Programming Languages Recitation Syntax and Semantics

Chirag Maheshwari
New York University

Email: chirag.m@nyu.edu
Office Hours: Thursday 8-9pm (After Recitation) CIWW 412

Syntax

- The rules governing the organization of symbols in a language. (Format)
- Usually defined by Regular Expressions and Context-Free Grammar.
- The first two phase of compilation consists of scanning syntax using:
  - Lexer – Lexical Analysis
  - Parser – Syntax Analysis

Semantics

- Give meaning to the dead symbols.
- Defines how a given syntax will be computed.
- Static Semantics:
  - Part of the program certain during compilation.
  - Eg: Types, functions arguments, etc.
  - Checks if a program is well-formed
- Dynamic Semantics:
  - Execution of well-formed program.

Features of a good Programming Language

- Unambiguous
- Turing Complete: If the language can express any computable function
  - Are all algorithms implementable in all languages?
  - If it is Turing Complete, it should be able to implement any given algorithm.
- Desirable features:
  - Readability (Some languages are made for obscurity)
  - Concise
  - Mathematical Foundation

Types of Languages

- Declarative
  - Functional: Lisp/Scheme, ML, Haskell
  - Dataflow: Id, Val
  - logic, constraint-based: Prolog, spreadsheets
  - template-based: XSLT
- Imperative
  - von Neumann: C, Ada, Fortran, . . .
  - Scripting: Perl, Python, PHP, . . .
  - object-oriented: Smalltalk, Eiffel, C++, Java, . . .

High Level vs Low Level languages

- Defines the degree of abstraction from the hardware.
- The goal of abstraction is:
  - Write once, run everywhere. (machine independence)
  - Improve readability. (ease of programming)

- machine code → assembly language → C/C++, others → Java, python → SQL

Compiled vs Interpreted

- Compilation converts a source program into a target program (usually machine code).
- Interpreter directly executes the source code.
  - Interpreter still requires lexer, parser, semantic analyser, etc.
- Every languages is finally “interpreted”
- Interpreted languages are easier to debug but are assumed to run slower than compiled languages.

Phases of compiler

- Lexer – lexical analyzer
- Parser – syntax analyzer
- Semantic Analyzer
- Intermediate code generator
- Optimization (based on architecture of the system)
- Target code generation

- Note: Optimization can be done in multiple phases between any given stages.

Formalizing Syntax

- Regular Expressions (RE)
- Context Free Grammar (CFG)

Regular Expressions

- Tokens are the basic building blocks of a program. They are the shortest strings of characters with individual meaning.
- Examples include keywords, identifiers, symbols, constants and numbers.
- In order to specify tokens we use the notation of regular expressions
- Used in the Phase 1 -lexical analysis (Scanner) of the compiler.

Regular Expressions

- Given regular expressions R1 and R2 the following operations can be performed on them:
  - Concatenation: Two regular expressions next to each other. Eg. R1 R2
  - Alternation: Two regular expressions separated by a vertical bar, meaning any string generated by the first one or any string generated by second one. Eg. R1|R2 (OR operation)
  - Kleene Star (*)

Examples

- a - matches the character ‘a’
- ε - matches a null string
- a|b|c - matches ‘a’ or ‘b’ or ‘c’
- abc- matches ‘abc’ (a concatenated with b concatenated with c)
- [a-z] - matches any character between ‘a’ through ‘z’ (Shorthand)
- Alphabet (Uppercase and Lowercase) → [A-Za-z]
- digit → 0|1|2|3|4|5|6|7|8|9
- integer → digit digit*
- number → integer|real
- Identifier → Alphabet (digit|Alphabet)*

Regular Expressions

- Drawbacks:
  - Nesting cannot be expressed in regular expressions which is central to programming languages.
  - For example: Nested parenthesis, palindromes

Context Free Grammar

- More powerful than regular languages/expressions.
- By adding RECURSION, we can define many more sets of strings
- Recognized by parsers.
- Every regular grammar is context free but not every context free grammar is regular.
- Used in the Phase 2 - Syntactic analysis (Parser) of the compiler.

Context Free Grammar

- Consists of
  - Productions (Substitution Rules) : Rules in a CFG of the form

    A ⟶ B
  - Nonterminals: Symbols on the left side of the production (A).
  - Terminals: Symbols that make up the strings derived from grammar. They cannot appear on the left hand side of any production. They represent language's tokens. In the production shown above B is a string of terminals and nonterminals.
  - Start symbol: The nonterminal on the left side of the first production.
- The notation of CFG is sometimes called Backus-Naur form