Lesson 3

• Structure of compilers
• Overview of a syntax-directed compiler front-end
Compilers and the Analysis-Synthesis Model of Compilation

• Compilers are **language processors**: they translate programs written in a language into equivalent programs in another language

• There are two parts to compilation:
  – **Analysis**: determines the operations implied by the source program which are recorded in a tree structure
  – **Synthesis**: takes the tree structure and translates the operations therein into the target program
Impact of Programming Language evolution on compilers

• Compilers depend on source and target language
  – Have to integrate algorithms to support new programming constructs
  – Have to make high-performance computer architecture effective
  – Optimality of translation for all input programs not decidable. Heuristics for best tradeoff necessary

• Compilers are complex and huge pieces of software. Need support for development
Building compilers

• Compiler design provide examples of real problems solved by abstracting it and applying mathematical techniques
• Is very challenging: design involves not only the compiler, but any (infinite) programs that will be translated.
• Right mathematical models and right algorithms
• Balancing generality and power vs. efficiency and simplicity
Other Tools that Use the Analysis-Synthesis Model

- Editors (syntax highlighting)
- Pretty printers (e.g. Doxygen)
- Static checkers (e.g. Lint and Splint)
- Interpreters
- Text formatters (e.g. TeX and LaTeX)
- Silicon compilers (e.g. VHDL)
- Query interpreters/compilers (Databases)

Several compilation techniques are used in other kinds of systems
Compilation goes through a set of phases

1. Lexical analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator
5. Code Optimizer
6. Code Generator
7. Peephole Optimization

1, 2, 3, 4: Front-End
5, 6, 7: Back-End
Single-pass vs. Multi-pass Compilers

• A collection of compilation phases is done only once (single pass) or multiple times (multi pass)

  • **Single pass**: more efficient and uses less memory  
    – requires everything to be defined before being used  
    – standard for languages like Pascal, FORTRAN, C  
    – Influenced the design of early programming languages

  • **Multi pass**: needs more memory (to keep entire program), usually slower  
    – needed for languages where declarations e.g. of variables may follow their use (Java, ADA, ...)  
    – allows better optimization of target code
Overview of a simple syntax-directed compiler front-end

- Definition of the context-free syntax of a programming language with (Context-Free) Grammars, Chomsky hierarchy
- Parse trees and top-down predictive parsing
- Ambiguity, associativity and precedence
Compiler Front- and Back-end

**Front end analysis**
- Source program (character stream)
  - Scanner
    - (lexical analysis)
  - Tokens
  - Parser
    - (syntax analysis)
  - Parse tree
  - Semantic Analysis
  - Abstract syntax tree, or …
  - Intermediate Code Generation
  - Three address code, or …

**Back end synthesis**
- Three address code, or …
  - Machine-Independent Code Improvement
  - Modified intermediate form
  - Target Code Generation
  - Assembly or object code
  - Machine-Specific Code Improvement
  - Modified assembly or object code
The Structure of the Front-End

Source Program (Character stream) → Lexical analyzer → Token stream → Parser / Syntax-directed translator → Intermediate representation

- Develop parser and code generator for translator
- Syntax definition (BNF grammar)
- IR specification
Syntax Definition: Grammars

• A **grammar** is a 4-tuple $G = (N, T, P, S)$ where
  – $T$ is a finite set of tokens (*terminal symbols*)
  – $N$ is a finite set of **nonterminals**
  – $P$ is a finite set of *productions* of the form $\alpha \rightarrow \beta$
    where $\alpha \in (N \cup T)^* N (N \cup T)^*$ and $\beta \in (N \cup T)^*$
  – $S \in N$ is a designated **start symbol**

• $A^*$ is the set of finite sequences of elements of $A$. If $A = \{a,b\}$, $A^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$

• $AB = \{ab \mid a \in A, b \in B\}$
Notational Conventions Used

• Terminals
  \(a, b, c, \ldots \in T\)
  specific terminals: 0, 1, id, +

• Nonterminals
  \(A, B, C, \ldots \in N\)
  specific nonterminals: expr, term, stmt

• Grammar symbols
  \(X, Y, Z \in (N \cup T)\)

• Strings of terminals
  \(u, v, w, x, y, z \in T^*\)

• Strings of grammar symbols
  \(\alpha, \beta, \gamma \in (N \cup T)^*\)
Derivations

• A one-step derivation is defined by
  \[ \gamma \alpha \delta \Rightarrow \gamma \beta \delta \]
  where \( \alpha \rightarrow \beta \) is a production in the grammar

• In addition, we define
  - \( \Rightarrow \) is leftmost \( \Rightarrow_{lm} \) if \( \gamma \) does not contain a nonterminal
  - \( \Rightarrow \) is rightmost \( \Rightarrow_{rm} \) if \( \delta \) does not contain a nonterminal
  - Transitive closure \( \Rightarrow^* \) (zero or more steps)
  - Positive closure \( \Rightarrow^+ \) (one or more steps)

• \( \alpha \) is a sentential form if \( S \Rightarrow^* \alpha \)

• The language generated by \( G \) is defined by
  \[
  L(G) = \{ w \in T^* \mid S \Rightarrow^+ w \}\]
Derivation (Example)

Grammar $G = (\{E\}, \{+,*,(,),-,id\}, P, E)$ with productions

$P = E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow - E$
$E \rightarrow id$

Example derivations:

$E \Rightarrow - E \Rightarrow - \text{id}$
$E \Rightarrow_{rm} E + E \Rightarrow_{rm} E + \text{id} \Rightarrow_{rm} \text{id} + \text{id}$
$E \Rightarrow * E$
$E \Rightarrow * \text{id} + \text{id}$
$E \Rightarrow^+ \text{id} * \text{id} + \text{id}$
Another grammar for expressions

\[ G = \langle \text{list}, \text{digit} \rangle, \{+,-,0,1,2,3,4,5,6,7,8,9\}, P, \text{list} \rangle \]

Productions \( P = \)

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

A leftmost derivation:

\[
\begin{align*}
\text{list} & \Rightarrow_{lm} \text{list} + \text{digit} \\
& \Rightarrow_{lm} \text{list} - \text{digit} + \text{digit} \\
& \Rightarrow_{lm} \text{digit} - \text{digit} + \text{digit} \\
& \Rightarrow_{lm} 9 - \text{digit} + \text{digit} \\
& \Rightarrow_{lm} 9 - 5 + \text{digit} \\
& \Rightarrow_{lm} 9 - 5 + 2
\end{align*}
\]
Chomsky Hierarchy: Language Classification

- A grammar $G$ is said to be
  - **Regular** if it is right linear where each production is of the form
    \[ A \rightarrow wB \text{ or } A \rightarrow w \]
  - or left linear where each production is of the form
    \[ A \rightarrow Bw \text{ or } A \rightarrow w \quad (w \in T^*) \]
  - **Context free** if each production is of the form
    \[ A \rightarrow \alpha \]
    where $A \in N$ and $\alpha \in (N \cup T)^*$
  - **Context sensitive** if each production is of the form
    \[ \alpha A \beta \rightarrow \alpha \gamma \beta \]
    where $A \in N, \alpha, \gamma, \beta \in (N \cup T)^*, |\gamma| > 0$
  - **Unrestricted**
Chomsky Hierarchy

\[ L(\text{regular}) \subset L(\text{context free}) \subset L(\text{context sensitive}) \subset L(\text{unrestricted}) \]

Where \( L(T) = \{ L(G) \mid G \text{ is of type } T \} \)
That is: the set of all languages generated by grammars G of type T

Examples:
Every finite language is regular! (construct a FSA for strings in \( L(G) \))

\( L_1 = \{ a^n b^n \mid n \geq 1 \} \) is context free
\( L_2 = \{ a^n b^n c^n \mid n \geq 1 \} \) is context sensitive
Parse Trees (context-free grammars)

• Tree-shaped representation of derivations
• The root of the tree is labeled by the start symbol
• Each leaf of the tree is labeled by a terminal (=token) or \( \varepsilon \)
• Each internal node is labeled by a nonterminal
• If \( A \rightarrow X_1 X_2 \ldots X_n \) is a production, then node \( A \) has immediate children \( X_1, X_2, \ldots, X_n \) where \( X_i \) is a (non)terminal or \( \varepsilon \) (\( \varepsilon \) denotes the empty string)
Parse Tree for the Example Grammar

Parse tree of the string $9-5+2$ using grammar $G$

The sequence of leafs is called the \textit{yield} of the parse tree.
Ambiguity

Consider the following context-free grammar:

$$G = \langle\{\text{string}\}, \{+,-,0,1,2,3,4,5,6,7,8,9\}, P, \text{string}\rangle$$

with production $P =$

$$\text{string} \rightarrow \text{string} + \text{string} | \text{string} - \text{string} | 0 | 1 | \ldots | 9$$

This grammar is ambiguous, because more than one parse tree represents the string $9-5+2$
Ambiguity (cont’d)
Associativity of Operators

Left-associative operators have left-recursive productions

\[ left \rightarrow left + \text{term} \mid \text{term} \]

String \( a+b+c \) has the same meaning as \( (a+b)+c \)

Right-associative operators have right-recursive productions

\[ right \rightarrow \text{term} = right \mid \text{term} \]

String \( a=b=c \) has the same meaning as \( a=(b=c) \)
Precedence of Operators

Operators with higher precedence “bind more tightly”

\[
expr \rightarrow expr + term \mid term \\
term \rightarrow term * factor \mid factor \\
factor \rightarrow number \mid ( expr )
\]

String \texttt{2+3*5} has the same meaning as \texttt{2+(3*5)}
Syntax of Statements

\[
\begin{align*}
\textit{stmt} & \rightarrow \texttt{id} := \textit{expr} \\
& \mid \texttt{if } \textit{expr} \texttt{ then } \textit{stmt} \\
& \mid \texttt{if } \textit{expr} \texttt{ then } \textit{stmt} \texttt{ else } \textit{stmt} \\
& \mid \texttt{while } \textit{expr} \texttt{ do } \textit{stmt} \\
& \mid \texttt{begin } \textit{opt}_\texttt{stmts} \texttt{ end} \\
\textit{opt}_\texttt{stmts} & \rightarrow \textit{stmt} ; \textit{opt}_\texttt{stmts} \\
& \mid \epsilon
\end{align*}
\]