CS242

Operator Precedence Parsing

Douglas C. Schmidt

Washingon University, St. Louis

The Role of the Parser

- Technically, parsing is the process of determining if a string of tokens can be derived from the start state of a grammar
  - However, languages we wish to recognize in practice are typically not fully describable by conventional grammars
  - Grammars are capable of describing most, but not all, of the syntax of programming languages
- Four Basic Parsing Approaches
  - *Universal parsing methods* – inefficient, but general
  - *Top-down* – generally efficient, useful for hand-coding parsers
  - *Bottom-up* – efficient, automatically generated
  - *Ad-hoc* – eclectic, combined approach

General Types of Parsers:

- *Universal parsing methods*
  - Cocke-Younger-Kasami and Earley's algorithm
- *Top-down*
  - Recursive-descent with backtracking
  - LL – *left-to-right scanning, leftmost derivation*
  - predictive parsing – non-backtracking LL(1) parsing
- *Bottom-up*
  - LR – *left-to-right scanning, reverse rightmost derivation*
  - LALR – *look ahead* LR
- *Ad-hoc*
  - *e.g.*, combine recursive descent with operator-precedence parsing

Context-Free Grammars (CFGs)

- Context-free languages can be described by a "context free grammar" (CFG)
- Four components in a CFG
  1. *Terminals* are tokens
     - *e.g.*, **if**, **then**, **while**, 10.83, foo_bar
  2. *Nonterminals* are syntactic abstractions denoting sets of strings
     - *e.g.*, STATEMENT, EXPRESSION, STATEMENT_LIST
  3. The *start symbol* is a distinguished nonterminal that denotes the set of strings defined by the language
     - *e.g.*, in Pascal the start symbol is *program*
  4. *Productions* are rewriting rules that specify how terminals and nonterminals can be combined to form strings, *e.g.*,:

     stmt → **if** '(' expr ')' stmt | **if** '(' expr ')' stmt **else** stmt

CFG Example

- *Boolean expressions*

  e.g., **true and false or** (**true or false**)

- *Grammar one*

  BEXPR → BEXPR **and** BEXPR
  BEXPR → BEXPR **or** BEXPR
  BEXPR → **true** | **false**
  BEXPR → '(' BEXPR ')'

- *Grammar two*

  BEXPR → OR_EXPR
  OR_EXPR → OR_EXPR **or** AND_EXPR | AND_EXPR
  AND_EXPR → AND_EXPR **and** TOKEN | TOKEN
  TOKEN → **true** | **false** | '(' OR_EXPR ')'

- Note that both grammars accept the same language

Grammar-Related Terms

- *Precedence*
  - Rules for binding operators to operands
  - Higher precedence operators bind to their operands before lower precedence ones
  - *e.g.*, / and * have equal precedence, but are higher than either + or −, which have equal precedence

- *Associativity*
  - Grouping of operands for binary operators of equal precedence
  - Either left-, right-, or non- associative
  - *e.g.*,

    +, -, *, / are left-associative
    = (assignment in C) and ** (exponentiation in Ada)
    are right-associative
    operators **new** and **delete** (free store allocation in C++)
    are non-associative

- *Derivations*
  - Applies rewriting rules to generate strings in a language described by a CFG
  - => means "derives" in one step
    - ▹ *e.g.*, B => B **or** B means B derives B **or** B
  - $\stackrel{*}{=}>$ means derives in zero or more steps
    - ▹ *e.g.*,

      B $\stackrel{*}{=}>$ B
      B $\stackrel{*}{=}>$ B **or** B
      B $\stackrel{*}{=}>$ **true or false**
  - $\stackrel{\pm}{=}>$ means derives in one or more steps
    - ▹ *e.g.*,

      B $\stackrel{\pm}{=}>$ **true or** B
      B $\stackrel{\pm}{=}>$ **true or false**
    - ▹ note that $\stackrel{\pm}{=}>$ defines L(G), the language generated by G

- *Parse trees*
  - Provides a graphical representation of derivations
  - A root (represents the start symbol)
  - Leaves – labeled by terminals
  - Internal nodes – labeled by non-terminals

- *Expression trees*
  - A more compact representation of a parse tree
  - Typically used to depict arithmetic expressions
  - Leaves → operands
  - Internal nodes → operators

- *Sentential form*
  - A string (containing terminals and/or nonterminals) that is derived from the start symbol, *e.g.*, B, B **or** B, B **or false**, **true or false**

- *Sentence*
  - Is a string of terminals derivable from the start symbol

    e.g.,
    true and false is a string in the boolean expr language
    true and or false is a *not*
  - The parser determines whether an input *string* is a sentence in the language being compiled

- *Push-down automata* (PDA)
  - A parser that recognizes a language specified by a CFG simulates a *push-down automata*, *i.e.*, a finite automata with an unbounded stack.

Why Use Context-Free Grammars?

- CFGs provide a precise and relatively comprehensible specification of programming language syntax

- Techniques exist for automatically generating efficient parsers for many grammars

- CFGs enable syntax-directed translation

- They facilitate programming language modifications and extensions

Operator-Precedence Parsing (OPP)

- Uses a restricted form of *shift-reduce* parsing to recognize *operator grammars*:
  - Contain no ϵ productions
  - Have no two adjacent non-terminals

- Table driven approach uses a matrix containing three disjoint precedence relations:

1. $<\cdot$
2. $\doteq$
3. $\cdot>$

Operator-Precedence Parsing (OPP) (cont'd)

- Strengths of OPP
  - Easy to implement by hand or by a simple table-driven generator

- Weaknesses of OPP
  - Need clever lexical analyzer to handle certain overloaded operators *e.g.*, unary $+$ and $-$ versus binary $+$ and $-$
  - Only handles a small class of languages (operator grammars)

OPP Algorithm

- General pseudo-code

```
while (next token is not EOF) {
      if (token is TRUE or FALSE)
            push (handle_stack, mk_node (token));
      else if (f[top (op_stack)] < g[token])
            push (op_stack, mk_node (token));
      else {
            while (f[top (op_stack)] > g[token])
                  push (handle_stack,
                        mk_node (pop (op_stack),
                              pop (handle_stack),
                              pop (handle_stack)));
            if (top (op_stack) == DELIMIT_TOK
                  && token == DELIMIT_TOK)
                  return pop (handle_stack);
            else if (top (op_stack) == LPAREN_TOK
                        && token == RPAREN_TOK) {
                  pop (op_stack);
                  continue;
                  /* Jump over push operation below. */
            }
            push (op_stack, mk_node (token));
      }
}
```