Language Design

Programmers and compiler writers need to agree upon two key features:

1. **Syntax**: What is the notation for a legal program?
   - Pascal: `X := A + B`
   - Fortran: `X = A + B`
   - Common Lisp: `(SETQ X (+ A B))`
   - C: `X = A + B;`

2. **Semantics**: What does the notation mean?
   - What is the value of the expression `(1 + 2 * 3)`?
     - 7 in C, Fortran, Java
     - 9 in APL, Smalltalk

Formal Syntax Specifications

- Language syntax can be specified **informally** in a variety of documents and manuals.
  - However, it is important to be able to **guarantee** that strings of symbols are interpreted correctly.
- A **formal grammar** is a tool to provide this guarantee.
  - From well-specified formal grammars, it is often possible to **automatically** generate programs to read lines of code.
  - These **scanners** and **parsers** begin the process of compiling them into executable programs.

Formal Grammars

- A **formal grammar** defines **legal strings** of a language.

The grammars we consider have four main elements:

\[ G = (N, \Sigma, P, S) \]

1. A finite set \( N \) of **nonterminal** symbols (grammar variables)
2. A finite set \( \Sigma \) of **terminal** symbols (language symbols)
   \[ (N \cap \Sigma) = \emptyset \]
3. A finite set \( P \) of **production rules**
   \[ N \rightarrow (N \cup \Sigma)^* \]
4. A single **start symbol** \( S \in N \)

**Note**: the Kleene Star in clause 3 means that there are “0 or more occurrences” of symbols from either set (or both).
Backus-Naur Form (BNF)

- BNF is a simple way to specify production rules
- Each rule is of the form:
  \[ A \rightarrow \gamma \]
  1. LHS \( A \) is a single nonterminal symbol
  2. RHS \( \gamma \) is a string of terminal and/or nonterminal symbols
- In these sorts of rules, special symbol \( \varepsilon \) is an empty string
- For convenience, if there are multiple possible replacements for a given nonterminal, we can list them as a single rule, using the vertical bar (|) to separate them

Note: some presentations of BNF replace the arrow with other possible symbols, like \( ::= \)

A BNF Example

\[ G = (N, \Sigma, P, S) \]
- A simple grammar for basic non-negative integers:
  \[ \begin{align*}
  N &= \{ \text{digit, integer} \} \\
  \Sigma &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
  S &= \text{integer} \\
  P &= \{ \text{integer} \rightarrow \text{digit} \mid \text{digit integer}, \\ & \quad \text{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \} 
\end{align*} \]
- The grammar has two rules for replacing the nonterminal symbols (digit or integer) with combinations consisting either of other nonterminal symbols or of the terminal (numerical) final symbols

Language of a Grammar

\[ G = (N, \Sigma, P, S) \]
- Given a formal grammar, we can derive a string consisting entirely of terminal language symbols by:
  1. Starting with the special nonterminal symbol \( S \)
  2. Replacing \( S \) with \( X \) for some rule in \( P \):
     \[ S \rightarrow X \]
  3. Repeating the process for any nonterminal found in \( X \), according to any other rule found in \( P \)
  4. Stopping when no nonterminal symbols are left
- The language of the grammar, \( L(G) \), is the set of all strings that can be derived in that grammar

Note: for convenience, we have underlined the nonterminal that is replaced at each line
Language and Derivation

- We can go from the starting nonterminal \( \text{integer} \) and end up with 543, using rules of grammar \( G \).
- Thus we say that 543 is in the language of \( G \):
  \[
  543 \in L(G)
  \]
- This is also written to indicate that we can derive 543 from \( \text{integer} \) in some number of steps:
  \[
  \text{integer} \rightarrow^{*} 543
  \]

Regular Grammars/Languages

- By allowing various different forms of production rules (or not), we can capture languages of different complexity.
- A regular grammar is one in which every rule in \( P \) is of one of three basic forms:
  \[
  A \rightarrow \epsilon \\
  A \rightarrow a \\
  A \rightarrow bB
  \]
- Where: \( A \neq B \) are single nonterminal symbols and \( a \) and \( b \) are any single terminal symbols.

Regular Grammars/Languages

- As presented, the integer grammar is not regular, since it has a rule with the same nonterminal on both sides, and has rules that have no terminals at all on the right:
  \[
  \text{integer} \rightarrow \text{digit} \mid \text{digit integer}
  \]
  \[
  \text{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
  \]
- Does this mean the integer language is not regular?
  - No: it just means that we don’t know if it is.

Regular Grammars/Languages

- We can change the grammar to make it regular, showing that the integer language is regular too:
  \[
  \text{integer} \rightarrow 0 \text{rest} \mid 1 \text{rest} \mid 2 \text{rest} \mid 3 \text{rest} \mid 4 \text{rest} \mid 5 \text{rest} \mid 6 \text{rest} \mid 7 \text{rest} \mid 8 \text{rest} \mid 9 \text{rest}
  \]
  \[
  \text{rest} \rightarrow \epsilon \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
  \]
  \[
  0 \text{integer} \mid 1 \text{integer} \mid 2 \text{integer} \mid 3 \text{integer} \mid 4 \text{integer} \mid 5 \text{integer} \mid 6 \text{integer} \mid 7 \text{integer} \mid 8 \text{integer} \mid 9 \text{integer}
  \]
- A language is regular if there is some regular grammar for it.
  - This doesn’t mean that every grammar for it is regular.
Another Derivation

- We can show that 543 is part of the language of the new grammar, too, just as it was of the original:

\[
\text{integer} \quad \Rightarrow \quad 5 \text{ rest} \\
\quad \Rightarrow \quad 54 \text{ integer} \\
\quad \Rightarrow \quad 543 \text{ rest} \\
\quad \Rightarrow \quad 543
\]

*Note:* in the last line, we have replaced rest with the empty string (ε)

Limitations of Regular Grammars

- We can use regular grammars to define many aspects of interest in a programming language

  - For example, we can define legal identifiers (variable/class/method names) for Java, C, C++, most other languages, using regular grammars

- There are limits, however: certain more complex parts of languages cannot be expressed using regular grammars:
  - Binary strings consisting of \( n \) 1’s followed by \( n \) 0’s
  - Arithmetic expressions with properly matched parentheses
  - Control structures with properly nested and matched braces
  - Many others...

*Note:* it can be proven mathematically that none of these are possible

Context-Free Grammars/Languages

- A more complex class of languages requires that we allow a variety of different rules, beyond the three basic types allowed by regular grammars

  - A *context-free* grammar allows any rule of the form

\[
A \quad \Rightarrow \quad \gamma
\]

  - Where: \( A \) is a single nonterminal symbol and \( \gamma \) is any combination of nonterminal and terminal symbols (including the same symbol \( A \) itself)

  - This allows us to define far more complex languages

Context-Free Grammars/Languages

- Sometimes the use of a CFG is just for convenience

  - The original integer grammar we saw is context-free, although we don’t have to use a CFG to define it

- Other times, we can define languages using CFG’s that we can’t define with only regular grammar rules

  - Example: the language of binary strings consisting of \( n \) 1’s followed by \( n \) 0’s (\( n \geq 0 \)), with the rules:

\[
S \quad \Rightarrow \quad \epsilon \mid 1S0
\]
This Week

- **Topic:** Formal Syntax
- **Read:** Text, chapters 1, 2.1
- **Office Hours:** Wing 210
  - Wednesday/Friday: 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Thursday: 2:00 PM – 3:00 PM