CSCI312 Principles of Programming Languages

LL Parsing

Xu Liu

Derived from Keith Cooper's COMP 412 at Rice University

Quiz

Are these two grammars can be parsed by LL(1) parser?

(1) A->aB | bC | C

B->b

C-> a

(2) A->aAb | Ab | b

Outline

See more general problems in a top down parser

- Backtracking — select appropriate productions
- Left recursion — revise grammars
- Predictive parsing — more than recursive descent
- Table-driven parsing

Remember the expression grammar?

We will call this version "the classic expression grammar"

— from previous lecture

0	Goal	→	Expr
1	Expr	→	Expr + Term
2		|	Expr - Term
3		|	Term
4	Term	→	Term * Factor
5		|	Term / Factor
6		|	Factor
7	Factor	→	(Expr)
8		|	number
9		|	id

And the input <u>x</u> - <u>2</u> * <u>y</u>

Example

Let's try x - 2 * y :

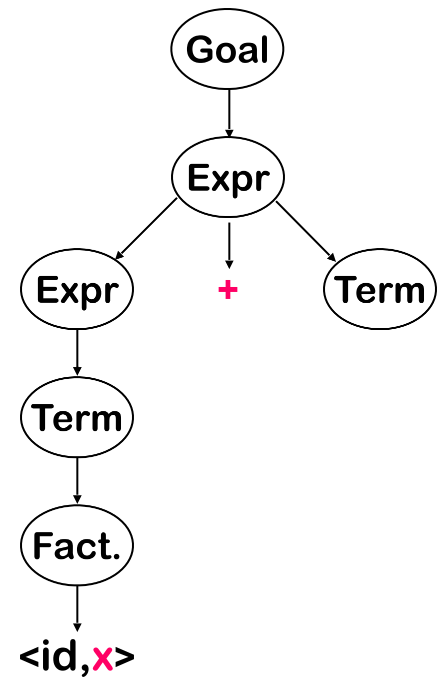
Rule	Sentential Form	Input
—	Goal	↑x - 2 * y

↑ is the position in the input buffer

Example

Let's try x – 2 * y :

Rule	Sentential Form	Input
—	Goal	↑x - 2 * y
0	Expr	↑x - 2 * y
1	Expr +Term	↑x - 2 * y
3	Term +Term	↑x - 2 * y
6	Factor +Term	↑x - 2 * y
9	<id,x> +Term	↑x - 2 * y
→	<id,x> +Term	x ↑- 2 * y



This worked well, except that "–" doesn't match "+"

The parser must backtrack to here

↑ is the position in the input buffer

Why this parser incurs backtracking?

- Select a wrong production
  - multiple choices
  - no hint to select the correct one

0	Goal	→	Expr
1	Expr	→	Expr + Term
2		|	Expr - Term
3		|	Term
4	Term	→	Term * Factor
5		|	Term / Factor
6		|	Factor
7	Factor	→	(Expr)
8		|	number
9		|	id

Left recursion

Other choices for expansion are possible

Rule	Sentential Form	Input
—	Goal	↑x - 2 * y
0	Expr	↑x - 2 * y
1	Expr +Term	↑x - 2 * y
1	Expr + Term +Term	↑x - 2 * y
1	Expr + Term +Term + Term	↑x - 2 * y
1	And so on	↑x - 2 * y

Consumes no input!

This expansion doesn't terminate

- Wrong choice of expansion leads to non-termination
- Non-termination is a bad property for a parser to have
- Parser must make the right choice

Why right recursion works fine?

1. E->T+E | T
2. T->a

Derive: a+a

E->T+E->a+E->a+T+E->a+a+E

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

FIRST sets

For some rhs $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) iff $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

This is almost correct

Building Top-down Parsers for LL(1) Grammars

Given an LL(1) grammar, and its FIRST & FOLLOW sets …

- Emit a routine for each non-terminal
  - Nest of if-then-else statements to check alternate rhs's
  - Each returns true on success and throws an error on false
  - Simple, working (perhaps ugly) code
- This automatically constructs a recursive-descent parser

Improving matters

- Nest of if-then-else statements may be slow
  - Good case statement implementation would be better
- What about a table to encode the options?
  - Interpret the table with a skeleton, as we did in scanning

I don't know of a system that does this …

Building Top-down Parsers

Strategy

- Encode knowledge in a table
- Use a standard "skeleton" parser to interpret the table

Example

- The non-terminal Factor has 3 expansions
  - (Expr) or Identifier or Number
- Table might look like:

0	Goal	→	Expr
1	Expr	→	Term Expr'
2	Expr'	→	+ Term Expr'
3		|	- Term Expr'
4		|	ε
5	Term	→	Factor Term'
6	Term'	→	* Factor Term'
7		|	/ Factor Term'
8		|	ε
9	Factor	→	number
10		|	id
11		|	(Expr)

Terminal Symbols

Non-terminal Symbols

	+	-	*	/	Id.	Num.	(	)	EOF
Factor	—	—	—	—	10	9	11	—	—

Cannot expand Factor into an operator ⇒ error

Expand Factor by rule 9 with input "number"

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T

LL(1) Expression Parsing Table

	+	-	*	/	Id	Num	(	)	EOF
Goal	—	—	—	—	0	0	0	—	—
Expr	—	—	—	—	1	1	1	—	—
Expr'	2	3	—	—	—	—	—	4	4
Term	—	—	—	—	5	5	5	—	—
Term'	8	8	6	7	—	—	—	8	8
Factor	—	—	—	—	10	9	11	—	—

Row we built earlier

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T
- Need an interpreter for the table (skeleton parser)

LL(1) Skeleton Parser

```
word ← NextWord()                 // Initial conditions, including
push EOF onto Stack               // a stack to track local goals
push the start symbol, S, onto Stack
TOS ← top of Stack

loop forever
  if TOS = EOF and word = EOF then
     break & report success   // exit on success

   else if TOS is a terminal then
     if TOS matches word then
        pop Stack              // recognized TOS
        word ← NextWord()
     else report error looking for TOS  // error exit

   else                       // TOS is a non-terminal
     if TABLE[TOS,word] is A→ B1B2…Bk then
        pop Stack                    // get rid of A
        push Bk, Bk-1, …, B1      // in that order
     else break & report error expanding TOS

  TOS ← top of Stack
```

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T
- Need a table-driven interpreter for the table
- Need an algorithm to build the table

Filling in TABLE[X,y], $X \in NT$, $y \in T$

1. entry is the rule $X \rightarrow \beta$, if $y \in FIRST^{+}(X \rightarrow \beta)$

– entry is error if rule 1 does not define

If any entry has more than one rule, G is not LL(1)

We call this algorithm the LL(1) table construction algorithm

LL and LR Parsers