CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107HS in the Winter 2015/2016 term at the University of Toronto

Reading Assignment

Fischer, Cytron, LeBlanc

Chapter 5

Omit Sections 5.8, 5.9

Top Down Parsing

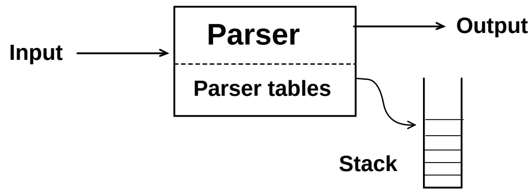
- Top down parsers are *predictive* parsers.
  Parse stack represents what the parser expects to see. As the parser encounters token that it expected to see, the parse stack gets modified to record this fact.
- If the top item in the parsers stack is a non terminal symbol A then a top down parser must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

$$\begin{aligned} A \; & \rightarrow \alpha_1 \\ & \rightarrow \alpha_2 \\ & \quad \ldots \\ & \rightarrow \alpha_n \end{aligned}$$

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

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.



Leftmost Derivation Example[a]

For the grammar:

$$
\begin{array}{lll}
S & \rightarrow & A\ B \\
A & \rightarrow & a\ A \\
 & | & a \\
B & \rightarrow & B\ b \\
 & | & b
\end{array}
$$

Leftmost derivation of a a a b b

[a] See Slide 69

Leftmost Derivation Example

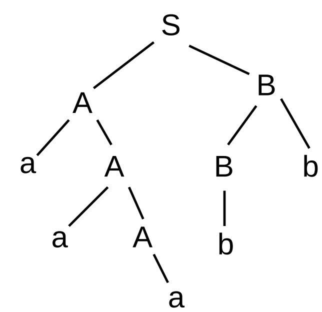
For the grammar:

$$
\begin{array}{lll}
S & \rightarrow & A\ B \\
A & \rightarrow & a\ A \\
 & | & a \\
B & \rightarrow & B\ b \\
 & | & b
\end{array}
$$

Leftmost derivation of a a a b b

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

Parse Tree



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

  $$
  \begin{array}{ll}
  A & \rightarrow \alpha_1 \\
   & \rightarrow \alpha_2 \\
   & \quad \ldots \\
   & \rightarrow \alpha_n
  \end{array}
  $$

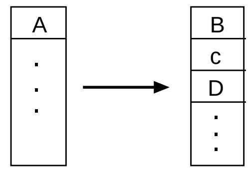
  The Predict set for rule $A \rightarrow \alpha_i$ is

  $Predict(A \rightarrow \alpha_i) = First(\alpha_i)$ $\quad\alpha_i$ not nullable

  $Predict(A \rightarrow \alpha_i) = First(\alpha_i) \cup Follow(A)$ $\quad\alpha_i$ 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).
- LL(1) parsers must make a parsing decision at the beginning of each rule. i.e. select which α_i to continue with.

- 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.

- 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



parse stack

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

LL(1) Parsing Example

Grammar

S $\rightarrow$ d S A
$\rightarrow$ b A c
A $\rightarrow$ d A
$\rightarrow$ c

Input Tokens

d b c c d 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

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

Issues for Top Down Parsers

- Grammar rules that have a **common prefix**.
  A $\rightarrow$ B C D x Y z
  A $\rightarrow$ B C D w U v

  A recursive descent parser can handle this.
  The grammar must be rewritten for LL(k) parsing See Slide 98
  A $\rightarrow$ Ahead Atail
  Ahead $\rightarrow$ B C D
  Atail $\rightarrow$ x Y z
  $\rightarrow$ w U v

- **left recursive** grammar rules
  a rule of the form A $\rightarrow$ 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 97

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

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:

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

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 $A_i \rightarrow \sigma_1\gamma \mid \ldots \mid \sigma_k\gamma$, where $A_j \rightarrow \sigma_1 \mid \sigma_2 \mid \ldots \mid \sigma_k$ are all current A_j-productions.

  – Eliminate immediate left recursion among the A_i productions

  Example:

$$\begin{array}{lll} E & \rightarrow & E + T \\ & \rightarrow & T \\ T & \rightarrow & T * P \\ & \rightarrow & P \\ P & \rightarrow & ID \end{array} \qquad \begin{array}{lll} E & \rightarrow & T\ Etail \\ Etail & \rightarrow & +\,T\ Etail \\ & \rightarrow & \lambda \\ T & \rightarrow & P\ Ttail \\ Ttail & \rightarrow & *\,P\ Ttail \\ & \rightarrow & \lambda \\ P & \rightarrow & ID \end{array}$$

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

LL(1) Table Construction Algorithm

1. 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 $\triangledown$
   - *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 $A \rightarrow B\,c\,D$

2. The parser initial stack contents is the $S\ \triangledown$ where S is the goal symbol for the grammar and $\triangledown$ 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

[a] For LL(k) the column indices become k-tuples of input symbols $(number\ of\ tokens)^k$ distinct columns

LL(1) Table Construction Algorithm

3 Construct the DPDA parser table.

3a row $\triangledown$, col $\$$ $\leftarrow$ ACCEPT % Stack empty , no more input

3b if terminal symbol c is a stack symbol % $A \rightarrow c$ or $A \rightarrow BcD$
row c, col c $\leftarrow$ POP ; NEXT % Expect c, found c

3c if $A \rightarrow c\ \beta$ is a production, c is a terminal symbol % Found start of $A \rightarrow c\ \beta$
row A, col c $\leftarrow$ REPLACE(β) NEXT % Expect β next

3d if $A \rightarrow B\ \alpha$ is a production
for each b in $Predict(A \rightarrow B\ \alpha)$ % Found start of $A \rightarrow B\ \alpha$
row A, col b $\leftarrow$ REPLACE($B\ \alpha$) % Expect $B\ \alpha$ next

3e if $A \rightarrow \lambda$ is a production
for each b in $Follow(A)$ % Found start of $A \rightarrow \lambda$
row A, col b $\leftarrow$ POP % No longer expecting A

3f All other entries in the table $\leftarrow$ ERROR

LL(1) Table Construction Example

Grammar:

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

B and D are *nullable*

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

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

[a]See Slides 80 and 81

LL(1) Example - Nontrivial Predict Set Calculations

A	$\rightarrow$	$B\,C\,c$	
		$First(\ B\,C\,c\ )$	
		$First(\ B\ ) \cup First(\ C\,c\ )$	B nullable
		$\{\ b\ \} \cup First(\ C\ )$	C not nullable
		$\{\ b\ \} \cup \{\ a,\ c,\ d\ \}$	
B	$\rightarrow$	λ	
		$First(\ \lambda\ ) \cup Follow(\ B\ )$	
		$\{\ \} \cup \{\ a,\ c,\ d,\ e,\ f,\ \$\ \}$	
C	$\rightarrow$	$D\,a\,B$	
		$First(\ D\ ) \cup First(\ a\,B\ )$	D nullable
		$\{\ d\ \} \cup \{\ a\ \}$	
D	$\rightarrow$	λ	
		$First(\ \lambda\ ) \cup Follow(\ D\ )$	
		$\{\ \} \cup \{\ a,\ b,\ c,\ e,\ f,\ \$\ \}$	

LL(1) Example - Predict Sets

A	$\rightarrow$	$B\,C\,c$	$\{\, a,\ b,\ c,\ d \,\}$
	$\rightarrow$	$e\,D\,B$	$\{\, e \,\}$
B	$\rightarrow$	λ	$\{\, a,\ c,\ d,\ e,\ f,\ \$ \,\}$
	$\rightarrow$	$b\,C\,D\,E$	$\{\, b \,\}$
C	$\rightarrow$	$D\,a\,B$	$\{\, a,\ d \,\}$
	$\rightarrow$	$c\,a$	$\{\, c \,\}$
D	$\rightarrow$	λ	$\{\, a,\ b,\ c,\ e,\ f,\ \$ \,\}$
	$\rightarrow$	$d\,D$	$\{\, d \,\}$
E	$\rightarrow$	$e\,A\,f$	$\{\, e \,\}$
	$\rightarrow$	c	$\{\, c \,\}$

LL(1) Example - Parse Table

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

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

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 ')'	{ (}

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.

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 99 and 100.
- There are complete LL(1) parser generators available that transform a grammar into LL(1) parsing tables using these techniques.

ANTLR [a]

- ANTLR is a complete scanner/parser generation tool that uses LL(*) parsing. i.e. efficient LL(k) for k $>$ 1
- ANTLR generates scanners and/or parsers in Java and C#
- It can also automatically generate Abstract Syntax Trees and tree parsers to process such trees.
- ANTLR v4 can handle direct left recursion automatically.
- ANTLR has been used in a number of production systems including Twitter query processing, 2 billion queries/day.

[a] `www.antlr.org`

Recursive Descent Parsing

- **Basic Concept:**
  Construct a mutually recursive set of functions that act as a parser for the language. Typically each function corresponds to one rule in a grammar. Recursive Descent parsers can make parsing decisions anywhere in a rule not just at the start. Example: A $\rightarrow$ B C D x Y z A $\rightarrow$ B C D w U v
- 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
    Java (javac) , Ada, Modula, PL/I, C (gcc), C++ (g++) , Fortran
  - Quick and Dirty compilers if a parser generator is unavailable.

Expression Grammar for Recursive Descent Parsing

expression	$\rightarrow$	term moreExpression
moreExpression	$\rightarrow$	'+' term moreExpression
	|	'-' term moreExpression
	|	
term	$\rightarrow$	factor moreTerm
moreTerm	$\rightarrow$	'*' factor moreTerm
	|	'/' factor moreTerm
	|	
factor	$\rightarrow$	primary
	|	'-' primary
primary	$\rightarrow$	variable
	|	constant
	|	'(' expression ')'

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.


```
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" ) ;
}
```


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 ) $
                                     )
                                     $
```

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 ;
```