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)**
  - **Context-Free Grammars (4.2)**
    - **The Formal Definition of a Context-Free Grammar (4.2.1)**
    - **Notational Conventions (4.2.2)**
    - **Derivations (4.2.3) (TBD, in Part3)**

Error-Recovery Strategies

- Some of General Guidelines: (fault-tolerance)
  - Once an error is detected, at least the parser has to quit with an informative error message when it detects the error (No Recovery)
  - A parser can restore itself to a state, where processing of the input can continue with reasonable hopes --- Not just a crash and tell us: I’m dead
  - If errors pile up, it is better for the compiler to give up after exceeding some error limit instead of crashed directly when there is just one error.

Error-Recovery Strategies (Recovery strategies from the book)

- Panic Mode Recovery
  - Once the error happens, the parser discards input until it encounters a *synchronizing token*.
  - The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous.
  - For example, when an error occurs, "…}", for the input, before it is reading to the "}", everything will be discarded!
  - The bad:
    - The panic-mode correction often skips a considerable amount of input without checking it for additional errors
  - The good:
    - Simplicity. Not going into an infinite loop is guaranteed

Error-Recovery Strategies (Recovery strategies from the book)

- **Phrase-Level Recovery**
  - Locally replace some prefix of the remaining input by some string
  - Simple cases are exchanging ; with , and = with ==.
  - Delete an **extraneous** semicolon, or insert a **missing** semicolon
  - The Bad:
    - It might have difficulties when the “real error” occurred long before an error was detected.
- **Error Productions**
  - There are some common errors which are observable
  - We can use the grammar to build up a series of pre-built errors
  - Then, the parser detects the anticipated errors when an error occurs

Error-Recovery Strategies (Recovery strategies from the book)

- **Global Correction**
  - We still hope the compiler to make as few changes as possible in processing an incorrect input string
  - Given an incorrect input, we what to find a parse tree for a related string (contains errors), such that the changes (i.e. insertions, deletions of the tokens) are as small as possible.
  - Hard to implement (costly)
  - Programmers still like low-cost “**Phrase-Level Recovery”**

Context-Free Grammars

- For such kind of expressions and statements, we are already familiar with

$$stmt \rightarrow \mathbf{if}\ (\ expr\ )\ stmt\ \mathbf{else}\ stmt$$

- In this section reviews the definition of a context-free grammar and introduces terminology for talking about parsing.
- Particularly, the notion of “derivations” is very helpful for discussing the order in which productions are applied during parsing.

The Formal Definition of a Context-Free Grammar (4.2.1)

Definition: A Context-Free Grammar consists of

1. Terminals: The basic components found by the lexer. They are sometimes called token names, i.e., the first component of the token as produced by the lexer.
2. Nonterminals: Syntactic variables that help define the syntactic structure of the language.
3. Start Symbol: A nonterminal that forms the root of the parse tree.
4. Productions:
    a. Head or left (hand) side or LHS. For *context-free* grammars, which are our only interest, the LHS must consist of just a single nonterminal.
    b. →
    c. Body or right (hand) side or RHS. A string of terminals and nonterminals.

Terminals are the leaves of the parsing tree
Non-terminals can derive Non-terminals or terminals
Start Symbol is the root of the parsing tree
For example,
X → Y, in CFG, LHS must consist of just a single non-terminal
Y can be terminal or non-terminals

The Formal Definition of a Context-Free Grammar (4.2.1)

- An example, in the following, it defines simple arithmetic expressions.
- In this grammar, terminal symbols are:

id, +, -, *, /, (,)

- The nonterminal symbols are expression, term and factor, and expression is the start symbol

$$
\begin{array}{rcl}
expression & \rightarrow & expression \; \texttt{+} \; term \\
expression & \rightarrow & expression \; \texttt{-} \; term \\
expression & \rightarrow & term \\
term & \rightarrow & term \; \texttt{*} \; factor \\
term & \rightarrow & term \; \texttt{/} \; factor \\
term & \rightarrow & factor \\
factor & \rightarrow & \texttt{(} \; expression \; \texttt{)} \\
factor & \rightarrow & \mathbf{id}
\end{array}
$$

Notational Conventions (4.2.2)

1. These symbols are terminals:

   (a) Lowercase letters early in the alphabet, such as a, b, c.

   (b) Operator symbols such as $+$, $*$, and so on.

   (c) Punctuation symbols such as parentheses, comma, and so on.

   (d) The digits $0, 1, \ldots, 9$.

   (e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

2. These symbols are nonterminals:

   (a) Uppercase letters early in the alphabet, such as A, B, C.

   (b) The letter S, which, when it appears, is usually the start symbol.

   (c) Lowercase, italic names such as *expr* or *stmt*.

   (d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

Notational Conventions (4.2.2)

- Some other rules:

3. Uppercase letters late in the alphabet, such as X, Y, Z, represent *grammar symbols*; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet, chiefly $u, v, \ldots, z$, represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \to \alpha$, where A is the head and α the body.

6. A set of productions $A \to \alpha_1$, $A \to \alpha_2, \ldots$, $A \to \alpha_k$ with a common head A (call them A-*productions*), may be written $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. Call $\alpha_1, \alpha_2, \ldots , \alpha_k$ the *alternatives* for A.

7. Unless stated otherwise, the head of the first production is the start symbol.

Notational Conventions (4.2.2)

- By using the conventions, this one LHS, can be written as this one RHS

$$
\begin{array}{rcl}
\mathit{expression} & \rightarrow & \mathit{expression} + \mathit{term} \\
\mathit{expression} & \rightarrow & \mathit{expression} - \mathit{term} \\
\mathit{expression} & \rightarrow & \mathit{term} \\
\mathit{term} & \rightarrow & \mathit{term} * \mathit{factor} \\
\mathit{term} & \rightarrow & \mathit{term} \, / \, \mathit{factor} \\
\mathit{term} & \rightarrow & \mathit{factor} \\
\mathit{factor} & \rightarrow & (\ \mathit{expression}\ ) \\
\mathit{factor} & \rightarrow & \mathbf{id}
\end{array}
$$

$$
\begin{array}{rcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T \, / \, F \mid F \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$