Lecture Notes 3

Describing Syntax

- Syntax – The form of the language statements and expressions
- Semantics – The meaning of the statements and expressions
- Lexemes – Small syntactic unit (analogous to word in natural language)
- Token – Category of lexeme such as identifier or literal (analogous to part of language such as noun, verb or adjective)
  - Reserved Words / Keywords – Tokens with special meaning
    - Examples: `if`, `while`, `do`, etc.
  - Literals / Constants – A literal numeric or string value
    - Examples: `42`, `"hello world"`, etc.
  - Special Symbols / Operators – Describes operations, or delimits sections
    - Examples: `+`, `/`, `{`, `>=`, etc.
  - Identifiers – Tokens that are like names of variables, functions, etc.
    - Examples: `foo`, `x`, `i`, `Var3`, etc.
    - Predefined Identifiers vs. Keywords – Example `main` vs. `for` in C
  - Principle of Longest Substring – Longest possible substring is collected into single token
    - Example: `doif` is identifier instead of `do` and `if` keywords
  - Token Delimiters / White Space – Principle of Longest Substring requires special tokens to mark end of identifiers
    - Example: Space, newline, operators, etc.
- Language Recognizer – A machine that specifies if a string of characters is in a language or not.
- Language Generator – A machine that generates a valid string in a language.

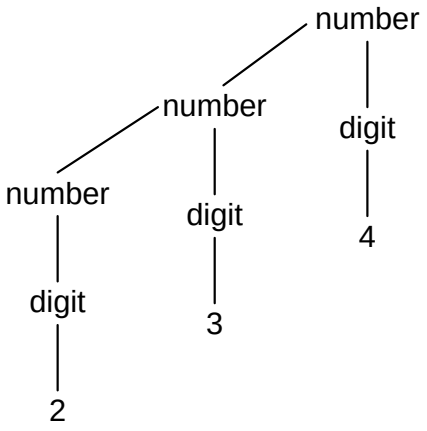
Formal Methods of Describing Syntax

- Regular Language – Language that can be accepted by a Deterministic Finite Automaton (DFA)
- Regular Expressions – Pattern matching expression (often used to describe token rules). Expresses a Regular Language.
  - `|` – Used for selection or choice
    - Example: `a | b` will select a or b
  - `( )` – Used for grouping in regex
    - Example: `(a | b)c` will select a or b and concatenate c
  - `*` – Zero or more repetition of preceding character or group
  - `+` – One or more repetition of preceding character or group
  - `?` – Optional value of preceding character or group
  - `[a-z]` – Marks a range of allowed characters

This content is protected and may not be shared, uploaded, or distributed.

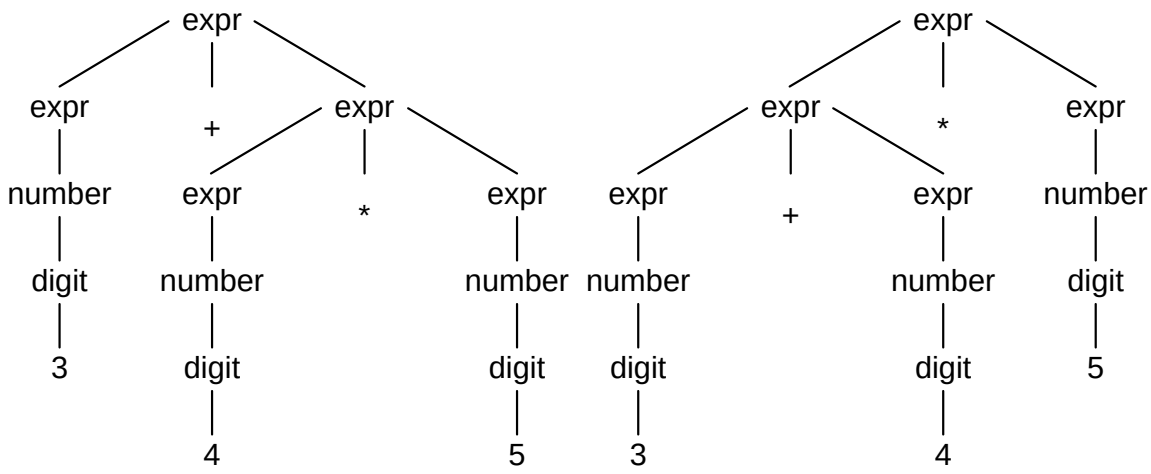
  - \ - Used as an escape character
- Backus-Naur Form (BNF) – Formal method of describing syntax
  - Metalanguage – Language that is used to describe another language
  - Metasymbol – Used as operators in BNF
    - → – Separates left and right hand sides
    - | – Marks selection like in regex (multiple rules can be used in place)
    - **:** = or ::= – Similar to arrow in alternative format
    - <> – Marks the difference between literal and non-terminal
  - Start Symbol – All grammars start with the first left hand side start symbol
  - Derivation – Replacing of left hand side with right hand side rules
  - Terminals – Tokens that cannot be broken up further
  - Non-Terminals – Rules that can be broken up further into other rules
  - Productions – The grammar rules they "produce" strings of the language through derivations
  - Example

    `<expr> → <expr>+<expr> | <expr>*<expr> | (<expr>) | <number>`
    `<number> → <number> <digit> | <digit>`
    `<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`
- Context-Free Grammar – Grammar where rules only have single non-terminal on left hand side, no "context" in which rule can be used.
- Parse Trees and Abstract Syntax Trees
  - Parse Tree – Includes all terminals
  - Abstract Syntax Tree – Only includes non-terminals



- Ambiguity
  - Leftmost Derivation – Left most non-terminal replaced first
  - Disambiguating Rule – Rule to remove ambiguity even with leftmost derivation

This content is protected and may not be shared, uploaded, or distributed.



- Associativity
  - Right or Left Associative
  - Right Associative Example: 2 + 3 + 4 is done as 2 + (3 + 4)
  - Left Associative Example: 2 + 3 + 4 is done as (2 + 3) + 4
- Precedence
  - Right or Left Recursive – Causes right or left associative
- Dangling Else – Problem of ambiguous parsing of nested if-else from rule like:
  `<if_stmt>` → **`if (`** `<logic_expr>` **`)`** `<stmt> |`
  `if (` `<logic_expr>` **`)`** `<stmt>` **`else`** `<stmt>`
- Syntax-directed Semantics – Associating semantics to a syntactic structure
- Extended Backus Naur Form (EBNF)
  - `{}` – Represents zero or more of enclosed
  - `[]` – Represents optional of enclosed

Attribute Grammars

- Attribute Grammar – Describes structure of programming language that cannot be described with Context-Free Grammar
- Static Semantics – Language rules that are difficult or impossible to describe in BNF, but are related to syntax rather than meaning.
  - Example – Variable must be declared before its use
- Attribute Computation Functions (or Semantic Functions) – Specify how attribute values are computed for a given grammar rule.
- Predicate Functions – State the static semantic rules of a programming language
- A(X) – Attributes associated with grammar rule X, consists of S(X) and I(X)
- S(X) – Synthesized attributes, semantic information passed up the parse tree.
- I(X) – Inherited attributes, semantic information passed down or across the parse tree
- Fully Attributed – Parse tree with all attribute values calculated

This content is protected and may not be shared, uploaded, or distributed.

- Intrinsic Attributes – Synthesized attributes of leaf nodes whose values are determined outside the parse tree
- Parse Tree Decoration – Calculation of the attribute values of the parse tree
- Simple Example – Begin/End module names must match:
  - Syntax Rule: `<module_def> → module <mod_name>[1] <module_body> end <mod_name>[2];`
  - Predicate: `<mod_name>[1].string == <mod_name>[2].string`

This content is protected and may not be shared, uploaded, or distributed.