Topic 2: Compiler Front-End

Reading List:

Aho-Sethi-Ullman:

Chapter 3.1, 3.3 ~ 3.5
Chapter 4.1 ~ 4.3
Chapter 5.1, 5.3

(Note: Glance through it only for intuitive understanding.

Also, some slides from 2 and 2a are from other sources such as Prof. Nelson, Prof. W.M. Hsu's slides with modification)
What Does the Front-end Do?

1. Translate programs from source language representation to an internal form suitable for compiler optimization and code generation.

2. Consist of those phases that depend on the source language but largely independent of the target machine.
The Structure of Front End

- **Lexical analysis**
  Stream of characters are grouped into *tokens* for follow up processing

- **Syntax analysis**
  Tokens are grouped hierarchically with target syntactic structure

- **Semantic Analysis:**
  Ensure the components of a program fit together.

- **Intermediate Code Generation**
  A internal representation for later processing: code optimization and generation
Lexical Analysis Example

a := b + c* 100

Lexical analysis:
characters are grouped into seven tokens:
a, b, c : identifiers
:= : assignment symbol
+, * : operators
100 : number
Syntax Analysis Example

\[ a := b + c \times 100 \]

- The seven tokens are grouped into a parse tree.
Semantic Analysis Example

\( a := b + c \times 100 \)

- Checks for semantic errors and gathers type information for code generation.

```
Semantics tree:

:=
  a
    +
      b
        *
          c
          100
  :=
    a
      +
        b
          *
            c
            100

Int-to-real
  100
```
Intermediate Representative Example

```
temp1 = int-to-real(100)
temp2 = id3(c) * temp1
temp3 = id2(b) + temp2
id1(a) = temp3
```
Lexical Analyzer and Parser

source program

lexical analyzer

get next token

token/attribute value

Parser

symbol table

pass token and

IR
Lexical Analysis

Perform **lexical** analysis on the input program, i.e., partition input program text into subsequences of characters corresponding to **tokens**, while leaving out white space and comments.
Lexical Analyzer

Functions
- Grouping input characters into tokens
- Stripping out comments and white spaces
- Correlating error messages with the source program

Issues (why separating lexical analysis from parsing)
- Simpler design
- Compiler efficiency
- Compiler portability
Token definition

How are tokens defined for a programming language and recognized by a scanner?

By using regular expressions to specify tokens as a formal regular language.

Example: Specify language of unsigned numbers (e.g., 5280, 39.37, 0.1, 1.0) as a regular expression
Examples of Tokens

token: smallest logically cohesive sequence of characters of interest in source program

Single-character operators: = + - >
Multi-character operators: := == << >> ->
Keywords: if while
Identifiers: my_variable flag1 My_Variable
Numeric constants/literals: 123 45.67 8.9e+05
Character literals: ‘a’ ‘~’ ‘\’
String literals: “abcd”
Examples of Non-Tokens

White space: space, tab, end-of-line

Comments:
// None of this text forms a token
Regular Expressions (RE)

Why RE?
Suitable for specifying the structure of tokens in programming languages

Basic concept
A RE defines a set of strings (called regular set).
- Vocabulary/Alphabet: a finite character set V
- Strings are built from V via catenation
- Three basic operations: concatenation, alternation ( | ) and closure (*).
Solution

For convenience in defining the regular expression, we introduce a sequence of regular definitions of the form:

- digit → 0 | 1 | ... | 9
- int → digit+
- optional_fraction → . int | ε
- num → int optional_fraction

**Observation:** Only three rules to build a regular expression: concatenation, alternation and closure.
Building a Recognizer for a Regular Language

General approach:
1. Directly build deterministic finite automaton (DFA) from regular expression $E$
2. Build a NFA from regular expression $E$. Simulate execution of NFA to determine whether an input string belongs to $L(E)$

Note: These days, the DFA construction will be done automatically by the *lex* tool.
Example

Use Transition Diagram to Recognize Identifier:

\[ ID = \text{letter(letter | digit)}\ * \]

# indicates input retraction
Mapping transition diagrams into C code:

```
switch (state) {
    ... 
    case 9: c = nextchar();
        if (isletter( c ) ) state = 10; else state = failure();
        break;
    case 10: ....
    case 11: retract(1); insert(id); return;
```
Lex: A Language for Specifying Lexical Analyzers

- Implemented by Lesk and Schmidt of Bell Lab initially for Unix
- Not only a table generator, but also allows “actions” to associate with RE’s.
- Lex is widely used in the Unix community
- Lex is not efficient enough for production compilers, however.
Using Lex

Lex source program lex.l

Lex compiler -> lex.yy.c

lex.yy.c

C compiler -> a.out

Input stream

a.out

sequence of tokens
Syntactic Analysis

- Syntax analysis and context-free grammars
- Bottom-up-parsing

Syntax analysis:

**Parsing**

tokens → parse tree

(syntactic structure of input program)

Based on context-free grammar (CFG)
Context-Free Grammar (CFG)

A context-free grammar is a formal system that describes a language by specifying how any legal text can be derived from a distinguished symbol. It consists of a set of productions, each of which states that a given symbol can be replaced by a given sequence of symbols.
Why CFG

CFG gives a precise syntactic specification of a programming language.

- Automatic efficient parser generator
- Enabling automatic translator generator
- Language extension becomes easier

CFG can be used to replace RE
Syntax Analysis
Problem Statement

Find a derivation sequence in grammar $G$ for the input token stream (or say that none exists).

*Rightmost derivation sequence*: a derivation sequence in which the rightmost nonterminal is replaced in every step.

(Leftmost derivation sequence is defined analogously)
Example of a Grammar

The following grammar describe “lists of digits separated by plus or minus signs”:

\[
\begin{align*}
\text{list} & \rightarrow \text{list} + \text{digit} & (2.2) \\
\text{list} & \rightarrow \text{list} - \text{digit} & (2.3) \\
\text{list} & \rightarrow \text{digit} & (2.4) \\
\text{digit} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 & (2.5)
\end{align*}
\]

Is 9-5+2 a list?

9 is a list (2.4), because 9 is a digit (2.5)
9-5 is a list (2.3), because 9 is a list and 5 is a digit
9-5+2 is a list (2.2), because 9-5 is a list and 2 is a digit
Parse Tree and Derivation

Parse tree can be viewed as a graphical representation for a derivation that ignore replacement order.

**Interior node**: non-terminal symbols

**Leaves**: terminal symbols
Example of Parse Tree

Given the grammar:

\[
\begin{align*}
\text{list} & \rightarrow \text{list} + \text{digit} \\
\text{list} & \rightarrow \text{list} - \text{digit} \\
\text{list} & \rightarrow \text{digit} \\
\text{digit} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

What is the parse tree for 9-5+2?
Abstract Syntax Tree (AST)

The AST is a condensed/simplified/abstract form of the parse tree in which:

1. Operators are directly associated with interior nodes (non-terminals)
2. Chains of single productions are collapsed.
3. Single productions (i.e. exp r -> term) is ignored

[Dragoon book, sec 2.5.1, p70]
Abstract and Concrete Trees

Parse or concrete tree

Abstract syntax tree

9 - 5 + 2

\[ 9 - 5 + 2 \]
Advantages of the AST Representation

- Convenient representation for semantic analysis and intermediate-language (IL) generation
- Useful for building other programming language tools e.t., a syntax-directed editor
Syntax Directed Translation (SDT)

Syntax-directed translation is a method of translating a string into a sequence of actions by attaching such actions to each rule of a grammar.

A syntax-directed translation is defined by augmenting the CFG: a translation rule is defined for each production. A translation rule defines the translation of the left-hand side nonterminal.
Syntax-Directed Definitions and Translation Schemes

A. Syntax-Directed Definitions:
- give high-level specifications for translations
- hide many implementation details such as order of evaluation of semantic actions.
- We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.

B. Translation Schemes:
- Indicate the order of evaluation of semantic actions associated with a production rule.
- In other words, translation schemes give more information about implementation details.
Example Syntax-Directed Definition

term ::= ID
  { term.place := ID.place ; term.code = "" }

term₁ ::= term₂ * ID
  {term₁.place := newtemp( );
   term₁.code := term₂.code || ID.code ||
   gen(term₁.place `:=` term₂.place `*` ID.place)
}

 expr ::= term
  { expr.place := term.place ; expr.code := term.code }

expr₁ ::= expr₂ + term
  { expr₁.place := newtemp( )
   expr₁.code := expr₂.code || term.code ||
   gen(expr₁.place `:=` expr₂.place `+` term.place
  }

YACC – Yet Another Compiler-Compiler

- A bottom-up parser generator
- It provides semantic stack manipulation and supports specification of semantic routines.
- Developed by Steve Johnson and others at AT&T Bell Lab.
- Can use scanner generated by Lex or hand-coded scanner in C
- Used by many compilers and tools, including production compilers.
Parser Construction with YACC

Yacc Specification Spec.y → Yacc Compiler → y.tab.c

y.tab.c → C Compiler → a.out

Input programs → a.out → output

output
Working with Lex

- parse.y
- scan.l

Yacc Compiler
  y.tab.c (yyparse)
  y.tab.h (with –d)

Lex
  lex.yy.c (yylex)

C compiler
  a.out

Source program
  a.out
  output
Working with Lex

- parse.y
- Included
- scan.l
- source program
- a.out
- y.tab.c (yyparse)
- C compiler
- output

Diagram:
- Yacc Compiler
- Lex
- lex.yy.c
- a.out
- Included
- parse.y
- scan.l
- source program
- a.out
- y.tab.c (yyparse)
- C compiler
- output
Summary

- Lexical analysis
  RE
- Syntax analysis
  CFG, Parse Tree
- Semantic Analysis:
  SDT

LEX and YACC