Chapter 2

Syntax and Grammars

Defining a Language

Syntax

The rules which govern the form or structure of expressions, statements, and program units

Semantics

The meaning of the expressions, statements, and program units

Language Terminology

Alphabet

A finite set of symbols from which sentences are constructed

Sentence

A string of symbols over some alphabet

Language

A set of properly formed sentences

Metalanguage

A language for defining languages

Often used to define the syntax of sentences in a programming language

Common metalanguages

- Automata
- Regular expressions
- Backus-Naur Form (BNF and EBNF)
- Syntax diagrams

Grammar

A finite collection of rules that defines the set of all sentences in a language

Noam Chomsky's definition (1950s) of **formal grammar**:

1. An *alphabet* (elements are called *terminals*)
2. A set of *nonterminals* (like variables that can represent a class of constructs)
3. A *start symbol* (the initial nonterminal)
4. A set of *productions* (the rules which define the syntax)

An Example Grammar

Alphabet: { 0, 1 }

Nonterminals: { <S> }

Initial nonterminal: <S>

Productions:

{ <S> → 1
<S> → 0
<S> → 1 <S>
<S> → 0 <S> }

The → symbol means "is defined as" or "is replaced by".

Derivations

Production rules can be used to derive sentences in a language

1. Start with the initial nonterminal
2. Replace it with the RHS of a production
3. Continue replacing nonterminals until only terminals remain

Leftmost derivations – Always replace the leftmost nonterminal

Sentence Derivations

Leftmost derivation the following:

1011

01001

R1: <S> → 1
R2: <S> → 0
R3: <S> → 1 <S>
R4: <S> → 0 <S>

Describe the language generated by this grammar.

Getting Formal: Regular Expressions

A regular expression is one of the following:

- A character
- The empty string (denoted by ε)
- Two regular expressions concatenated
- Two regular expressions separated by | (or)
- A regular expression followed by the Kleene star (concatenation of zero or more strings)

Regular Expression Example

Numeric constants:

number → *integer* | *real*
integer → *digit digit**
real → *integer exponent* | *decimal* (*exponent* | ε)*
decimal → *digit** (`.` *digit* | *digit* `.`) *digit**
exponent → (`e` | `E`) (`+` | `-` | ε) *integer*
digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Context-Free Grammars

All productions are of the form $A \rightarrow \gamma$ where A is a nonterminal symbol and γ is a string of terminals and non-terminals

Backus-Naur Form (BNF)

A metalanguage for context-free grammar rules

BNF Examples

Example 1

<*max_3_digit_number*> → <*digit*>
| <*digit*> <*digit*>
| <*digit*> <*digit*> <*digit*>
<*digit*> → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

BNF Examples

Example 2

<*instr_sequence*> → ε | <*instr*> **;** <*instr_sequence*>

<*instr*> → **while (**<*cond*>) <*instr*>
| **if (**<*cond*>) <*instr*>
| {<*instr_sequence*> }
| more instructions…

<*cond*> → defined elsewhere…

BNF Examples

Example 3

<*unsigned_real*> → <*integer*>.<*integer*>

<*integer*> → <*digit*> | <*integer*><*digit*>

<*digit*> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

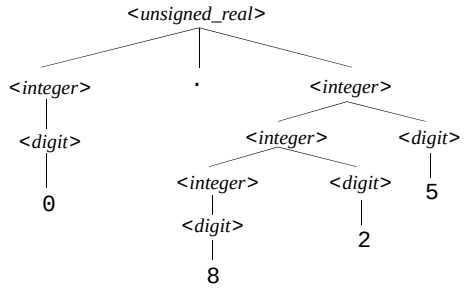
Derive the sentence: **0.825**

Parse Tree

A hierarchical representation of a derivation (aka *derivation tree*)

- Each leaf node is a terminal
- Each internal node is a nonterminal
- Each internal node is the LHS of a production and its children (in left-to-right order) form the RHS of that production
- The root is the starting non-terminal

Parse Tree for 0.825



Ambiguous Grammar

Has at least one sentence with more than one distinct parse tree

Example: The Pascal (and C) "if" statement

<*if_stmt*> → **if** <*cond*> **then** <*stmt*>
| **if** <*cond*> **then**
<*stmt*>
else
<*stmt*>

<*stmt*> → <*if_stmt*> | . . .

Arithmetic Expressions

Version 1

expr → id | number | - *expr* | (*expr*)
| *expr op expr*

op → + | - | * | /

Arithmetic Expressions

Version 2

1. *expr* ⟶ *term* | *expr add_op term*
2. *term* ⟶ *factor* | *term mult_op factor*
3. *factor* ⟶ `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* ⟶ `+` | `-`
5. *mult_op* ⟶ `*` | `/`