

CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107S in the Winter 2012/2013 term at the University of Toronto

Copying for purposes other than this use and all forms of distribution are expressly prohibited.

©David B. Wortman, 2002,2003,2004,2008,2009,2010,2012,2013

©Marsha Chechik, 2005,2006,2007

0

Top Down Parsing

- Top down parsers are *predictive* parsers.
Parse stack represents what the parser expects to see. As the parser encounters tokens that it expected to see, the parse stack gets modified to record this fact.
- If the top item in the parser's stack is a non-terminal symbol A then a top-down parser must select one of the rules defining A as its next target.

$$\begin{aligned} A &\rightarrow \alpha_1 \\ &\rightarrow \alpha_2 \\ &\dots \\ &\rightarrow \alpha_n \end{aligned}$$

- Recursive Descent and LL(k) (usually LL(1)) are the two most common top-down parsing techniques.

92

Reading Assignment

Fischer, Cytron, LeBlanc

Chapter 5

Omit Sections 5.8, 5.9

91

Issues for Top Down Parsers

- Grammar rules that have a **common prefix**.

$$A \rightarrow \mathbf{B C D} x Y z$$
$$A \rightarrow \mathbf{B C D} w U v$$

A carefully written recursive descent parser can handle this.

The grammar must be rewritten for LL(k) parsing. See Slide 103

$$A \rightarrow \text{Ahead Arest}$$
$$\text{Ahead} \rightarrow \text{B C D}$$
$$\text{Arest} \rightarrow x Y z \quad | \quad w U v$$

- left recursive** grammar rules

a rule of the form $A \rightarrow \mathbf{A} B C$

would cause a top-down parser to infinitely search for an A .

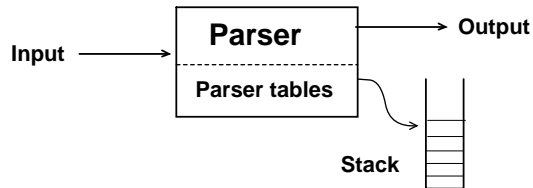
The grammar must be modified to remove all left recursive rules. See Slide

102

93

LL(1) Parsing

- LL(1) is a Top Down parsing technique.
Scans input from the **Left** producing a **Leftmost** derivation
- LL(1) parser is controlled by the **one incoming token** and the **top item** in the parse stack.
- The **parse stack represents what the parser expects to see**. As the parser encounters a token that it expected to see, the parse stack gets modified to record this fact.



94

Leftmost Derivation Example^a

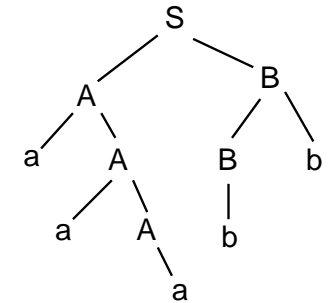
For the grammar:

$S \rightarrow AB$
 $A \rightarrow aA$
 $A \rightarrow a$
 $B \rightarrow Bb$
 $B \rightarrow b$

Leftmost derivation of a a a b b

S
 $\rightarrow AB$
 $\rightarrow aAB$
 $\rightarrow a aAB$
 $\rightarrow a a aB$
 $\rightarrow a a a Bb$
 $\rightarrow a a a b b$

Parse Tree



^aSee Slide 75

95

LL(1) Parsing Example

Grammar

$S \rightarrow dSA$
 $S \rightarrow bAc$
 $A \rightarrow dA$
 $A \rightarrow c$

LL(1) Parse Table

	d	b	c	\$
S	pop S push dSA	pop S push bAc	Error	Error
A	pop A push dA	Error	pop A push c	Error
d	pop d next	Error	Error	Error
b	Error	Pop b next	Error	Error
c	Error	Error	pop c next	Error
✓	Error	Error	Error	Accept

next – advance one token in the input

pop A – pop expected symbol A from parse stack

push B – push B onto the parse stack.

push xYz means push z push Y push x

Input Tokens

d b c c d c \$

Parse Stack	Input	Action
✓ S	d	pop S ; push dSA
✓ A S d	d	pop d ; next
✓ A S	b	pop S ; push bAc
✓ A c A b	b	pop b ; next
✓ A c A	c	pop A ; push c
✓ A c c	c	pop c ; next
✓ A c	c	pop c ; next
✓ A	d	pop d ; pushd dA
✓ A d	d	pop d ; next
✓ A	c	pop A ; push c
✓ c	c	pop c ; next
✓	\$	Accept

96

LL(1) - Predict Sets

- The LL(1) predict sets are the decision mechanism that is used to select among various alternatives for rewriting a nonterminal symbol.

- Define: **Predict set**

Given a nonterminal A with several alternative definitions

$A \rightarrow \alpha_1$

$\rightarrow \alpha_2$

...

$\rightarrow \alpha_n$

The Predict set for rule $A \rightarrow \alpha_i$ is

$Predict(A \rightarrow \alpha_i) = First(\alpha_i)$

α_i not nullable

$Predict(A \rightarrow \alpha_i) = First(\alpha_i) \cup Follow(A)$ α_i is nullable

- For each nonterminal symbol in the grammar, the Predict sets for the definitions of the nonterminal **must be disjoint** for the language to be LL(1).

97

- If a non terminal symbol A is on top of the LL(1) parse stack this means that the parser is trying to find an A . To do this it needs to apply one of the production rules that define A .
- If $inToken$ is the next incoming lexical token, then the parser searches for this token in the Predict sets for the rules that define A
 - if $inToken$ is in $Predict(A \rightarrow \alpha \beta \gamma)$ then the rule $A \rightarrow \alpha \beta \gamma$ should be applied.
 - If $inToken$ is not in any of the Predict sets then a syntax error is detected.
 - $inToken$ cannot occur in more than one Predict set for A in a correctly constructed LL(1) parser.
- Note that the case of A being nullable is automatically taken into account by the construction of the Predict set.

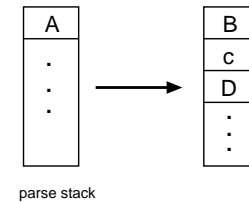
98

Table Driven LL(1) Parsing

- Although the lookup of a terminal symbol in a Predict set can be implemented efficiently using bitsets, most LL(1) parser generators use the Predict sets to build a two dimensional **parse table** that can be efficiently indexed by nonterminal and terminal symbols..
- For each of the rules in the grammar e.g. $A \rightarrow \alpha \beta \gamma$ compute **once** the action required for each of the terminal symbols in $Predict(A \rightarrow \alpha \beta \gamma)$ and cache the result in the parse table.
- The LL(1) table building process
 - Clean up the grammar by removing dead, extraneous and unreachable nonterminal symbols.
 - Replace any left recursive grammar rules.
 - Generate the Predict sets for the grammar.
Fix any Predict set conflicts.
 - Generate the parse table from the grammar and the Predict sets

100

- Given a grammar rule: $A \rightarrow B c D$ and the next incoming symbol is in $Predict(B c D)$ then one Derivation Step would be



- The parser was looking for A now it's looking for B followed by c followed by D

99

Remove Dead, Extraneous and Unreachable Symbols

- Define: **extraneous nonterminals**
Nonterminal symbols in a grammar are extraneous if they are
 - a) *dead* - they do not produce any terminal strings
 - b) *unreachable* - cannot be derived from the goal symbol S .
- Define: **unreachable non-terminal**
The goal symbol S is reachable.
If $A \rightarrow \alpha$ and A is reachable then all nonterminals in α are reachable.
Iterate (transitive closure) until all reachable nonterminals have been detected. Any remaining nonterminals are unreachable.
- Define: **dead nonterminals**
Dead non-terminals never produce a complete terminal string. Example:

$$\begin{array}{ll}
 A \rightarrow a A & B \rightarrow b A \\
 \rightarrow B & \rightarrow A
 \end{array}$$

101

Remove Left Recursion

- LL(1) parsers cannot handle production rules that are left recursive, for example: $A \rightarrow A \alpha$.
- Usually left recursion ($A \rightarrow A \alpha$) can be removed by introducing new non-terminal symbols and factoring the rules so that the revised rules satisfy the LL(1) property:
 - Replace each production $A_i \rightarrow A_j \gamma$ by $A_i \rightarrow \sigma_1 \gamma \mid \dots \mid \sigma_k \gamma$, where $A_j \rightarrow \sigma_1 \mid \sigma_2 \mid \dots \mid \sigma_k$ are all current A_j -productions.
 - Eliminate immediate left recursion among the A_i productions

Example:

$E \rightarrow E + T$	$E \rightarrow T \text{ Etail}$
$\quad \rightarrow T$	$\text{Etail} \rightarrow + T \text{ Etail}$
$T \rightarrow T * P$	$\quad \rightarrow \lambda$
$\quad \rightarrow P$	$T \rightarrow P T \text{ tail}$
$P \rightarrow ID$	$T \text{ tail} \rightarrow * P T \text{ tail}$
	$\quad \rightarrow \lambda$
	$P \rightarrow ID$

102

Fix Predict Set Conflicts

- The Predict sets for each non-terminal in the grammar **must be disjoint** for the grammar to be LL(1).
- Usually non disjoint Predict sets can be fixed by introducing extra non-terminal symbols to give the parser more context. (In effect, locally increasing the amount of lookahead.) Example:

		Predict Set			Predict Set
S	$\rightarrow a B$	{ a }	S	$\rightarrow a E$	{ a }
	$\rightarrow a c a$	{ a }		$\rightarrow d$	{ d }
	$\rightarrow d$	{ d }	E	$\rightarrow b c$	{ b }
B	$\rightarrow b c$	{ b }		$\rightarrow c F$	{ c }
	$\rightarrow c b$	{ c }	F	$\rightarrow b$	{ b }
				$\rightarrow a$	{ a }

103

LL(1) Table Construction Algorithm

- The input set for the input state control of the DPDA (table column indices)^a is the set of terminal symbols plus the end marker $\$$.
The symbol set for the stack top control of the DPDA (table row indices) is
 - the set of *nonterminal symbols*
 - the bottom of stack marker ∇
 - stack symbols* – any terminal symbols that occur in the right hand side of productions in positions other than the extreme left, e.g. c in $A \rightarrow B c D$
- The parser initial stack contents is the $S \nabla$ where S is the goal symbol for the grammar and ∇ is the bottom of stack marker.

Notation:

REPLACE($\alpha \beta$) means replace the top item in the parse stack with

$\alpha \beta$ i.e. Push(β) Push(α)

NEXT means advance the input to the next token.

POP means pop the parse stack

^aFor LL(k) the column indices become k-tuples of input symbols (number of tokens)^k distinct columns

104

LL(1) Table Construction Algorithm

- Construct the DPDA parser table.
 - row ∇ , col $\$$ \leftarrow ACCEPT
 - if terminal symbol c is a stack symbol
row c , col c \leftarrow POP ; NEXT
 - if $A \rightarrow c \beta$ is a production, c is a terminal symbol
row A , col c \leftarrow REPLACE(β) NEXT
 - if $A \rightarrow B \alpha$ is a production
for each b in $Predict(A \rightarrow B \alpha)$
row A , col b \leftarrow REPLACE($B \alpha$)
 - if $A \rightarrow \lambda$ is a production
for each b in $Follow(A)$
row A , col b \leftarrow POP
 - All other entries in the table \leftarrow ERROR

105

LL(1) Table Construction Example

Grammar:

```

1      A → B C c
2          → e D B
3      B → λ
4          → b C D E
5      C → D a B
6          → c a
7      D → λ
8          → d D
9      E → e A f
10         → c
    
```

B and D are *nullable*

106

LL(1) Example - First & Follow Sets^a

• First Sets

$First(A) = \{a, b, c, d, e\}$

$First(B) = \{b\}$

$First(C) = \{a, c, d\}$

$First(D) = \{d\}$

$First(E) = \{c, e\}$

• Follow Sets

$Follow(B) = \{a, c, d, e, f, \$\}$

$Follow(D) = \{a, b, c, e, f, \$\}$

^aSee Slides 87 and 88

107

LL(1) Example - Nontrivial Predict Set Calculations

```

A → BCc
   First(BCc)
   First(B) ∪ First(Cc)      B nullable
   {b} ∪ First(C)           C not nullable
   {b} ∪ {a, c, d}

B → λ
   First(λ) ∪ Follow(B)
   {} ∪ {a, c, d, e, f, $}

C → DaB
   First(D) ∪ First(aB)      D nullable
   {d} ∪ {a}

D → λ
   First(λ) ∪ Follow(D)
   {} ∪ {a, b, c, e, f, $}
    
```

108

LL(1) Example - Predict Sets

```

A → BCc      {a, b, c, d}
   → eDB      {e}
B → λ         {a, c, d, e, f, $}
   → bCDE      {b}
C → DaB       {a, d}
   → ca        {c}
D → λ         {a, b, c, e, f, $}
   → dD        {d}
E → eAf        {e}
   → c         {c}
    
```

109

LL(1) Example - Parse Table

	a	b	c	d	e	f	\$
A	Replace(<i>BCc</i>)	Replace(<i>BCc</i>)	Replace(<i>BCc</i>)	Replace(<i>BCc</i>)	Replace(<i>DB</i>) Next		
B	Pop	Replace(<i>CDE</i>) Next	Pop	Pop	Pop	Pop	
C	Replace(<i>DaB</i>)		Replace(<i>a</i>) Next	Replace(<i>DaB</i>)			
D	Pop	Pop	Pop	Replace(<i>D</i>) Next	Pop	Pop	
E			Pop Next		Replace(<i>Af</i>) Next		
▽							Accept
a	Pop Next						
c			Pop Next				
f						Pop Next	

110

LL(1) Example - Parse b a d e e f c a c \$

Stack	Input	Table	Rule	Action
▽ A	b	A, b	1	Replace(B C c)
▽ c C B	b	B, b	4	Replace(C D E) ; Next
▽ c C E D C	a	C, a	5	Replace(D a B)
▽ c C E D B a D	a	D, a	7	Pop
▽ c C E D B a	a	a, a		Pop ; Next
▽ c C E D B	d	B, d	3	Pop
▽ c C E D	d	D, d	8	Replace(D) ; Next
▽ c C E D	e	D, e	7	Pop
▽ c C E	e	E, e	9	Replace(A f) ; Next
▽ c C f A	e	A, e	2	Replace(D B) ; Next
▽ c C f B D	f	D, f	7	Pop
▽ c C f B	f	B, f	3	Pop
▽ c C f	f	f, f		Pop ; Next
▽ c C	c	C, c	6	Replace(a) ; Next
▽ c a	a	a, a		Pop ; Next
▽ c	c	c, c		Pop ; Next
▽	\$	▽, \$		Accept

111

Example – Expression Grammar for LL(1) Parsing

		Predict Set
expression	→ term moreExpression	{ First(term) }
moreExpression	→ '+' moreExpression	{ + }
	'-' term	{ - }
		{ Follow(expression) }
term	→ factor moreFactor	{ First(factor) }
moreFactor	→ '*' moreFactor	{ * }
	'/' moreFactor	{ / }
		{ Follow(factor) }
factor	→ primary	{ First(primary) }
	'-' primary	{ - }
primary	→ variable	{ First(variable) }
	constant	{ First(constant) }
	'(' expression ')'	{ (}

112

LL(1) Error Detection

- At first invalid input token can generate specific error message from the parse table:
While looking for one of the following *list of terminal symbols* instead *input token* was found.
- Can use information from the parse table to attempt a recovery from a syntax error. See Slides 148 ... 151

113

Automating LL(1) Table Generation

- The *First* and *Follow* sets can be computed manually for small grammars but for larger grammars (i.e. for real programming languages) determining *First* and *Follow* manually is tedious and error prone.
- The *First* and *Follow* sets can be mechanically computed for an arbitrarily large grammar using techniques based on the manipulation of Boolean matrices, e.g. Warshall's algorithm.
- Once these sets have been computed, generation of LL(1) parse tables can be accomplished using the algorithm in Slides 104 and 105.
- There are complete LL(1) parser generators available^a that transform a grammar into LL(1) parsing tables using these techniques.

^aSee for example: [www.antlr.org](http://wwwantlr.org)

114

Expression Grammar for Recursive Descent Parsing

expression	→	term moreExpression
moreExpression	→	'+' term moreExpression
		'-' term moreExpression
term	→	factor moreTerm
moreTerm	→	'*' factor moreTerm
		'/' factor moreTerm
factor	→	primary
		'-' primary
primary	→	variable
		constant
		'(' expression ')'

116

Recursive Descent Parsing

- **Basic Concept:**
Construct a mutually recursive set of functions that act as a parser for the language. Like LL(k) but more flexible.
Typically each function corresponds to one rule in a grammar.
- Usually easy to write, convenient for semantic analysis and code generation.
- Backtracking is possible if each function is written to fail cleanly (i.e. without any side effects) if its recognition fails.
- Can implement k token lookahead *selectively* i.e only where it is necessary to solve a particular problem.
- Recursive descent is a good choice for
 - Languages with difficult or complicated syntax, e.g. Java, Ada, Modula, PL/I
 - Quick and Dirty compilers if a parser generator is unavailable.

115

Recursive Descent Expression Parser

```

expression( ... ) {
    term( ... );
    while ( nextCh == '+' || nextCh == '-' ) {
        getNext( ... );
        term( ... );      /* moreExpression */
    }
}

term( ... ) {
    factor( ... );
    while ( nextCh == '*' || nextCh == '/' ) {
        getNext( ... );
        factor( ... );    /* moreTerm */
    }
}

factor( ... ) {
    if ( nextCh == '-' )
        getNext( ... );    /* unary minus */
    primary( ... );
}

```

nextCh is the next input token, *getNext* advances the input.

117

```

primary( ... ) {
    if variable
        ... /* process variable */
    else if constant
        ... /* process constant */
    else if nextCH == '(' {
        getNext( ... );
        expression( ... ); /* parenthesized expression */
        if nextCh == ')'
            getNext( ... );
        else
            error( "missing ) after expression" );
    }
    else
        error( "ill-formed expression" );
}

```

118

Recursive Descent Parse of $A * - B / (7 - C) \$$

Function Calls	Input
→ expression	A * - B / (7 - C) \$
→ term	A * - B / (7 - C) \$
→ factor	A * - B / (7 - C) \$
→ primary	* - B / (7 - C) \$
→ factor	- B / (7 - C) \$
→ primary	B / (7 - C) \$
→ factor	(7 - C) \$
→ primary	(7 - C) \$
→ expression	7 - C) \$
→ term	7 - C) \$
→ factor	7 - C) \$
→ primary	- C) \$
→ term	C) \$
→ factor	C) \$
→ primary	C) \$
)
	\$

119

Backtracking Example

```

PL/I  DECLARE ( A, B, C, D) FIXED BINARY ; /* Declaration */
      DECLARE ( A, B, C, D) = 23 ; /* Assignment */

ParseDeclaration( ... ) : parseResult
var beforeDeclare : parseState ;
saveParserState( beforeDeclare ) ;
assert( Lookahead( "DECLARE" ) ) ;
advanceInput( ... ) ; /* skip over DECLARE */
if parseDeclarationList( ... ) then
    if Lookahead( "=" ) then /* Assignment !! */
        /* # $ % & * @ keyword languages */
        restoreParserState( beforeDeclare ) ; /* Backtrack */
        return ParseAssignment( ... ) ;
    else
        return parseDeclarationTail( ... ) ;
fi
else /* no list after DECLARE */
    if Lookahead( "=" ) then /* Assignment DECLARE = expn */
        restoreParserState( beforeDeclare ) /* Backtrack */ ;
        return ParseAssignment( ... ) ;
    else
        syntaxError( "Missing List in Declaration" );
        return FAIL ;
    fi
fi
end ParseDeclaration ;

```

120