Review of Lecture 3

- Scanning and Parsing
  - Lexical Structure: The structure of tokens (words)
  - Syntactical Structure: The structure of programs

Automatic Tools

regular expression description of the tokens
→ (lex/flex)
scanner of a language

- Example: Figure 4.1 (page 82)

Context-Free Grammar
→ (Yacc/Bison)
parser

- Example: Figure 4.13 (page 112)

Review of Lecture 3

- Context-Free Grammar (Backus-Naur Forms (BNF))
  - Grammar rules (productions):
    - left-hand side: single nonterminal
    - nonterminals (structured names)
    - terminals (tokens)
    - start symbol
  - Language: the set of token strings that can be produced by derivation from the start symbol
  - Derivation:
    - number ⇒ number digit ⇒ number digit digit ⇒ digit
digit digit ⇒ digit digit digit ⇒ 2 3 4

Review of Lecture 3

- Tokens:
  - Reserved words (keywords)
  - Literals/constants
  - Special symbols
  - Identifiers
- Principle of Longest Substring
  - The longest possible string of characters is collected into a single token.
  - The principle requires that tokens are separated by white spaces.

Review of Lecture 3

- Derivations: different derivations for the same syntactic structure.
- Parse trees: capture intrinsic structure, more intuitive, still tedious.
- Abstract Syntax Trees (Syntax Trees): Remove “unnecessary” symbols, meanwhile completely determine syntactic structure.
Topics Today

- Topics:
  - Ambiguities in Grammar
  - Parsing Techniques
- Intuitive analysis and conclusion
- No formal theorems and rigorous proofs
- More details: compilers, automata theory

What is Parsing?

- Given a grammar and a token string:
  - determine if the grammar can generate the token string?
  - i.e., is the string a legal program in the language?
- In other words, to construct a parse tree for the token string.

What’s significant about parse tree?

A parse tree gives a unique syntactic structure

- Leftmost, rightmost derivation
- There is only one leftmost derivation for a parse tree, and symmetrically only one rightmost derivation for a parse tree.

Example

\[
\text{expr} \rightarrow \text{expr} + \text{expr} \mid \text{expr} + \text{expr} \mid (\text{expr}) \mid \text{number} \\
\text{number} \rightarrow \text{number} \mid \text{digit} \\
\text{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

Relationship among language, grammar, parser

- Chomsky Hierarchy
- A language can be described by multiple grammars, while a grammar defines one language.
- A grammar can be parsed by multiple parsers, while a parser accepts one grammar, thus one language.
- Should design a language that allows simple grammar and efficient parser
- For a language, we should construct a grammar that allows fast parsing
- Given a grammar, we should build an efficient parser
Ambiguity

• Ambiguous grammar: There can be multiple parse trees for the same sentence (program)
  — In other words, multiple leftmost derivations.

• Why is it bad?
  — Multiple meaning

• Multiple derivation can be ok, but multiple leftmost derivation is the same as multiple parse tree.

Deal with Ambiguity

• disambiguating rules:
  — Use semantics in determining which parse tree to construct
  or

• unambiguous grammar
  — Rewrite the grammar to avoid ambiguity.

Eliminating Ambiguity for Precedence

• Establish “precedence cascade”: using different structured names for different constructs, adding grammar rules.
  — Higher precedence: lower in cascade

expr → expr + expr | expr * expr | ( expr ) | number

term → term + term | ( expr ) | number

Example of Ambiguity: Precedence

expr → expr + expr | expr * expr | ( expr ) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Two different parse trees for expression 3*4+5

expr → expr + expr | expr * expr | ( expr ) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Example of Ambiguity: Associativity

expr → expr + expr | ( expr ) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Two different parse trees for expression 5-2-1

expr → expr + expr | ( expr ) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

expr.val = 4

expr.val = 2

- is right-associative, which is against common practice in integer arithmetic
- is left-associative, which is what we usually assume
Associativity

• Left-Associative: + * /
• Right-Associative: =

What is meant by a=b=c=1?

Eliminating Ambiguity for Associativity

• left-associativity: left-recursion
  expr → expr + expr | expr * expr | (expr) | number
  expr → expr - term | term
  term → (expr) | number

• right-associativity: right-recursion
  expr → expr= expr | a | b | c

Putting Together

expr → expr + expr | expr * expr | (expr) | number
number → number digit | digit
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

We want to make + left-associative, and * has higher precedence than +

Example of Ambiguity: Dangling-Else

stmt → if(expr) stmt
    | if(expr) stmt else stmt
    | other-stmt
matched_stmt → if(expr) matched_stmt else matched_stmt
    | other-stmt
unmatched_stmt → if(expr) stmt
    | if(expr) matched_stmt else unmatched_stmt

Two different parse trees for “if(expr) if(expr) stmt else stmt”

Eliminating Dangling-Else

stmt → matched_stmt
    | unmatched_stmt
matched_stmt → if(expr) matched_stmt else matched_stmt
    | other-stmt
unmatched_stmt → if(expr) stmt
    | if(expr) matched_stmt else unmatched_stmt