Concepts Introduced in Chapter 2

- A more detailed overview of the compilation process.
  - Parsing
  - Scanning
  - Semantic Analysis
  - Syntax-Directed Translation
  - Intermediate Code Generation

Model of A Compiler Front-End

Context-Free Grammar

- A grammar can be used to describe the possible hierarchical structure of a program.
- A context free grammar has 4 components:
  - A set of tokens, known as terminal symbols.
  - A set of nonterminals.
  - A set of productions where each production consists of a nonterminal, called the left side of the production, an arrow, and a sequence of tokens and/or nonterminals, called the right side of the production.
  - A designation of one of the nonterminals as the start symbol.
- The token strings that can be derived from the start symbol forms the language defined by the grammar.

Example Grammar

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

- A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal.
- The set of terminal strings that can be derived from the start symbol form the language defined by the grammar.
- Parsing is the process of taking a string of terminals and figuring out how to derive it from the start symbol of the language.

Parse Trees

- A parse tree pictorially shows how the start symbol of a grammar derives a specific string in the language.
- Given a context free grammar, a parse tree is a tree with the following properties:
  - The root is labeled by the start symbol.
  - Each leaf is labeled by a token or by ε.
  - Each interior node is labeled by a nonterminal.
  - If A is the nonterminal labeling some interior node and X₁, X₂, ..., Xₙ are the labels of the children of that node from left to right, then A → X₁X₂...Xₙ is a production.

Ambiguous Grammars

- The leaves (tokens) of a parse tree read from left to right form a legal string in the language defined by the associated grammar.
- If a grammar can have more than one parse tree generating the same string of tokens, then the grammar is said to be ambiguous.
- For a grammar representing a programming language, we need to ensure that the grammar is unambiguous or there are additional rules to resolve the ambiguities.

\[
\text{string} \rightarrow \text{string + string | string - string} \\
\text{string} \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

precedence and associativity

- Precedence determines which operator is applied first when different operators appear in an expression and parentheses do not explicitly indicate the order.
- Associativity is used to define the order of operations when there are multiple operators with the same precedence in an expression.
  - Left associativity means that (x op1 y) is applied first in the expression (x op1 y op2 z) when op1 and op2 have the same precedence.
  - Right associativity means that (y op2 z) is applied first in the expression (x op1 y op2 z) when op1 and op2 have the same precedence.
Syntax-Directed Translation

• Syntax-directed translation is the process of converting a string in the language specified by the grammar into a string in some other language.
• Syntax-directed translation is achieved by attaching rules or program fragments to productions in the grammar.
• Execution of these attached rules or program fragments, during parsing, results in the translation of the input string.

Converting Infix to Postfix

• If $E$ is a variable or constant, then the postfix notation for $E$ is $E$ itself.
• If $E$ is an expression of the form $E_1 \text{ op } E_2$, where $\text{op}$ is any binary operator, then the postfix notation for $E$ is $E_1' \ E_2'$ $\text{op}$, where $E_1'$ and $E_2'$ are the postfix notations for $E_1$ and $E_2$, respectively.
• If $E$ is an expression of the form $(E_1)$, then the postfix notation for $E_1$ is also the postfix notation for $E$.

\[ (9-5)+2 \Rightarrow 95-2+ \quad 9-(5+2) \Rightarrow 952+- \]

Syntax-Directed Definition

• Uses a grammar to define the syntactic structure.
• Associates attributes with each grammar symbol.
• Associates semantic rules for computing the values of the attributes.

Example Syntax-Directed Definition

• $\text{seq} \rightarrow \text{seq instr | begin}$
• $\text{instr} \rightarrow \text{east | north | west | south}$

followed by Fig. 2.9, 2.10
**Keeping Track of a Robot's Position**

Input String:
begin west south east east east north north

followed by Fig. A, B, 2.11

**Translation Scheme**

- A translation scheme is a grammar with program fragments called semantic actions that are embedded within the right hand side of the productions.
- Unlike a syntax-directed definition, the order of evaluation of the semantic rules is explicitly shown.

**Syntax Trees**

- Concrete Syntax Tree - a parse tree
- Abstract Syntax Tree
  - Each interior node is an operator rather than a nonterminal.
  - Convenient for translation.

**Parsing**

- Parsing is the process of determining how/if a string of tokens can be generated by a grammar.
- Parsing Methods
  - Top-Down
    - Construction starts at the root and proceeds to the leaves.
    - Can be easily constructed by hand.
  - Bottom-Up
    - Construction starts at the leaves and proceeds to the root.
    - Can accept a larger class of grammars.

followed by Fig. 2.17, 2.18
Recursive Descent Parsing

- Top-down method for syntax analysis.
- A procedure is associated with each nonterminal of a grammar.
- Can be implemented by hand.
  - Decides which production to use by examining the lookahead symbol.
  - The appropriate procedure is invoked for each nonterminal in the rhs of the production.
- Predictive parsing means that a single lookahead symbol can be used to determine the procedure to be called for the next nonterminal.

Example Grammar for Recursive Descent Parsing

- Must not be left recursive.
- Must be left factored.

\[
\begin{align*}
\text{expr} & \rightarrow \text{term rest} \\
\text{rest} & \rightarrow + \text{term} \{ \text{print(’+’)} \} \text{rest} | - \text{term} \{ \text{print(’-’)} \} \text{rest} | \epsilon \\
\text{term} & \rightarrow 0 \{ \text{print(’0’)} \} \\
\text{term} & \rightarrow 1 \{ \text{print(’1’)} \} \\
\text{term} & \rightarrow 9 \{ \text{print(’9’)} \}
\end{align*}
\]

Lexical Analysis Terms

- A token is a group of characters having a collective meaning.
  - id
- A lexeme is an actual character sequence forming a specific instance of a token.
  - num
- Characters between tokens are called whitespace.
  - blanks, tabs, newlines, comments

Inserting a Lexical Analyzer
Recognizing Keywords and Identifiers

- **Keywords** are character strings such as `if`, `for`, `do`, used in languages to identify constructs.
- Character strings for variables, arrays, functions, etc. are returned as **identifiers**.

```plaintext
count = count + increment
=>
<id,count> = <id,count> + <id,increment>
```

- Distinguish keywords from identifiers
  - keywords are reserved in many languages
  - initialize symbol table with keywords

Symbol Table

- Used to save lexemes (identifiers) and their attributes.
- It is common to initialize a symbol table to include reserved words so the form of an identifier can be handled in a uniform manner.
- Attributes are stored in the symbol table for later use in semantic checks and translation.

Symbol Table Per Scope

- Scope of a declaration is the portion of a program to which the declaration applies.
- The **most-closely nested** rule for blocks is that an identifier `x` is in the scope of the most-closely nested declaration of `x`.
- Implementing the most-closely nested rule:
  - create a distinct symbol table for each block.
  - chain the symbol tables in a hierarchical tree structure.

I-values and r-values

- **I-value**
  - Used on the left side of an assignment statement.
  - Used to refer to a location.
- **r-value**
  - Used on the right side of an assignment statement.
  - Used to refer to a value.
Intermediate Code Generation

- The front-end of the compiler produces intermediate code, from which the back-end generates the target program.
- Two important intermediate representation:
  - syntax trees
    - syntax tree nodes represent significant programming constructs
    - provides a pictorial, hierarchical structure
  - three-address code
    - list of elementary programming steps
    - a useful format for code optimization

followed by Fig. 2.39

Three-Address Code

- Format of three-address code instructions:
  - General Format: \( x = y \ op z \)
  - Arrays: \( x[ y ] = z, \ x = y[ z ] \)
  - Copy: \( x = y \)
  - Control flow: \( \) ifFalse \ x goto \ L, \( \)
  - \( \) ifTrue \ x goto \ L, \( \)
  - goto \ L

Translation to Three-Address Code

**Translation of Statements:**

if expr then stmt

ifFalse \ x goto after

code for stmt

code to compute expr into \( x \)

**Translation of Expression:**

\( a[i] = 2*a[j-k] \)

\( t3 = j-k \)
\( t2 = a[t3] \)
\( t1 = 2*t2 \)
\( a[i] = t1 \)

Static Checking

- Static checks are consistency checks done during compilation.
- Static checking includes:
  - Syntactic checking
    - syntax checks that are not enforced by the grammar.
  - Type checking
    - Type checking assures that the type of a construct matches that expected by its context.
    - Coercions: automatic conversion of the type of an operand to that expected by the operator.