Formal Grammars and BNF

Robert Lowe

Department of Computer Science
Southeast Missouri State University

Outline

Outline

SOUTHEAST MISSOURI STATE UNIVERSITY · 1873®

Why use formalisms?

- Precise Specifications
- Mathematical Validation of Implementations
- Accurate Documentation of a Language

Language Definition

$$L = \{\text{Set of all strings in language L}\}$$

- A language is a set of strings.
- Most interesting languages are infinite in length.
- The strings within a language must be identifiable. That is, given a string *s*, the following most hold:

$$\forall s,\ s \in L \vee s \notin L$$

Formal Grammar

$$L = \{s \in \Sigma^* : s \text{ is generated by G}\}$$

$$s \in L \iff s \text{ is generated by G}$$

- A **formal grammar** G is a set of rules which generates the strings in L.
- Σ is the **alphabet** of the language.
- Σ^* is the **Kleene Closure** of the alphabet.
  - Set of all strings of all lengths consisting of the symbols in Σ
  - Example $\{a, b\}^* = \{\emptyset, a, b, aa, ab, ba, bb, aab, aba, \ldots\}$

Example Formal Grammar

Rules in G

1. $s \rightarrow e$
2. $e \rightarrow e + e$
3. $e \rightarrow e - e$
4. $e \rightarrow e * e$
5. $e \rightarrow e/e$
6. $e \rightarrow n$
7. $n \rightarrow nd$
8. $n \rightarrow d$
9. $d \rightarrow 0$
10. $d \rightarrow 1$

Some of the strings generated by G

```
0
01
10
11
100
101
1111111111000001111
1+1
10+11
101+111*11
1+1-1+1-1+1
```

Components of a Grammar

$$G = \langle \Sigma, N, S, P \rangle$$

- Σ — Set of **Terminal Symbols**
- N — Set of **Non-Terminal Symbols**
- S — The **Start Symbol**
- P — Set of **Production Rules**

Full Formal Example

$$G = \langle \Sigma, N, S, P \rangle$$

- $\Sigma = \{+, -, *, /, 0, 1\}$
- $N = \{s, e, n, d\}$
- $S = s$
- $P = \{s \rightarrow e, e \rightarrow e + e, e \rightarrow e - e, e \rightarrow e * e, e \rightarrow e/e, e \rightarrow n, n \rightarrow nd, n \rightarrow d, d \rightarrow 0, d \rightarrow 1\}$

Generating Strings from a Grammar

General algorithm framework:

```
1.)    s = S
2.)    while s contains a non-terminal:
3.)       select a sub-string and matching replacement rule
4.)       replace the sub-string with replacement rule
```

Recursive Random Generation

Python Excerpt

```
def e():
    i = randint(1, 5)
    if i == 1:
        return e() + '+' + e()
    elif i == 2:
        return e() + '-' + e()
    elif i == 3:
        return e() + '*' + e()
    elif i == 4:
        return e() + '/' + e()
    else:
        return n()
```

- Each non-terminal becomes a function.
- Randomly select the rule to expand.
- Call the appropriate non-terminals and concatenate with the terminals.
- NOTE: Usually, we must limit the depth otherwise the strings become too long!

Outline

Classifications of Languages and Recognizers

Language Recognition

A compiler and/or interpreter is a program that essential recognizes a string and then processes it.

- Languages are classified by the types of rules in their grammar.
- This corresponds to the complexity of recognizing the languages.

The Chomsky Hierarchy

- Type-0 **Recursively Enumerable**

$$\gamma \rightarrow \alpha$$

- Type-1 **Context Sensitive**

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

- Type-2 **Context-Free**

$$A \rightarrow \alpha$$

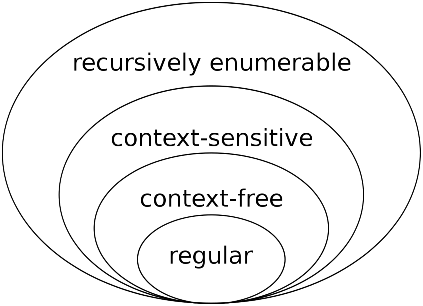
- Type-3 **Regular**

$$A \rightarrow \alpha \text{ and } A \rightarrow aB$$

Definitions

- a — terminal
- A, B — non-terminal
- α, β, γ — string of terminals and/or non-terminals
  - α, β may be empty
  - γ never empty

Container Hierarchy



Chomsky-Hierarchy

Image Source: Wikipedia

Language Recognition Requirements

- Type-0 Recursively Enumerable — Turing Machine
- Type-1 Context Sensitive — Linear Bounded Turing Machine
- Type-2 Context-Free — Non-Deterministic Push-Down Automaton
- Type-3 Regular — Finite State Automaton

Outline

Programming Languages and Formal Grammars

Most programming languages are:

- Predominantly context-free syntax.
- Some context-sensitive elements (though usually not expressed in the grammar.)

Syntax

The **syntax** of a programming language is its grammar. It determines if a string is a well-formed program.

BNF Notation

BNF

Backus-Naur Form is a convenient plain-text formal representation of context-free grammars.

- Non-terminals are denoted in angle brackets: `< Name >`
- Terminals are denoted in quotes: `"0"`
- Arrows are rendered as: `::=`
- Multiple rules are joined using the or symbol: `|`

BNF Example

```
< Start >      ::= < Expression >

< Expression > ::= < Expression > "+" < Expression >
                 | < Expression > "-" < Expression >
                 | < Expression > "*" < Expression >
                 | < Expression > "/" < Expression >
                 | < Number >

< Number >     ::= < Number > < Digit >
                 | < Digit >

< Digit >      ::= "0" | "1"
```

Reading Assignment

- *Three Models for the Description of Language* by Noam Chomsky (chomsky-1956.pdf)
- *On Certain Formal Properties of Grammars* by Noam Chomsky (chomsky-1959.pdf)
- Chapter 3 of your Textbook