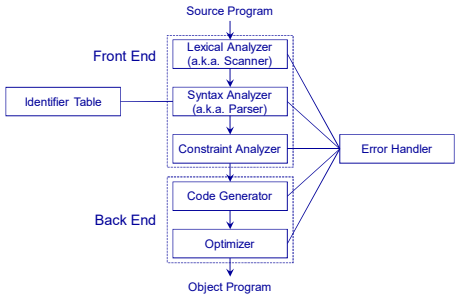
Structure of Compilers

Structure of a Compiler



Comments on Compiler Structure

- Not every phase is separated out as a distinct collection of code modules; e.g., syntax analysis and constraint analysis might be intertwined.
- Many compilers use intermediate code during the compilation process
  - abstract syntax tree (high-level intermediate code representing the basic structure of the program)
  - low-level intermediate code similar to machine code (but machine independent)
- Some optimizations can be performed on the intermediate code.

Compiler Front End

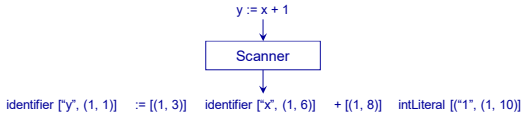
- Analysis
  - lexical analysis
  - syntax analysis
  - constraint analysis
- Determine if the source code is valid
- Determine the intended effect of the program
- Heavily dependent on the source language
- Relatively independent of target machine
- Can include some high-level optimizations

Compiler Back End

- Synthesis
  - code generation
  - optimization
- Generate a semantically equivalent and reasonably efficient machine code program
- Heavily dependent on the target machine
- Relatively independent of source language

Scanner
(Lexical Analysis)

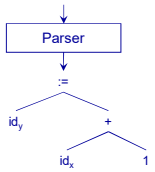
- Identifies basic lexical units of the language
  - called tokens or symbols
  - based on regular expressions
- Removes extraneous white space and comments
- Reports any errors



Parser
(Syntax Analysis)

- Verifies that the grammatical rules of the language are satisfied
- Based on context-free grammars (a.k.a. BNF)

identifier ["y", (1, 1)] := [(1, 3)] identifier ["x", (1, 6)] + [(1, 8)] intLiteral [("1", (1, 10)]



Constraint Analyzer

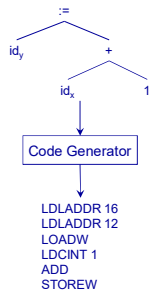
- Handles language requirements that can't be expressed in a context free grammar; e.g., a variable must be declared exactly once if referenced in an assignment statement
- Performs scope and type analysis
- Usually just checks for validity with little or no modification of current representation

Constraint analysis is sometimes referred to as analysis of static semantics.

Code Generator

- Translates intermediate code into machine code or assembly language
- Highly machine dependent

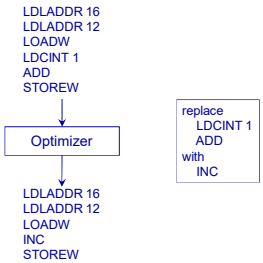
Code Generator
(continued)



Optimizer

- Time/space performance of the code
- Register allocation
- Move invariant computations outside of a loop
- Compile time arithmetic
- Both intermediate and object code optimizations
- Local versus global optimization
- Optimizing compilers

Optimizer
(continued)



Tables and Maps

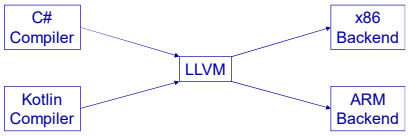
- Often drive the compilation process
  - e.g., table-driven parsers
- Record information (attributes) about identifiers
  - identifier table/map
  - symbol table/map
  - type table/map

Error Handler

- Reports nature and location of errors (error messages)
- Most of the time, a compiler is used to compile incorrect programs!
- Error recovery
  - Turbo Pascal versus traditional approach
  - All errors reported after the first error are suspect.

Compiler Construction Tools

- Scanner/Parser generators; e.g., Antlr, Coco/R, Flex/Bison, Lex/Yacc, JavaCC
- Syntax directed translation engines
- Code generators and optimizers for common low-level intermediate code; e.g., LLVM



Passes

- ***Pass*** – a complete traversal of the source program or an equivalent intermediate representation.
- Often involves disk I/O (i.e., reading and/or writing a file to disk), but the intermediate representation can be in memory.

  Note: Some authors restrict the definition of compiler to a traversal that involves disk I/O, but we will use a more general definition.
- A single-pass compiler makes only one traversal of the source program. (early Pascal compilers)
- A multi-pass compiler makes several traversals.

Single-pass Versus Multi-pass Compilers

- Advantages of multi-pass compilers
  - increased modularity
  - improved ability to perform global analysis (optimization)
  - usually less memory required at run-time (passes overlaid)
  - ideal for multiprocessor systems
  - some languages require more than one pass (e.g., if the defining occurrence of an identifier can follow an applied occurrence)
- Disadvantages of multi-pass compilers
  - can be slower, especially if extra disk I/O is involved
  - usually larger (in terms of SLOCs) and more complex
  - requires design of intermediate language(s)/representation(s)

Passes in the Compiler Project

- Pass 1: Reads/analyzes source text and produces intermediate representation (AST's)
- Pass 2: Performs constraint analysis
- Pass 3: Generates assembly language for the CVM

  (Note: Some optimizations are performed by the assembler.)

For the project, all intermediate passes use in-memory data structures called abstract syntax trees. The only I/O to disk occurs when reading the source file and generating the object code.

Possible Compiler Design Goals
(Conflicts and Tradeoffs)

- Reliability (Rule #1: A compiler must be error free.)
- Modularity/maintenance
- Fast object programs
- Small object programs
- Fast compilation times
- Small compiler size
- Good diagnostic and error recovery capabilities
- Minimize compiler development time

