CSE401: Parsing (B)

David Notkin
Autumn 2000

Objectives: today

- Issues in designing a grammar
- AST extensions for the 401 project
- Overview of parsing algorithms
- Motivation and details of top-down, predictive parsers
- Recursive descent parsing
- Today++: a walk through the PL/0 parser

Next week

- Monday: no class
- Wednesday: Mark S. on bottom-up parsing
- Friday: TBA
  - Maybe a lecture
  - Maybe a tools scavenger hunt
  - Maybe …

Designing a grammar:

on what basis?

- Accuracy
- Readability, clarity
- Unambiguity Were you good today--?
- Limitations of CFGs
- Ability to be parsed by a particular parsing algorithm
  - Top-down parser => LL(k) grammar
  - Bottom-up parser => LR(k) grammar

AST extensions in project (I)

- Expressions
  - `true` and `false` constants
  - array index expression (an lvalue)
  - function call expression
  - `and` and `or` operators
  - tests are expressions
  - constant expressions
- Statements
  - `for`
  - `break`
  - `return`
  - `if` with `else`
- Declarations
  - procedures with result types
  - `var` parameters
- Types
  - `bool`
  - `array`

Parsing algorithms

- Given input (sequence of tokens) and grammar, how do we find an AST that represents the structure of the input with respect to that grammar?
- Two basic kinds of algorithms
  - Top-down: expand from grammar's start symbol until a legal program is produced
  - Bottom-up: create sub-trees that are merged into larger sub-trees, finally leading to the start symbol

Top-down parsing

- Build AST from top (start symbol) to leaves (terminals)
  - Represents a leftmost derivation (e.g., always expand leftmost non-terminal)
- Basic issue: when replacing a non-terminal with a right-hand side (rhs), which rhs should you use?
- Basic solution: “Use the input tokens, Luke!”

```
Stmt  ::= Call | Assign | If
Call  ::= Id
Assign::= Id := Expr
If    ::= if Test then
          Stmts end
If    ::= if Test then
          Stmts else
          Stmts end
```


Predictive parser

- A top-down parser that can select the correct rhs looking at at most *k* input tokens (the *lookahead*)
- Efficient
  - No backtracking is needed
  - Linear time to parse
- Implementation
  - Table-driven: pushdown automaton (PDA) — like table-driven FSA plus stack for recursive FSA calls
  - Recursive-descent parser [used in PL/0]
    - Each non-terminal parsed by a procedure
    - Call other procedures to parse sub-non-terminals, recursively

LL(k), LR(k), …?

- These parsers have generally snazzy names
- The simpler ones look like the ones in the title of this slide
  - The first L means “walk the token sequence left to right”
  - The second letter means “produce a (right|left)most derivation”
    - Leftmost => top-down
    - Rightmost => bottom-up
  - The k means “k tokens of lookahead”
- We won’t discuss LALR(k), SLR, and lots more parsing algorithms

LL(k) grammars

- It’s easy to construct a predictive parser if a grammar is LL(k)
  - **L**eft-to-right scan on input, **L**eftmost derivation, **k** tokens of lookahead
- Restrictions <u>include</u>
  - Unambiguous
  - No common prefixes of length ≥ k
  - No left recursion
- Collectively, the restrictions guarantee that, given k input tokens, one can always select the correct rhs to expand

Common prefix

```
S ::= if Test then
        Stmts end |
      if Test then
        Stmts else
        Stmts end | …
```

Left recursion

```
E ::= E op E | …
```


Eliminating common prefixes

- *Left factor* them, creating new non-terminal for the common prefix and/or different suffixes
- 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
  ```
- Grammar is a big uglier
- Easy to do manually in a recursive-descent parser

Eliminating left recursion:

rewrite the grammar

- Before
  ```
  E ::= E + T | T
  T ::= T * F | F
  F ::= id | ( E ) | …
  ```
- After
  ```
  E     ::= T ECont
  ECont ::= + T ECont | ε
  T     ::= F TCont
  TCont ::= * F TCont | ε
  F ::= id | ( E ) | …
  ```

Just add sugar

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

- Sugared form is still pretty readable
- Concrete syntax tree is not as close to abstract syntax tree
- Easy to implement in hand-written recursive descent parser

Table-driven predictive parser

- Automatically compute PREDICT table from grammar
- PREDICT(nonterminal,input-symbol) => rhs

Example

```
Stmt  ::= 1 if expr then Stmt else Stmt |
          2 while Expr do Stmt |
          3 begin Stmts end
Stmts ::= 4 Stmt ; Stmts | 5 ε
Expr  ::= 6 id
```

	if	then	else	while	do	begin	end	id	;
Stmt	1			2		3			
Stmts	4			4		4	5		
Expr								6	

Construct PREDICT table

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

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

Another example

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

Sugared

```
E   ::= 1 T E′
E′  ::= 2 + T E′ | 3 ε
T   ::= 4 F T′
T′  ::= 5 * F T′ | 6 ε
F   ::= 7 - F | 8 id | 9 ( E )
```

Unsugared

PREDICT

	id	+	-	*	/	(	)
E							
E′							
T							
T′							
F							

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
  - LL(a) => each column labeled by one token
- Can have multiple entries in each cell
  - Ex: common prefixes, left recursion, ambiguity
  - Can patch table manually, if you "know" what to do
  - Or can use more powerful parsing technique