parsing.

**Assembly Code** Unix compilers do this. Slow, but easy for the compiler.

**Object Code** .o-files on Unix. Faster, since we don’t have to call the assembler.

**Executable Code** Called a load-and-go-compiler.

**Abstract Machine Code** Serves as input to an interpreter. Fast turnaround time.

**C-code** Good for portability.

Compiler Tasks

**Static Semantic Analysis** Is the program (statically) correct? If not, produce error messages to the user.

**Code Generation** The compiler must produce code that can be executed.

**Symbolic Debug Information** The compiler should produce a description of the source program needed by symbolic debuggers. Try `man gdb`.

**Cross References** The compiler may produce cross-referencing information. Where are identifiers declared & referenced?

**Profiler Information** The compiler should produce profiler information. Where does my program spend most of its execution time? Try `man gprof`.

What’s a Compiler???
Compiler Phases

**ANALYSIS**
- Lexical Analysis
- Syntactic Analysis
- Semantic Analysis

**SYNTHESIS**
- Intermediate Code Generation
- Code Optimization
- Machine Code Generation

One Pass Analysis and Synthesis

Fast. OK for definition-before-use languages like Pascal. No explicit intermediate representation. Target machine code is generated on-the-fly. Very little optimization is possible since we can’t “look forward”. Difficult to retarget, since semantic analysis and code generation are performed simultaneously.

One Pass Plus Peephole Optimization

Better code generation by performing a scan over the machine code and making local improvements.

One Pass Analysis + IR Generation

Machine code is produced from an explicit intermediate representation. Better chances that the front-end & back-end can be recycled.

Compiler Organization I (b)

One Pass Analysis and Synthesis

Source Code

Analysis

Synthesis

Machine Code

One Pass plus Peephole Opt.

Source Code

Analysis

Synthesis

Intermediate Code

Peephole Optimization

Machine Code


Source Code

Analysis

Synthesis

Intermediate Code

Machine Code

Compiler Organization II (a)

Multipass w/ Interm. Files

Early compilers were severely constrained by the size of available primary storage. Therefore the compiler was often organized as a series of passes, where each pass wrote its output to an intermediate file which then became input to the next pass. Still a good design if you’re not worried about speed.

Multipass Analysis

Languages that allow “use-before-declaration”, require the compiler to process the program more than once.

Multipass Synthesis

Highly optimizing compilers usually process the intermediate representation in several passes. Often, we separate machine-independent and machine-dependent optimizations.
Multi-pass Compilation I

- We are going to work with compilers with multi-pass analysis and multi-pass synthesis parts.
- These compilers are very general:
  - They can handle any language, whether free or fixed declaration order.
  - They can produce efficient code.
  - They are portable since the front- and back-ends can be reused for compilers for new languages or new architectures.
- We will assume that the parser builds a tree (an abstract syntax tree) that is modified during semantic analysis, and then used during code generation.
Example I

- Let's go through the compilation of a procedure Foo, from start to finish:
  
  ```pascal
  PROCEDURE Foo ();
  VAR i : INTEGER;
  BEGIN
    i := 1;
    WHILE i < 20 DO
      PRINT i * 2;
      i := i * 2 + 1;
  ENDDO;
  END Foo;
  ```

- The compilation phases are:
  
  Lexical Analysis ⇒
  
  Syntactic Analysis ⇒
  
  Semantic Analysis ⇒
  
  Intermediate code generation ⇒
  
  Code Optimization ⇒
  
  Machine code generation.

---

Example II – Lexical Analysis

- Break up the source code (a text file) and into tokens.

---

Example III/A – Syntactic Analysis

Example III/B – Syntactic Analysis
Readings and References

- Read the Dragon Book:
  Introduction pp. 1–24

- A Simple Compiler pp. 25–82

- Some Compilers pp. 733–744

- Or the Tiger Book:
  Introduction pp. 1–15

Summary I

- The structure of a compiler depends on
  1. the complexity of the language we’re working on (higher complexity ⇒ more passes),
  2. the quality of the code we hope to produce (better code ⇒ more passes),
  3. the degree of portability we hope to achieve (more portable ⇒ better separation between front- and back-ends),
  4. the number of people working on the compiler (more people ⇒ more independent modules).

- Some highly retargetable compilers for high-level languages produce C-code, rather than machine code. This C-code is then compiled by the native C compiler to machine code.
Summary II

- Some languages (APL, LISP, Smalltalk, Java, ICON, Perl, Awk) are traditionally \textit{interpreted} (executed in software by an \textit{interpreter}) rather than compiled to machine code.

- Some interpreters use \textit{dynamic compilation} (or jitting), switching between
  1. interpreting the virtual machine code,
  2. translating the virtual machine code to native machine code,
  3. executing the native machine code,
  4. optimizing the native and/or virtual machine code, and
  5. throwing native code away if it is no longer needed or takes up too much room.

All this is done dynamically at runtime.

Slide 1–24