

Recap of LL(1)

Comp 412

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Parsing Techniques



Top-down parsers (LL(1), recursive descent)

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" \Rightarrow may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers (LR(1), operator precedence)

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars



Recursive Descent (Summary)

1. Massage grammar to have $LL(1)$ condition
 - a. Remove left recursion
 - b. Build FIRST (and FOLLOW) sets
 - c. Left factor it, as needed
2. Define a procedure for each non-terminal
 - a. Implement a case for each right-hand side
 - b. Call procedures as needed for non-terminals
3. Add extra code, as needed
 - a. Perform context-sensitive checking
 - b. Build an IR to record the code

Can we automate this process?



Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$\begin{array}{l}
 Fee \rightarrow Fee \alpha \\
 \quad \quad | \beta
 \end{array}$$

where neither α nor β start with Fee

We can rewrite this fragment as

$$\begin{array}{l}
 Fee \rightarrow \beta Fie \\
 Fie \rightarrow \alpha Fie \\
 \quad \quad | \epsilon
 \end{array}$$

where Fie is a new non-terminal

The new grammar defines the same language as the old grammar, using only right recursion.

Added a reference to the empty string



Eliminating Left Recursion

The transformation eliminates immediate left recursion
What about more general, indirect left recursion ?

The general algorithm:

arrange the NTs into some order A_1, A_2, \dots, A_n

for $i \leftarrow 1$ to n

for $s \leftarrow 1$ to $i - 1$

replace each production $A_i \rightarrow A_s \gamma$ with $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$,

where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all the current productions for A_s

eliminate any immediate left recursion on A_i

using the direct transformation

Must start with 1 to ensure that $A_1 \rightarrow A_1 \beta$ is transformed

This assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$),
and no epsilon productions



Remember the expression grammar?

We will call this version "the classic expression grammar"
— from last lecture

0	Goal	\rightarrow	Expr
1	Expr	\rightarrow	Expr + Term
2			Expr - Term
3			Term
4	Term	\rightarrow	Term * Factor
5			Term / Factor
6			Factor
7	Factor	\rightarrow	(Expr)
8			<u>number</u>
9			<u>id</u>

And the input $\underline{x} - \underline{z} * y$



Eliminating Left Recursion

Substituting them back into the grammar yields

0	<i>Goal</i>	\rightarrow	<i>Expr</i>
1	<i>Expr</i>	\rightarrow	<i>Term Expr'</i>
2	<i>Expr'</i>	\rightarrow	$+$ <i>Term Expr'</i>
3		$ $	$-$ <i>Term Expr'</i>
4		$ $	ϵ
5	<i>Term</i>	\rightarrow	<i>Factor Term'</i>
6	<i>Term'</i>	\rightarrow	$*$ <i>Factor Term'</i>
7		$ $	$/$ <i>Factor Term'</i>
8		$ $	ϵ
9	<i>Factor</i>	\rightarrow	$($ <i>Expr</i> $)$
10		$ $	<u>number</u>
11		$ $	<u>id</u>

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.



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

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

The LL(1) Property

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

$$FIRST(\alpha) \cap FIRST(\beta) = \emptyset$$

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

This is almost correct
See the next slide



Predictive Parsing

What about ϵ -productions?

⇒ They complicate the definition of LL(1)

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\epsilon \in \text{FIRST}(\alpha)$, then we need to ensure that $\text{FIRST}(\beta)$ is disjoint from $\text{FOLLOW}(A)$, too, where

$\text{FOLLOW}(A)$ = the set of terminal symbols that can immediately follow A in a sentential form

Define $\text{FIRST}^+(A \rightarrow \alpha)$ as

- $\text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$, if $\epsilon \in \text{FIRST}(\alpha)$
- $\text{FIRST}(\alpha)$, otherwise

Then, a grammar is LL(1) iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

$$\text{FIRST}^+(A \rightarrow \alpha) \cap \text{FIRST}^+(A \rightarrow \beta) = \emptyset$$



Left Factoring a Grammar

Eliminating left recursion does not guarantee an LL(1) grammar

- The right recursive grammar may not be predictive
 - First word in a production may not determine the correct “pick”
 - We can refactor the grammar to improve this situation

Consider a NT A such that $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$, where α is a common prefix for two or more alternative right hand sides.

Replace $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$ with

$$A \rightarrow \alpha L \mid \gamma$$

$$L \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

where L is a new nonterminal symbol

Repeat this process until no two alternatives for a single non-terminal have a common prefix

Left factoring does not always produce an LL(1) grammar.



Left Factoring a Grammar

Left factoring a right recursive grammar has an effect that is similar to left-recursion removal on a left-recursive grammar

Consider a right-recursive expression grammar

0	<i>Goal</i>	\rightarrow	<i>Expr</i>
1	<i>Expr</i>	\rightarrow	<i>Term + Expr</i>
2			<i>Term - Expr</i>
3			<i>Term</i>
4	<i>Term</i>	\rightarrow	<i>Factor * Term</i>
5			<i>Factor / Term</i>
6			<i>Factor</i>
7	<i>Factor</i>	\rightarrow	<u><i>id</i></u>

Expr is not LL(1) because *Term* \Rightarrow^* *id* as its first word, and all 3 alternatives begin with *Term*

Left factoring eliminates a problem with common prefixes in right recursive grammars. That problem can arise in grammars produced by left-recursion elimination, or in hand-crafted right-recursive grammars.



Left Factoring a Grammar

Replace $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$ with
 $A \rightarrow \alpha L \mid \gamma$
 $L \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
 where *L* is a new nonterminal symbol

1	<i>Expr</i>	\rightarrow	<i>Term + Expr</i>		1	<i>Expr</i>	\rightarrow	<i>Term NExpr</i>
2			<i>Term - Expr</i>	becomes	2	<i>NExpr</i>	\rightarrow	<i>+ Expr</i>
3			<i>Term</i>		3			<i>- Expr</i>
					4			ϵ

If we think of *NExpr* as *Expr'*, and apply the same transformation to *Term*, we end up with the same right-recursive, predictive expression grammar that we obtained by removing left recursion from the left-recursive expression grammar.

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*)

Building Top-down Parsers



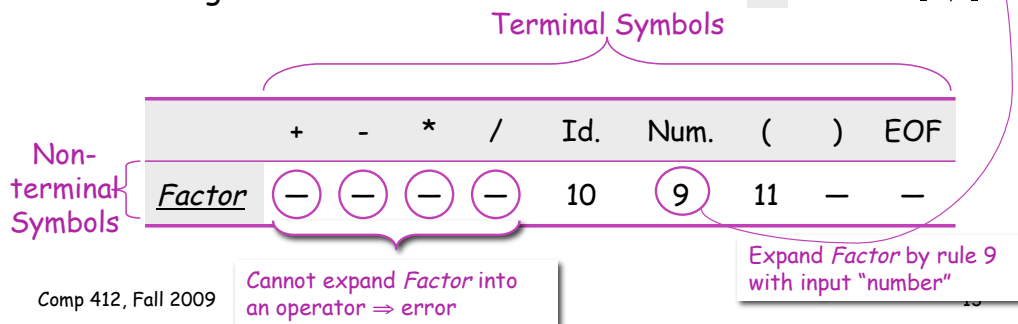
Strategy

- 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 }



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
```

Comp 412, Fall 2009

14

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 $y \in FIRST^+(X \rightarrow \beta)$
2. 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

Comp 412, Fall 2009

15

Grammar and Sets LL(1) Construction



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

Prod'n	FIRST+
0	(, id, num
1	(, id, num
2	+
3	-
4	ε,), eof
5	(, id, num
6	*
7	/
8	ε, +, -,), eof
9	<u>number</u>
10	<u>id</u>
11	(

Right-recursive variant of the classic expression grammar
Comp 412, Fall 2009

Augmented FIRST sets for the grammar

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	-	-

