

Programming Languages: Parsing

Onur Tolga Şehitoğlu

Computer Engineering,METU

27 May 2009

Outline

1 Parsing

- Top-down Parsing
- Recursive Descent Parser
- LL Parsers

Parsing

- **input** result of the lexical analysis
- **output** parse tree or intermediate code
- Two types:
 - Top-down
 - Bottom-up

Top-down Parsing

- Start from the starting non-terminal, apply grammar rules to reach the input sentence

$$\begin{aligned}\langle \text{assign} \rangle &\rightarrow a = \langle \text{expr} \rangle \rightarrow a = \langle \text{expr} \rangle + \langle \text{term} \rangle \rightarrow \\&a = \langle \text{term} \rangle + \langle \text{term} \rangle \rightarrow a = \langle \text{fact} \rangle + \langle \text{term} \rangle \rightarrow \\&a = a + \langle \text{term} \rangle \rightarrow a = a + \langle \text{term} \rangle * \langle \text{fact} \rangle \rightarrow \\&a = a + \langle \text{fact} \rangle * \langle \text{fact} \rangle \rightarrow a = a + b * \langle \text{fact} \rangle \rightarrow \\&a = a + b * a \rightarrow\end{aligned}$$

- Simplest form gives leftmost derivation of a grammar processing input from left to right.
- Left recursion in grammar is a problem. Elimination of left recursion needed.
- Deterministic parsing: Look at input symbols to choose next rule to apply.
- recursive descent parsers, LL family parsers are top-down parsers

Recursive Descent Parser

```
typedef enum {ident, number, lparen, rparen, times,
              slash, plus, minus} Symbol;
int accept(Symbol s) { if (sym == s) { next(); return 1; }
                      return 0;
}
void factor(void) {
    if (accept(ident)) ;
    else if (accept(number)) ;
    else if (accept(lparen)) { expression(); expect(rparen);}
    else { error("factor:syntax error at ", currsym); next(); }
}
void term(void) {
    factor();
    while (accept(times) || accept(slash))
        factor();
}
void expression(void) {
    term();
    while (accept(plus) || accept(minus))
        term();
}
```

- Each non-terminal realized as a parsing function
- Parsing functions calls the right handside functions in sequence
- Rule choices are based on the current input symbol. accept checks a terminal and consumes if matches.
- Cannot handle direct or indirect left recursion. A function has to call itself before anything else.
- Hand coded, not flexible.

LL Parsers

- First L is 'left to right input processing', second is 'leftmost derivation'
- Checks next N input symbols to decide on which rule to apply: $\text{LL}(N)$ parsing.
- For example $\text{LL}(1)$ checks the next input symbol only.
- $\text{LL}(N)$ parsing table: A table for $V \times \Sigma^N \mapsto R$
- for expanding a nonterminal $NT \in V$, looking at this table and the next N input symbols, $\text{LL}(N)$ parser chooses the grammar rule $r \in R$ to apply in the next step.

Bottom-up Parsing

- Start from input sentence and merge parts of sentential form matching RHS of a rule into LHS at each step. Try to reach the starting non-terminal. reach the input sentence

$$\begin{aligned} a = a + b * a &\mapsto a = \langle \text{fact} \rangle + b * a \mapsto a = \langle \text{term} \rangle + b * a \mapsto \\ a = \langle \text{expr} \rangle + b * a &\mapsto a = \langle \text{expr} \rangle + \langle \text{fact} \rangle * a \mapsto \\ a = \langle \text{expr} \rangle + \langle \text{term} \rangle * a &\mapsto a = \langle \text{expr} \rangle + \langle \text{term} \rangle * \langle \text{fact} \rangle \mapsto \\ a = \langle \text{expr} \rangle + \langle \text{term} \rangle &\mapsto a = \langle \text{expr} \rangle \mapsto \langle \text{assign} \rangle \mapsto \end{aligned}$$

- Simplest form gives rightmost derivation of a grammar (in reverse) processing input from left to right.
- Shift-reduce parsers are bottom-up:
 - shift:** take a symbol from input and push to stack.
 - reduce:** match and pop a RHS from stack and reduce into LHS.
- Deterministic parsers LALR, SLR(1).