Library routines and linking (Fortran)

• The compilation of source code counts on the existence of a library of subroutines invoked by the program

Post-compilation assembly (gcc)

• Source code is first compiled to assembly code, and then the assembler translates it to machine code
  – To facilitate debugging (assembly code is easier to read)
  – To isolate the compiler from changes in the format of machine language files (only the commonly shared assembler must be changed)
Source-to-Source Translation

• AT&T C++ compiler
  – To translate C++ programs to C programs
  – To facilitate reuse of compilers or language support

Bootstrapping

• Many compilers are self-hosting:
  – They are written in the language they compile
  – Bootstrapping is used to compile the compiler in the first place
Overview of Compilation

- Character stream
- Token stream
- Parse tree
- Abstract syntax tree or other intermediate form
- Modified intermediate form
- Target language (e.g., assembler)
- Modified target language

Front end:
- Scanner (lexical analysis)
- Parser (syntax analysis)
- Semantic analysis and intermediate code generation

Back end:
- Machine-independent code improvement (optional)
- Target code generation
- Machine-specific code improvement (optional)

Symbol table
Front end & back end

• Front end
  – To analyze the source code in order to build an internal representation (IR) of the program
  – It includes: lexical analysis, syntactic analysis, and semantic analysis

• Back end
  – To gather and analyze program information from IR, to optimize the code, and to generate machine code
  – It includes: optimization and code generation

Scanning (Lexical Analysis)

• Break the program into “tokens”—the smallest meaningful units
  – This can save time, since character-by-character processing is slow

• We can tune the scanner better
  – E.g., remove spaces & comments

• A scanner uses a Deterministic Finite Automaton (DFA) to recognize tokens
A running example: Greatest Common Divisor (GCD)

```c
int main() {
    int i = getint(),
        j = getint();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i)
}
```

Token sequence:

```c
int main ( ) { 
int i = getint ( ) , j = getint ( ) ; 
while ( i != j ) { 
    if ( i > j ) i = i - j ; 
    else j = j - i ;
} 
putint ( i )
}
```

---

Parsing

- Organize tokens into a parse tree that represents higher-level constructs (statements, expressions, subroutines)
  - Each construct is a node in the tree
  - Each construct’s constituents are its children
Semantic Analysis

- Determine the meaning of a program
- A semantic analyzer builds and maintains a symbol table data structure that maps each identifier to the information known about it, such as the identifier's type, internal structure, and scope
Semantic Analysis

• With the symbol table, the semantic analyzer can enforce a large variety of rules to check for errors

• Sample rules:
  – Each identifier is declared before it is used
  – Any function with a non-void return type returns a value explicitly
  – Subroutine calls provide the correct number and types of arguments

Semantic Analysis

• Static semantics
  – Rules that can be checked at compile time

• Dynamic semantics
  – Rules that must be checked at run time, such as
    • Variables are never used in an expression unless they have been given a value
    • Pointers are never dereferenced unless they refer to a valid object
Syntax Tree

• A parse tree is known as a **concrete syntax tree**
  – It demonstrates concretely, how a particular sequence of tokens can be derived under the rule of the context-free grammar
• However, much of the information in a concrete syntax tree is irrelevant
  – E.g., $\varepsilon$ under some branches

Syntax Tree

• In the process of checking static semantic rules, a semantic analyzer transforms the parse tree into an **abstract syntax tree (AST, or syntax tree)** by
  – removing “unimportant” nodes, and
  – annotating remaining nodes with information like pointers from identifiers to their symbol table entries
**Intermediate Form (IF)**

- *Generated after semantic analysis*
  - In many compilers, an **AST** is passed as IF from the front end to the back end
  - In other compilers, a **control flow graph** is passed as IF
Optimization [1]

• High-level optimization
  – Goal: perform high-level analysis and optimization of programs
  – Input: AST + symbol table
  – Output: low-level program representation, such as 3-address code
  – Tasks:
    • Procedure/method inlining
    • Array(pointer) dependence analysis
    • Loop transformations: unrolling, permutation, ...

Optimization [1]

• Low-level optimization
  – Goal: perform low-level analysis and optimizations
  – Input: low-level representation of programs, such as 3-address code
  – Output: optimized low-level representation, and additional information, such as def-use chains
  – Tasks:
    • Dataflow analysis: live variables, reaching definitions, ...
    • Scalar optimizations: constant propagation, partial redundancy elimination, ...
Code Generator [1]

- **Goal:** produce assembly/machine code from optimized low-level representation of programs
- **Tasks:**
  - Register allocation
  - Instruction selection

Reference

Programming Language Syntax

In Text: Chapter 2

Outline

• Basic concepts
  – Programming language, regular expression, context-free grammars
• Lexical analysis
  – Scanner, Deterministic finite automaton (DFA)
• Syntactic analysis
  – Parser
What is a “Language”?

• A language is a set of strings of symbols that are constrained by rules
• A sentence is a string of symbols
• Syntax (Grammar)
  – To describe the structure of a language
• Semantics
  – To describe the meaning or sentences, phrases, or words

Natural languages are ambiguous

• “I saw a man on a hill with a telescope”
• Programming languages should be precise and unambiguous
  – Both programmers and computers can tell what a program is supposed to do