Parsing: continued

David Notkin
Autumn 2008

Parsing Algorithms

- Earley's algorithm (1970) works for all CFGs
  - $O(N^3)$ worst case performance – $O(N^2)$ for unambiguous grammars
  - Based on dynamic programming, used primarily for computational linguistics
- Different parsing algorithms generally place various restrictions on the grammar of the language to be parsed

- Top-down
- Bottom-up
- Recursive descent
- LL
- LR
- LALR
- SLR
- CYK
- GLR
- Simple precedence parser
- Bounded context
- …

- ACM digital library returned 5600+ articles matching "parsing algorithm"
- Google Scholar almost 34,000

Top Down Parsing

- Build parse tree from the top (start symbol) down to leaves (terminals)
- Basic issue: when expanding a nonterminal, which right hand side should be selected?
- Solution: look at input tokens to decide

```
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
```


Predictive Parser

- Predictive parser: top-down parser that uses at most the next k tokens to select production (the *lookahead*)
- Efficient: no backtracking needed, linear time to parse
- Implementations (analogous to lexing)
  - recursive-descent parser
    - each nonterminal parsed by a procedure
    - call other procedures to parse sub-nonterminals, recursively
    - typically written by hand
  - table-driven parser
    - push-down automata: essentially a table-driven FSA, plus stack to do recursive calls
    - typically generated by a tool from a grammar specification

LL(k) Grammars

LL (k)
Find Left derivation

- Can construct predictive parser automatically and easily if grammar is *LL(k)*
  - Left-to-right scan of input, finds leftmost derivation
  - k tokens of look ahead needed
- Some restrictions including
  - no ambiguity
  - no common prefixes of length ≥ k:

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

  - no left recursion (e.g., `E ::= E Op E | ...`)
- Restrictions guarantee that, given k input tokens, can always select correct right hand side to expand nonterminal.

What if there are common prefixes?

- Left factor common prefixes to eliminate them
  - create new nonterminal for different suffixes
  - delay choice until 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
```


Left recursion? Rewrite…

Before

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

After

```
E    ::= T ECon
ECon ::= + T ECon | ε
T    ::= F TCon
TCon ::= * F TCon | ε
F    ::= id | ...
```

- May not be as clear; can sugar it

```
E ::= T { + T }
T ::= F { * F }
F ::= id | ( E ) | …
```

- Greater distance from concrete syntax to abstract syntax

Table-driven predictive parser

- Automatically compute PREDICT table from grammar
- PREDICT(nonterminal,input-token) => right hand side

Compute PREDICT table

- Compute FIRST set for each right hand side
  - All tokens that can appear first in a derivation from that right hand side
- In case right hand side can be empty
  - Compute FOLLOW set for each non-terminal
    - All tokens that can appear immediately after that non-terminal in a derivation
- Compute FIRST and FOLLOW sets mutually recursively
- PREDICT then depends on the FIRST set

Example for you to do: if you want

	FIRST	FOLLOW
S ::= **if** E **then** S **else** S		
| **while** E **do** S		
| **begin** Ss **end**		
Ss ::= S ; Ss		
| ε		
E ::= **id**		

PREDICT and LL(1)

- If PREDICT table has at most one entry per cell
  - Then the grammar is LL(1)
  - There is always exactly one right choice
    - So it's fast to parse and easy to implement
- If multiple entries in each cell
  - Ex: common prefixes, left recursion, ambiguity
  - Can rewrite grammar (sometimes)
  - Can patch table manually, if you "know" what to do
  - Or can use more powerful parsing technique

Top down implementation

- For years the 401 compiler was a top-down predictive parser, implemented by a method for each nonterminal
  - We have shifted to a bottom-up, automatically generated parser
  - But if you're going to build a simple one, this is usually best
- Examples from http://en.wikibooks.org/wiki/Compiler_construction
  - Helper functions on right

```
int accept(Symbol s) {
   if (sym == s) {
       getsym();
       return 1;
   }
   return 0;
}

int expect(Symbol s) {
   if (accept(s))
       return 1;
   error("expect: unexpected
  symbol");
   return 0;
}
```

Example method

```
void factor(void) {
   if (accept(ident)) {
       ;
   } else if (accept(number)) {
       ;
   } else if (accept(lparen)) {
       expression();
       expect(rparen);
   } else {
       error("factor: syntax error");
       getsym();
   }
}
```


Example method

```
void statement(void) {
   if (accept(ident)) {
       expect(becomes);
       expression();
    …
   } else if (accept(ifsym)) {
       condition();
       expect(thensym);
       statement();
   } else if (accept(whilesym)) {
       condition();
       expect(dosym);
       statement();
   }
}
```


Bottom up parsing

- Construct parse tree for input from leaves up
  - reducing a string of tokens to single start symbol by inverting productions
- Bottom-up parsing is more general than top-down parsing and just as efficient – generally preferred in practice

int * int + int	`T ::= int`
int * T + int	`T ::= int * T`
T + int	`T ::= int`
T + T	`E ::= T`
T + E	`E ::= T + E`
E	

Read the productions found by bottom-up parse bottom to top; this is a rightmost derivation!

"Shift-reduce" strategy

- read ("shift") tokens until the right hand side of "correct" production has been seen
- reduce handle to nonterminal, then continue
- done when all input read and reduced to start nonterminal

```
xyzabcdef
        ^
```

A ::= **bc**.D

LR(k)

- LR(k) parsing
  - Left-to-right scan of input, rightmost derivation
  - k tokens of look ahead
- Strictly more general than LL(k)
  - Gets to look at whole right hand side of production before deciding what to do, not just first k tokens
  - Can handle left recursion and common prefixes
  - As efficient as any top-down parsing
- Complex to implement
  - Generally 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 may build huge tables

Questions?

Ada language/compiler color

- US DoD wanted (roughly) a single, high-level programming language
- They wrote requirements for this language and received 14 bids (1977)
- Four semi-finalists (1978): green (Cii), red for (Intermetrics), blue (SofTech), yellow for (SRI)
- Two finalists: green and red – requirements finalized as Steelman document

General syntax: examples from Steelman

- 2A. Character Set. The full set of character graphics that may be used in source problems shall be given in the language definition. Every source program shall also have a representation that uses only the following 55 character subset of the ASCII graphics: …
- 2B. Grammar. The language should have a simple, uniform, and easily parsed grammar and lexical structure. The language shall have free form syntax and should use familiar notations where such use does not conflict with other goals.
- 2D. Other Syntactic Issues. Multiple occurrences of a language defined symbol appearing in the same context shall not have essentially different meanings. …
- 2E. Mnemonic identifiers. Mnemonically significant identifiers shall be allowed. There shall be a break character for use within identifiers. The language and its translators shall not permit identifiers or reserved words to be abbreviated. …
- 2G. Numeric Literals. There shall be built-in decimal literals. There shall be no implicit truncation or rounding of integer and fixed point literals

York Ada compiler (c. 1986)

"Facts and Figures About the York Ada Compiler" (Wand et al.)

- Written in C
- About 80 KLOC for compiler
  - Front-end about 57 KLOC, code gen about 20 KLOC, VAX-specific code gen about 3 KLOC
- 7 KLOC for run-time
- "It is difficult to make an accurate estimate of the time taken to write the compiler because the compiler writers had other demands on their time (completing PhDs, teaching, etc.) . Fourteen individuals have been involved at various times during the project and have contributed approximately 20 man years to the design and construction of the software . The money spent directly to support the construction of the compiler was [approximately $340k], however this included neither the salaries of four members of the project nor the cost of computer time (we used approximately 30% of a VAX-11/780 over a five year period)."