Compiler Design

Lecture-9

Parsing Algorithm

Topics Covered

Designing A Grammar

Parsing Algorithms

Top Down Parsing

Bottom Up Parsing

Designing A Grammar

Concerns:

- Accuracy
- Unambiguity
- Formality
- Readability, Clarity
- Ability to be parsed by a particular algorithm:
  - Top down parser ==> LL(k) Grammar
  - Bottom up Parser ==> LR(k) Grammar
- Ability to be implemented using particular approach
  - By hand
  - By automatic tools

Parsing Algorithms

Given a grammar, want to parse the input programs

- Check legality
- Produce AST representing the structure
- Be efficient

- Kinds of parsing algorithms
  - Top down
  - Bottom up

Top Down Parsing

Build parse tree from the top (start symbol) down to leaves (terminals)

Basic issue:

- when "expanding" a nonterminal with some r.h.s., how to pick which r.h.s.?

E.g.

```
Stmts  ::= Call | Assign | If | While
Call   ::= Id ( Expr {,Expr} )
Assign ::= Id = Expr ;
If     ::= if Test then Stmts end
         | if Test then Stmts else Stmts end
While  ::= while Test do Stmts end
```

Solution: look at input tokens to help decide

Predictive Parser

Predictive parser: top-down parser that can select rhs by looking at most k input tokens (the **lookahead**)

Efficient:

- no backtracking needed
- linear time to parse

Implementation of predictive parsers:

- recursive-descent parser
  - each nonterminal parsed by a procedure
  - call other procedures to parse sub-nonterminals, recursively
  - typically written by hand
- table-driven parser
  - PDA:liketable-driven FSA, plus stack to do recursive FSA calls
  - typically generated by a tool from a grammar specification

LL(k) Grammars

Can construct predictive parser automatically / easily if grammar is LL(k)

- Left-to-right scan of input, Leftmost derivation
- **k** tokens of lookahead needed, ≥ 1

Some restrictions:

- no ambiguity (true for any parsing algorithm)
- no **common prefixes** of length ≥ k:

```
If ::= if Test then Stmts end |
       if Test then Stmts else Stmts
end
```

- no **left recursion**:

```
E  ::= E Op E | ...
```

- a few others

Restrictions guarantee that, given k input tokens, can always select correct rhs to expand nonterminal Easy to do by hand in recursive-descent parser

Eliminating common prefixes

Can **left factor** common prefixes to eliminate them

- create new nonterminal for different suffixes
- delay choice till after common prefix

- Before:

```
If ::= if Test then Stmts end |
       if Test then Stmts else Stmts
  end
```

- After:

```
If     ::= if Test then Stmts IfCont
IfCont ::= end | else Stmts end
```

Eliminating Left Recursion

- Can Rewrite the grammar to eliminate left recursion
- Before

```
E ::= E + T | T
T ::= T * F | F
F ::= id | ...
```

- After

```
E    ::= T ECon
ECon ::= + T ECon | e
T    ::= F TCon
TCon ::= * F TCon | e
F    ::= id | …
```

Bottom Up Parsing

Construct parse tree for input from leaves up

- reducing a string of tokens to single start symbol (inverse of deriving a string of tokens from start symbol)

"Shift-reduce" strategy:

- read ("shift") tokens until seen r.h.s. of "correct" production
- reduce handle to l.h.s. nonterminal, then continue
- done when all input read and reduced to start nonterminal

LR(k)

- LR(k) parsing
  - **L**eft-to-right scan of input, **R**ightmost derivation
  - ***k*** tokens of lookahead

- Strictly more general than LL(*k*)
  - Gets to look at whole rhs of production before deciding what to do, not just first k tokens of rhs
  - can handle left recursion and common prefixes fine
  - Still as efficient as any top-down or bottom-up parsing method
- Complex to implement
  - need automatic tools to construct parser from grammar

LR Parsing Tables

Construct parsing tables implementing a FSA with a stack

- rows: states of parser
- columns: token(s) of lookahead
- entries: action of parser
  - shift, goto state X
  - reduce production "`X ::= RHS`"
  - accept
  - error

Algorithm to construct FSA similar to algorithm to build DFA from NFA

- each state represents set of possible places in parsing

LR(k) algorithm builds huge tables

LALR-Look Ahead LR

LALR(*k*) algorithm has fewer states ==> smaller tables

- less general than LR(*k*), but still good in practice
- size of tables acceptable in practice

- k == 1 in practice
  - most parser generators, including `yacc` and `jflex`, are LALR(1)

Global Plan for LR(0) Parsing

- Goal: Set up the tables for parsing an LR(0) grammar
  - Add S’ --> S$ to the grammar, i.e. solve the problem for a new grammar with terminator
  - Compute parser states by starting with state 1 containing added production, S’ --> .S$
  - Form closures of states and shifting to complete diagram
  - Convert diagram to transition table for PDA
  - Step through parse using table and stack

LR(0) Parser Generation

Example grammar:

```
S' ::= S $          // always add this production
S ::= beep | { L }
L ::= S | L ; S
```

- Key idea: simulate where input might be in grammar as it reads tokens
- "Where input might be in grammar" captured by set of items, which forms a state in the parser's FSA
  - LR(0) item: `lhs ::= rhs` production, with dot in rhs somewhere marking what's been read (shifted) so far
    - LR(k) item: also add *k* tokens of lookahead to each item
  - Initial item: `S' ::= . S $`

Closure

Initial state is **closure** of initial item

- closure: if dot before non-terminal, add all productions for non-terminal with dot at the start
  - "epsilon transitions"

Initial state (1):

```
S’::= . S $
S ::= . beep
S ::= . { L }
```