CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

- **Syntax Analysis (Ch 4)**
  - **Introduction (Ch 4.1)**
    - **The Role of the Parser (4.1.1)**
    - **Representative Grammars (4.1.2)**
    - **Syntax Error Handling (4.1.3)**
    - **Error-Recovery Strategies (4.1.4) (TBD, in Part2)**

Syntax Analysis

- **Syntax Analysis** is a phase in checking for the structure of the formal grammar.
- It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.
- Syntax Analysis in Compiler Design process comes **after** the Lexical analysis phase.

Syntax vs. Lexical Analyser

Syntax Analyser	Lexical Analyser
The syntax analyser mainly deals with recursive constructs of the language.	The lexical analyser eases the task of the syntax analyser.
The syntax analyser works on tokens in a source program to recognize meaningful structures in the programming language.	The lexical analyser recognizes the token in a source program.

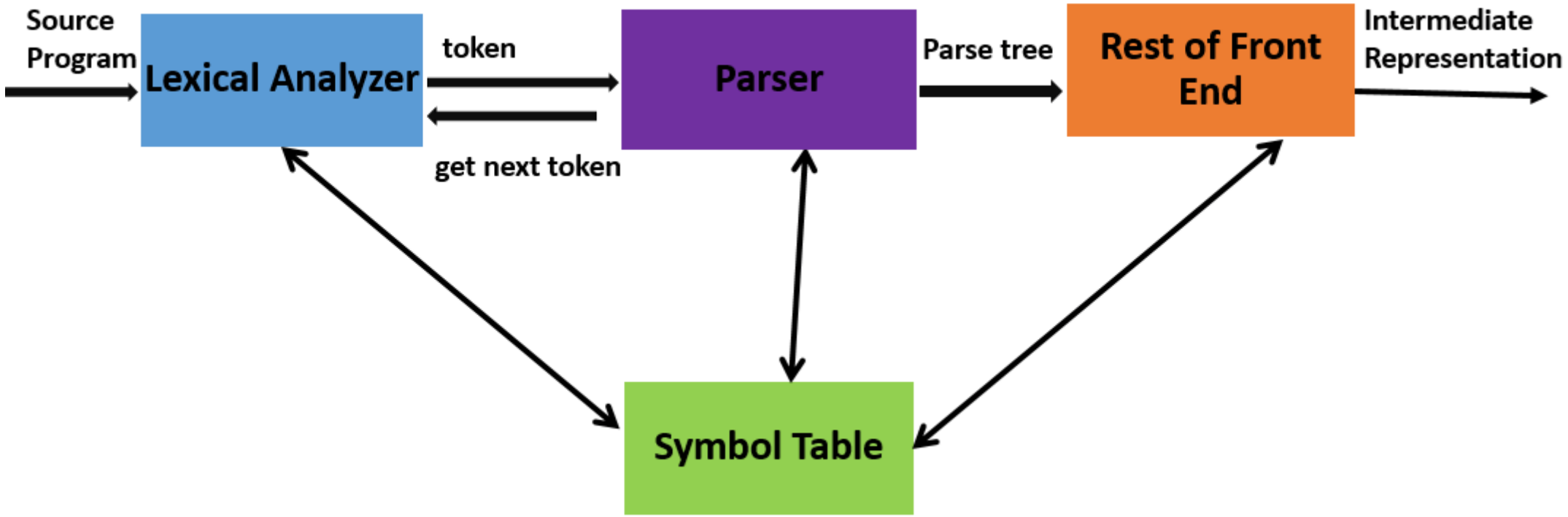
Syntax Analysis

- The jobs for **Syntax Analyzer (parser)**
  - Check if the code is valid grammatically
  - Helps you to detect all types of Syntax errors
  - Find the position at which error has occurred
  - Clear & accurate description of the error. (if it is necessary)
  - Recovery from an error to continue and find further errors in the code.

  (Some modern IDE has such kind of function. i.e. Give you some options to recover from the syntactical errors)
  - The parse must reject invalid texts by reporting syntax errors
  - **Construct the parsing tree.**
- The syntax of programming language constructs can be specified by context-free grammars or **BNF** (Backus-Naur Form) notation, introduced in Section 2.2.

Syntax Analysis

- Here is the process…
- In our compiler model, the parser obtains a sequence of tokens from the lexical analyzer and produces a parse tree.



The Role of the Parser (4.1.1)

- There are three general types of parsers for grammars
  - Universal (inefficient in the production compilers)
  - Top-down (commonly used): build parse trees from the top (root) to the bottom (leaves)
  - Bottom-up (commonly used): start from the leaves and work their way up to the root.
- In either case, the input to the parser is scanned from **left to right**, one symbol at a time

The Role of the Parser (4.1.1)

- The **LL** (top down) and **LR** (bottom-up) parsers are important in practice.
- Hand written parsers are often **LL**.
- Specifically, the **predictive parsers** we looked at in Ch. 2 are for LL grammars.
- The **LR** grammars form a larger class.
  - Parsers for this class are usually constructed with the **aid** of **automatic tools.**

Representative Grammars (4.1.2)

- Think of the following grammar

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
\qquad (4.1)
$$

- E represents **expressions** consisting of terms separated by + signs
- T represents **terms** consisting of factors separated by * signs,
- F represents **factors** that can be either **parenthesized** expressions or **identifiers**:

Representative Grammars (4.1.2)

- Expression grammar (4.1) belongs to the class of **LR** grammars that are suitable for **bottom-up parsing.**
- This grammar can be adapted to handle additional operators and additional levels of precedence
- It **cannot** be used for **top-down parsing** because it is **left recursive.**
- The following **non-left-recursive variant (4.2)** of our **original expression grammar (4.1)** will be used for **top-down parsing:** (see next page)

Representative Grammars (4.1.2)

- (4.2) See? It sometimes, **gives**

epsilon to keep the tree growing

out of control

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
\tag{4.2}
$$

- **[Ambiguity]** The definition of ambiguous grammar is:
  - It can yield the **same final string**
  - It can construct 2 ore more "**different parse trees**"

  Both **a and b**, 2 conditions are holding

Representative Grammars (4.1.2)

- There is another very special example:
- The following grammar treats + and * alike, so it is useful for illustrating techniques for **handling ambiguities** during parsing

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id} \qquad (4.3)$$

- Because if E+E and E*E are treated differently, it has more chances to produce different parsing trees
- In this example, E represents expressions of all types
- Grammar (4.3) **permits more than one parse tree** for expressions like a + b * c.

Representative Grammars (4.1.2)

- Generally speaking, we will try to avoid ambiguity

Syntax Error Handling (4.1.3)

- There are different levels of errors.
  - Lexical errors: For example, the spelling
  - Syntactic errors: For example, missing the ; or extra “{“ , “}”
  - Semantic errors: For example, type mismatches between operators and operands. In Java, “void” for a function means no returns but we put a return value for this function
  - Logical errors: For example, incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator “=“, instead of the comparison operator “==“

Syntax Error Handling (4.1.3)

- The error handler in a parser has goals as follows:
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect **subsequent errors**
  - Add minimal overhead to the processing of correct programs
- How should an error handler report the presence of an error? At the very least, it must report the place in the source program where an error is detected → provide the line number