Chapter 4: Syntax Analysis

Principles of Programming Languages

Contents

Non-Regular Language

- Write regular expression for following language

  $L = \{a^n b^n : n \geq 0\}$

Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages

Chomsky Hierarchy

Grammars	Languages	Automaton	Restrictions (w1 → w2)
Type-0	Phrase-structure	Turing machine	w1 = any string with at least 1 non-terminal w2 = any string
Type-1	Context-sensitive	Bounded Turing machine	w1 = any string with at least 1 non-terminal w2 = any string at least as long as w1
Type-2	Context-free	Non-deterministic pushdown automaton	w1 = one non-terminal w2 = any string
Type-3	Regular	Finite state automaton	w1 = one non-terminal w2 = tA or t (t = terminal A = non-terminal)

Backus-Naur Form (BNF)

- Backus-Naur Form (1959)
  - Invented by John Backus to describe ALGOL 58
  - Revised by Peter Naur in ALGOL 60
  - BNF is equivalent to context-free grammars
  - BNF is a *metalanguage* used to describe another language

BNF Fundamentals

- Non-terminals: BNF abstractions
- Terminals: lexemes and tokens
- Grammar: a collection of rules

Rules

- A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of *terminal* and *nonterminal* symbols
- A nonterminal symbol can have more than one RHS

```
stmt → single_stmt
     | "begin" stmt_list "end"
```

Regular vs. Context-Free

- Regular languages are subset of context-free languages
- Languages are generated by grammars
- Regular grammars have form

  $$A \rightarrow \alpha \mid \alpha B$$

- Context-free grammars have form

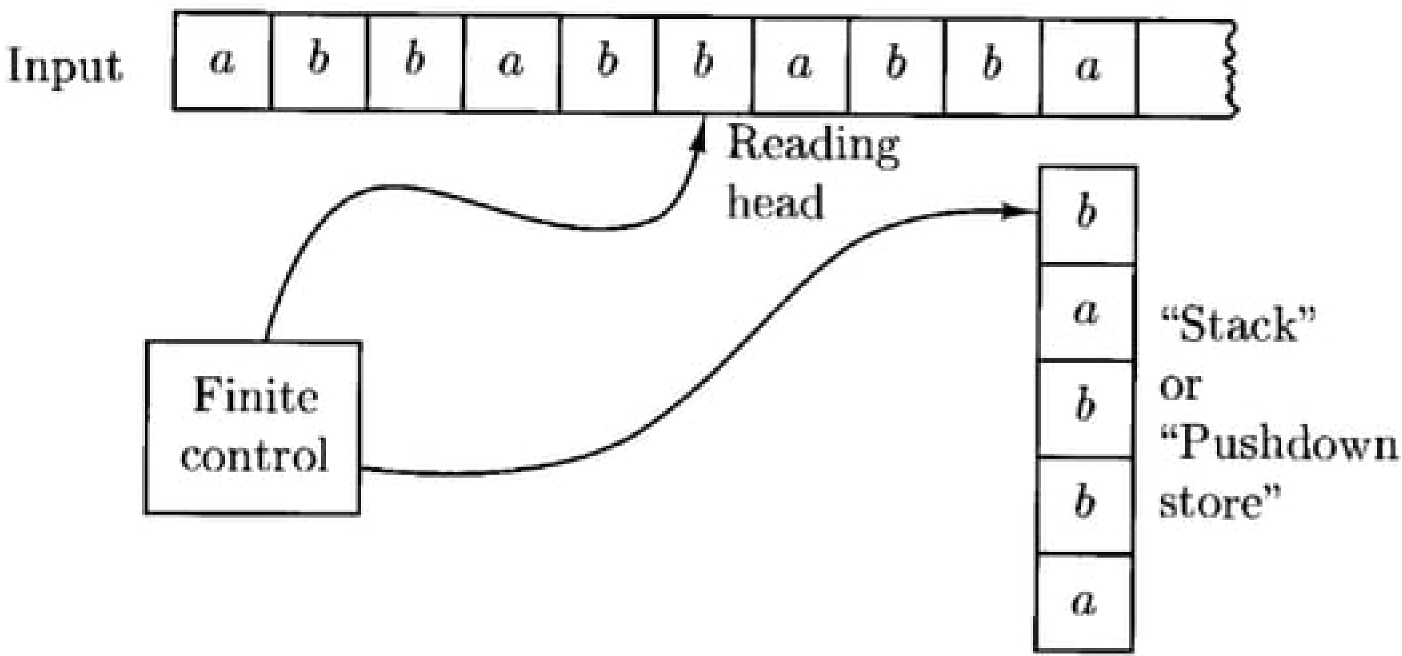
  $$A \rightarrow \alpha B \beta$$

Regular vs. Context-Free

- Context-Free languages are recognized by Nondeterministic Pushdown Automata
- A Pushdown Automaton is a Finite Automaton with "pushdown store", or "stack"

  e.g., $L = \{a^n b^n : n \geq 0\}$

Nondeterministic Pushdown Automata



Describing Lists

- List syntax: `a,b,c,d,…`
- Syntactic lists are described using recursion
- To describe comma-separated list of *IDENT*

```
ident_list → IDENT
           | IDENT "," ident_list
```

Derivation

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

```
program    → stmts
stmts      → stmt | stmt ";" stmts
stmt       → var "=" expr
var        → "a" | "b" | "c" | "d"
expr       → term "+" term | term "-" term
term       → var | CONST
```

An Example Derivation

```
<program> => <stmts>  => <stmt>
                      => <var> = <expr> => a =<expr>
                      => a = <term> + <term>
                      => a = <var> + <term>
                      => a = b + <term>
                      => a = b + CONST
```

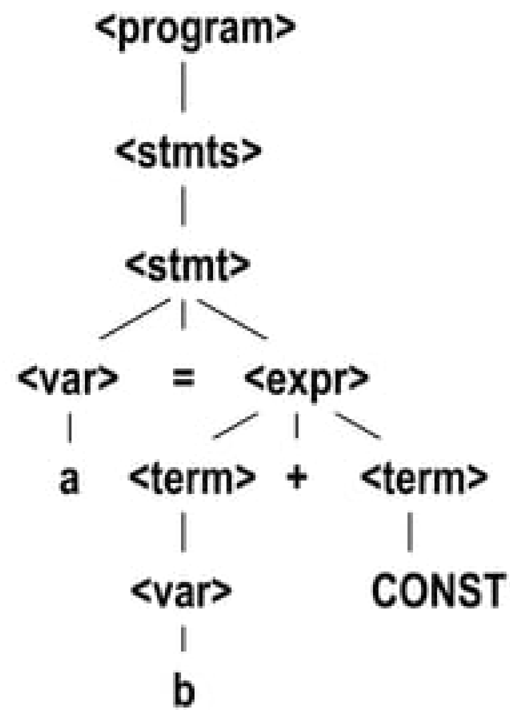
Derivation

- A leftmost (rightmost) derivation is one in which the leftmost (rightmost) nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Parse Tree

- A hierarchical representation of a derivation

```
<program> => <stmts> => <stmt>
=> <var> = <expr>
=> a = <expr>
=> a = <term> + <term>
=> a = <var> + <term>
=> a = b + <term>
=> a = b + CONST
```



Exerices

- Consider the BNF

  S → "(" L ")" | "a"

  L → L "," S | S

  Draw parse trees for the derivation of:

  (a, a)

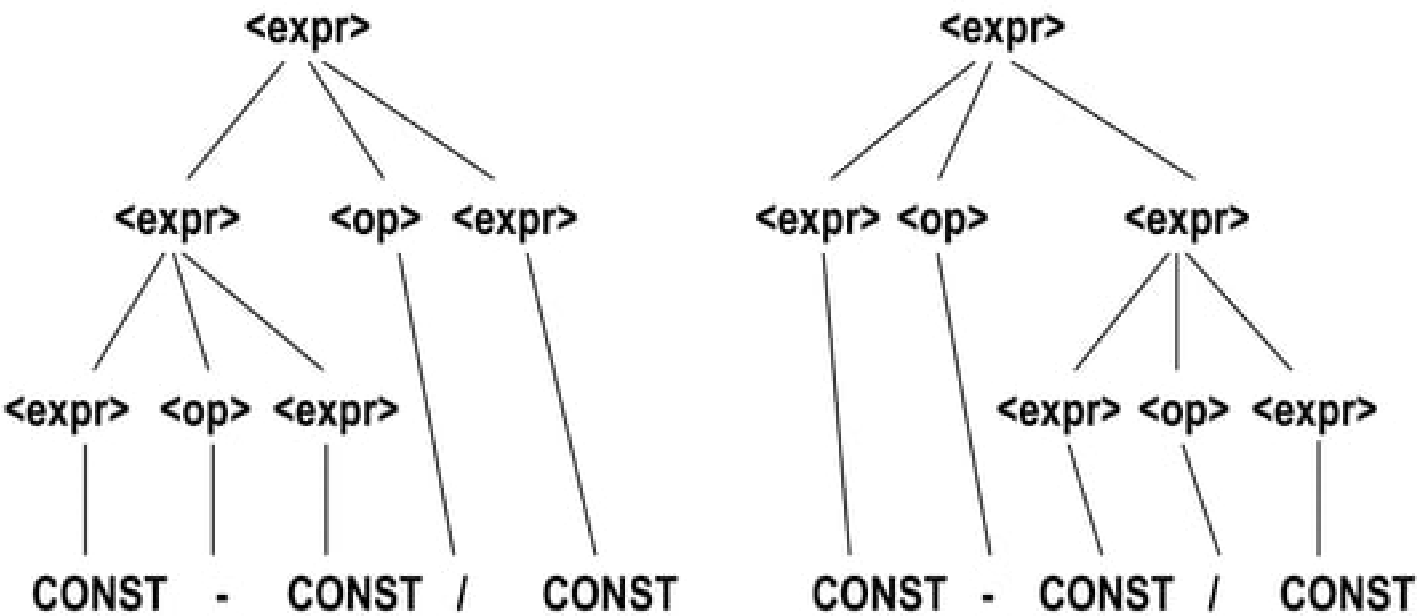
  (a, ((a, a), (a, a)))

Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

```
expr → expr op expr  |  CONST
op   → "/"  |  "-"
```



Exercises

- Is the following grammar ambiguous?

```
A → A "and " A | "not" A | "0" | "1"
```

Exercises

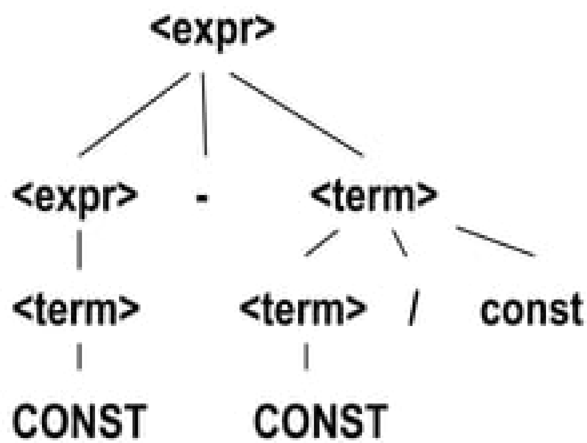
- Write BNF for well-formed parentheses

  ( ), ((())), ~~))((~~, ~~((()~~ …

Precedence of Operators

- Use the parse tree to indicate precedence levels of the operators

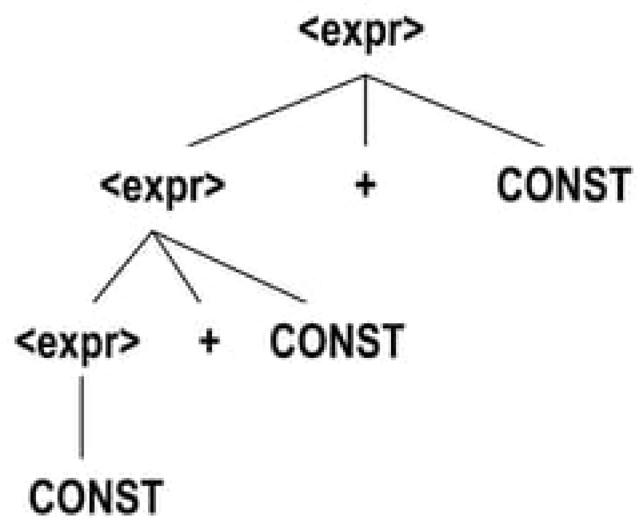
```
expr → expr "-" term | term
term → term "/" CONST| CONST
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
expr → expr "+" expr |  const  (ambiguous)
expr → expr "+" CONST |  CONST  (unambiguous)
```



Exercises

- Rewrite the grammar to fulfill the following requirements:
  - operator “*” takes lower precedence than “+”
  - operator “-” is right-associativity

Extended BNF

- Optional parts are placed in `( )?`

  `proc_call → IDENT ("(" expr_list ")")?`
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

  `term → term ("+"|"-") CONST`
- Repetitions (0 or more) are placed inside braces `( )*`

  `IDENT → letter (letter|digit)*`

BNF and EBNF

- BNF

```
expr → expr "+" term
     | expr "-" term
     | term
term → term "*" factor
     | term "/" factor
     | factor
```

- EBNF

```
expr → term (("+" | "-") term)*
term → factor (("*" | "/") factor)*
```

Exercises

- Write EBNF descriptions for a function header as follow:
- The function declaration begins with a **function** keyword, then the function name, an opening parenthesis '(', a semicolon-separated parameter list, a closing parenthesis ')', a colon ':', a return type, a semi-colon and the body of the function. A function declaration is terminated by a semi-colon ';'.
- The parameter list of a function declaration may contain zero or more parameters. A parameter consists of an identifier, a colon and a parameter type. If two or more consecutive parameters have the same type, they could be reduced to a shorter form: a comma delimited list of these parameter names, followed by a colon ':' and the shared parameter type.
- For example
- **function** area(a: **real**; b: **real**; c: **real**): **real**; could be rewritten as follows
- **function** area(a,b,c: **real**): **real**;
- A return type must be a primitive type. A parameter type could be a primitive type or an array type. The body of a function is also simply a block statement.

Parsing

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message, and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program

Parser Generator

- Parser generator: Produces a parser from a given grammar
- Works in conjunction with lexical analyzer
- Examples: yacc, Antlr, JavaCC
- Programmer specifies actions to be taken during parsing
- No free lunch: need to know parsing theory to build efficient parsers
- Scala has a built-in "combinator parser" (later slides)

Parsing

- Two categories of parsers
  - *Top down* - produce the parse tree, beginning at the root
  - *Bottom up* - produce the parse tree, beginning at the leaves (not mention here)
- Parsers look only one token ahead in the input

The Parsing Problem

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input)
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time ($O(n)$, where n is the length of the input)

Top-down Parsers

- Top-down Parsers
  - Given a sentential form, $xA\alpha$, the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A
- The most common top-down parsing algorithms:
  - Recursive descent - a coded implementation
  - LL parsers - table driven implementation

Recursive-Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing (cont.)

- A grammar for simple expressions:

```
expr → term (("+" | "-") term)*
term → factor (("*" | "/") factor)*
factor → id | "(" expr ")"
```

Recursive-Descent Parsing (cont.)

- Assume we have a lexical analyzer which puts the next token code in a variable
- The coding process when there is only one RHS:
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive-Descent Parsing (cont.)

```
expr → term (("+" | "-") term)*
```

Recursive-Descent Parsing (cont.)

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error

Recursive-Descent Parsing (cont.)

```
factor → ID  |  "(" expr ")"
```

Top-down Problems 1

- The LL Grammar Class
  - The Left Recursion Problem
    - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
      - A grammar can be modified to remove left recursion

Left Recursion Elimination

- For each nonterminal, A,
  1. Group the A-rules as

     $A \rightarrow A\alpha_1\ , | \ldots | A\alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$
  2. Replace the original A-rules with

     $A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$

     $A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | e$

Example

```
E → E "+" T | T
T → T "*" F | F
F → "(" E ")" | id
```

Top-down Problems 2

- Have to be pairwise disjointness
  - Have to determine the correct RHS on the basis of one token of lookahead
  - Def: FIRST(α) = {a | α =>* aβ }
  - For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

  $$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$$

Pairwise Disjointness

- Examples:

  $A \rightarrow a \quad | \quad bB \quad | \quad cAb$

  $A \rightarrow a \quad | \quad aB$

Pairwise Disjointness

- Left factoring can resolve the problem

  Replace

  ```
  variable → IDENTIFIER  |  IDENTIFIER "[" expression "]"
  ```

  with

  ```
  variable → IDENTIFIER new
       new → e   |  "[" expression "]"
  ```

  or

  ```
  variable → IDENTIFIER ("["expression"]")?
  ```

Example

$A \rightarrow a \mid aB$

Building Grammar Techniques

- Repetition
  - [10][10][20]
- Repetition with separators
  - id(arg1, arg2, arg3,…)
- Operator precedence
- Operator associativity
- Avoiding left recursion
- Left factoring
- Disambiguation

Scala Combinator Parser

- Each nonterminal becomes a function
- Terminals (strings) and nonterminals (functions) are combined with operators
  - Sequence ~
  - Alternative |
  - 0 or more `rep(...)`
  - 0 or 1 `opt(...)`

Combinator Parser

```
expr   →  term (("+" | "-") expr)?
term   →  factor (("*" | "/") term)?
factor →  wholeNumber  |  "(" expr ")"

class SimpleLanguageParser extends JavaTokenParsers {
  def expr: Parser[Any] = term ~ opt(("+" | "-") ~ expr)
  def term: Parser[Any] = factor ~ opt(("*" | "/" ) ~ term)
  def factor: Parser[Any] = wholeNumber | "(" ~ expr ~ ")"
}
```

- Will replace `Parser[Any]` with something more useful later

Combinator Parser Results

- String returns itself
- `opt(P)` returns `Option`: Some of the result of P, or None
- `rep(P)` returns `List` of the results of P
- `P ~ Q` returns instance of class ~ (similar to a pair)

Example

```
val parser = new SimpleLanguageParser
val result = parser.parse(parser.expr, "3 - 4 *
  5")
```

sets result to

```
((3~None)~Some((-~((4~Some((*~(5~None)))~None))))
```

- After changing `(x~None)` to x, `Some(y)` to y, and ~ to spaces: `(3 (- (4 (* 5))))`
- Way to do it: wait until labs

Summary

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- Syntax analyzers:
  - Detects syntax errors
  - Produces a parse tree
- A recursive-descent parser is an LL parser
  - EBNF