Parsing Techniques

Fan Long

University of Toronto

UNIVERSITY OF TORONTO

Two Steps to Parse a Program

- **Lexical Analysis:** Chop the program into tokens
  - **Input:** A character stream
  - **Output:** A token stream

- **Syntax Analysis:** Parse the tokens into a parse tree
  - **Input:** A token stream
  - **Output:** A parse tree of the program

Lexical Analysis (Scanning)

- Lexical analysis transforms its input (a stream of *characters*) from one or more source files into a stream of language-specific *lexical tokens*.
- Lexical Analysis Tasks
  - Delete irrelevant information (whitespace, comments)
  - Determine lexical token boundaries in source stream
  - Transmit lexical tokens to next pass.
  - Deal with ill-formed lexical tokens, recover from lexical errors.
  - Process literal constants (optional).
  - Transmit source coordinates (file, line number) to next pass.

Lexical Analysis

- Programming language objects a lexical analyzer must deal with.
  - Identifiers
  - Reserved words or Keywords.
  - Literal constants: integer, real, string, etc.
  - Special characters: “+” “-” “*” “/” “(” “)” etc.
  - Comments
- A programming language *should* be designed so that the lexical analyzer can determine lexical token boundaries easily
- In most languages, lexical tokens are designed to be easily separated based a single left to right scan of the input stream without backup.

Lexical Analysis Example

```
/* Turing program to find prime numbers <= 10,000 */
const n := 10000
var sieve, primes : array 2 .. n  of boolean
var next :  int := 2
for k1 := 2 .. n  /* initialize  */
   sieve[ k1 ] := true    primes[ k1 ] := false
end for
loop
    exit when next > n
    primes[ next ] := true
    for k2 : next .. n by next
        sieve[ k2 ] := false
    end for
    loop   /* find next prime */
        exit when next > n or sieve[ next ]
        next += 1
    end loop
end loop
/*  prime[i]  is true only for prime numbers */
```

Lexical Analysis Example

- Input – A stream of characters

```
/* Turing program to find prime numbers <= 10,000 */ const n := 10000 var
sieve, primes : array 2 .. n of boolean var next : int := 2 for k1 := 2 .. n /* initialize
*/ sieve[ k1 ] := true primes[ k1 ] := false end for loop exit when next > n
primes[ next ] := true for k2 : next .. n by next sieve[ k2 ] := false end for loop /*
find next prime */ exit when next > n or sieve[ next ] next += 1 end loop
end loop /* prime[i] is true only for prime numbers */
```

- Output – A stream of lexical tokens

```
const n := 10000 var sieve, primes : array 2 .. n of boolean var next : int := 2 for k1
:= 2 .. n sieve [ k1 ] := true primes [ k1 ] := false end for loop exit when next > n
primes [ next ] := true for k2 : next .. n by next sieve [ k2 ] := false end for loop exit
when next > n or sieve [ next ] next += 1 end loop end loop
```

Building Lexical Analysis

- Describe the lexical structure of the language using *regular expressions*
- Modern parsing tools like Flex, JFlex, and ANTLR will:
  - Generate a non-deterministic finite automata (NFA) that recognizes the strings (regular sets) specified by the regular expressions.
  - Convert the NFA into a *deterministic finite automata* (DFA).
  - Implement the DFA using a table-driven algorithm.
- You can also implement the DFA algorithm manually.

Regular Expression

- Regular expression notation is a convenient way to precisely describe the lexical tokens in a language.
- The sets of strings defined by regular expressions are *regular sets*. Each regular expression defines a regular set.
- Useful notations: | for alternatives, + and * for repetitions, [] for denoting a set of characters, ( ) for parenthesis
- Characters may be quoted in single quotes ‘ to avoid ambiguity. Mandatory for the meta characters: ’(’ ’)’ ’|’ ’+’ ’*’

ANTLR4 Example for MiniC

- Identifiers: [a-zA-Z][a-zA-Z0-9_]*
- Digits: [0] | ([1-9][0-9]*)
- WhiteSpaceBlock: [\t\r\n]+

Finite Automata

- Finite automata (state machines) can be used to recognize tokens specified by a regular expression.
- A finite automaton (FA) is
  - A finite set of states
  - A set of transitions from one state to another. Transitions depend on input characters from the vocabulary.
  - A distinguished *start* state.
  - A set of final (*accepting*) states.
- Finite automata can be represented graphically using state transition diagrams.
- If the FA has a *unique* transition for every (state, input character) pair then it is a *deterministic* FA (DFA).

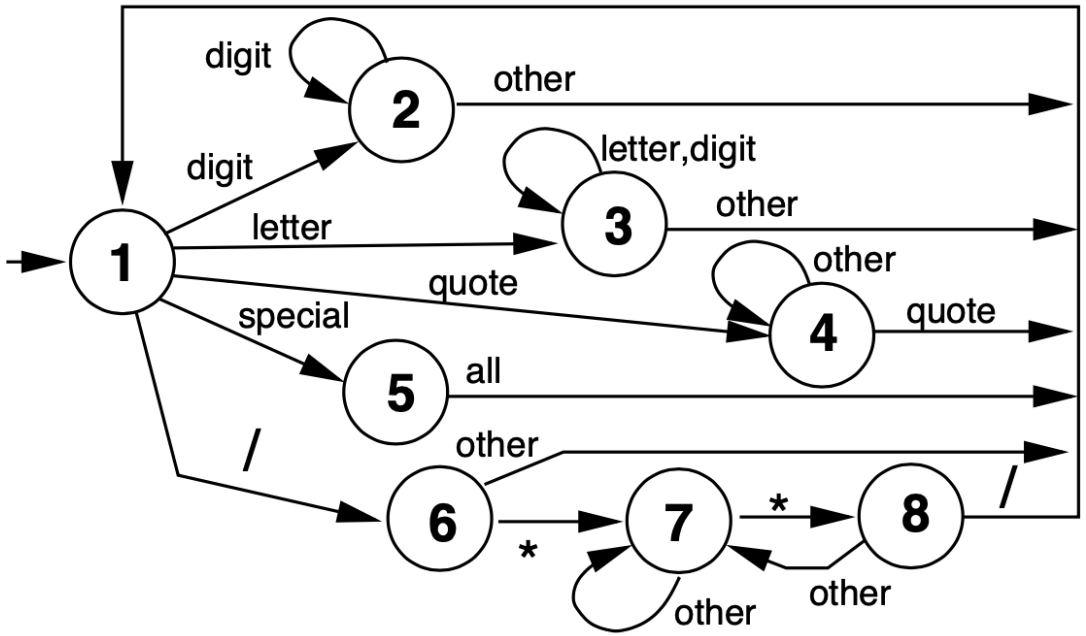
How Lexical Analyzer with DFA works?

- The input to the DFA is the next input character to the lexical analyzer.
- The main state of the DFA uses the character to classify the type of possible token, e.g., identifier, number, special character, whitespace.
- For lexical tokens that can consist of more than one character, the main state transitions to a sub-state that deals with accumulating one type of lexical token, e.g identifier, number.
- Once a complete token has been recognized by the DFA, it is packaged into a lexical token data structure and sent onward to syntax analysis.
- Each sub-state consumes the characters in its lexical token and returns to the main state to deal with the character immediately after its token.

Table Driven DFA Scanner Example

- Language with whitespace, integers, identifiers, strings , special characters, C style comments.
- Make lexical analysis decisions with 1 character look ahead.
- Advance in input, save start of long tokens.
- Save input pointer on entry to each state.
  If appropriate emit accumulated token on return to state 1
- Notation:
  - N – change to state N
  - +N – advance input pointer and change to state N

State	Input Character							Purpose
	blank	digit	letter	quote	Special	/	*	
1	+1	2	3	+4	5	+6	5	blanks & dispatch
2	1	+2	1	1	1	1	1	integers
3	1	+3	+3	1	1	1	1	identifiers
4	+4	+4	+4	+1	+4	+4	+4	'string'
5	1	1	1	1	+1	1	+1	special chars.
6	1	1	1	1	1	1	+7	emit token '/'
7	+7	+7	+7	+7	+7	+7	+8	comment body
8	+7	+7	+7	+7	+7	+1	7	end comment



Lexical Errors

- Lexical analyzer must deal with ill-formed and incomplete tokens.
- Examples:
  - Incorrectly formed: 3.4.5 3.4E- 0x7FFG8
  - Oversize tokens
  - Numeric value out of range: 99999999999999999
  - Unterminated string or comment.
  - Illegal characters
- Error handling strategies
  - Emit error message for lexical error
  - Report source code location

Managing Source Program Coordinates

- The subsequent phases of the compiler need to know where each lexical token originated to be able to emit source code specific error messages: *Syntax Error on line 23 offset 10 of file foo.c*
- Conceptually tag each lexical token with an identification of the source file and line number where it originated.
- Possibly also include the offset in the line.

- Most parsing tools provide API to handle this conveniently.

Syntax Analysis

- The *syntax* of a language defines the sequences of language tokens that form legal sentences in the language.
- Examples:
  - X = Y * Z + W / X — OK
  - X = * Y Z + / W X — Error
  - for (i = 0; i < N; i = i + 1) — OK
  - for (; ;) — OK
  - for (i 0; ;) — Error
- Syntax analysis is the process of analyzing a sequence of lexical tokens to determine if they are a legal sentence in some given language.
- Syntax analysis is based on a formal definition of the source language.

Syntax Analysis Tasks

- Analyze a sequence of lexical tokens to determine its syntactic structure.
- Verify that this syntactic structure corresponds to a legal sentence in the language being processed.
- Transform the token stream into some intermediate form (a parse tree or an abstract syntax tree) that is suitable for subsequent processing.
- Handle syntax errors in the token stream.
- Maintain source program coordinates for use by subsequent passes.

How A Parser Operates

- **Input:** a stream of tokens from lexical analysis
- **Output:** a parse tree (or some equivalent form)
- Information available to the parser:
  - The sequence of tokens that it has already received from lexical analysis.
  - The sequence of steps that it has already taken in attempting to analyze the program.
  - The next *k* lexical tokens in its input stream. (*k*-symbol lookahead).
- Most parsers use a stack to record the first two items.
- For simple languages, parsers process the token stream left to right *without backtracking* and use *k* = 1.

Top Down v.s. Bottom Up Parsing

- **Top Down** parsers start with the start symbol for the language (*S*) and attempt to find a sequence of production rules that transforms the start symbol into a given sequence of input tokens.
  - Conceptually it builds the parse tree from the root to the leaves.
  - A top down parse is called a **derivation**.
  - Recursive Descent and LL(k) are top down parsing techniques.
- **Bottom Up** parsers start with a sequence of input tokens and attempt to find a sequence of production rules that will reduce the entire sequence of terminal symbols to the start symbol.
  - Conceptually it builds the parse tree from the leaves to the root.
  - A bottom up parse is called a **reduction** or a **reverse derivation**.
  - LR(k), SLR(k) and LALR(k) are bottom up parsing techniques.

Terminologies

Σ	A finite set of (terminal) symbols	{a, b, c}
String	A finite sequence of symbols from Σ	abbca
λ	The empty string	
$	End of file on input stream	
Concatenation	X Y	
	X^i means XX…X i times	
	X* zero or more instances of X	
	X+ one or more instances of X	
Language	Any set of strings formed from some vocabulary	

Notations

A, B, S, …	Nonterminal symbols
a, b, c, ...	Terminal symbols
α, β, γ, …	Strings of terminal and non-terminal symbols
N	Set of nonterminal symbols. A, B, S $\in N$
S	The distinguished non-terminal start symbol
α, β, γ, …	Strings in ($N \cup \Sigma$)*

Production Rules

- The most general (context sensitive) form of a production rule is:

  $\gamma A \omega \rightarrow \gamma \alpha \omega$

- For context free grammars, γ, ω are null and all rules are of the form:

  $A \rightarrow \alpha$

- We focus on (mostly) context free grammars for compilers.

Context Free Grammar Example

- For the language definition:

  S → A B

  A → a A

  | a

  B → B b

  | b

- Non terminals: *N* = { A, B, S }
- Terminals: Σ = { a, b}
- Goal Symbols: S
- Productions: P = { S → AB, A → a A, A → a, B → B b, B → b}

Informal Definitions

- The productions in a grammar are a set of rules (in some notation) that define the sequences of terminal symbols that make up legal sentences in the language defined by the grammar.
- A grammar is unambiguous if there is exactly **one sequence of rules** for forming **each** legal sentence in the language. A **language** is ambiguous if **all** grammars for the language are ambiguous.
- A grammar is context-free (CF) if every rule can be applied completely independent of the surrounding context.
- A language is deterministic if it is always possible to determine which rule to apply next when parsing every sentence in the language.

ANTLR4 Grammar Example for MiniC

```
exrp:  '-'  expr
       |  expr ( '*' | '/' ) expr
       |  expr ( '+' | '-' ) expr
       |  var
       |  …
var:    IDENTIFIER  |  IDENTIFIER '[' expr ']'
…
```

Grammar Design for Compiling

- A typical programming language will have two grammars
  - A *reference grammar* that is made available to *users* of the language.
  - A *compiler grammar* that is used by the *implementation* of a language.
  - These two grammars **should** describe exactly the same language.
- The design of the compiler grammar
  - Affects the effort required to implement semantic analysis and code generation.
  - Determines the order in which constructs in the language will be processed (for a given parsing algorithm).
  - Determines the class of parsers that must be used to perform syntax analysis.

Grammar Design for Compiling

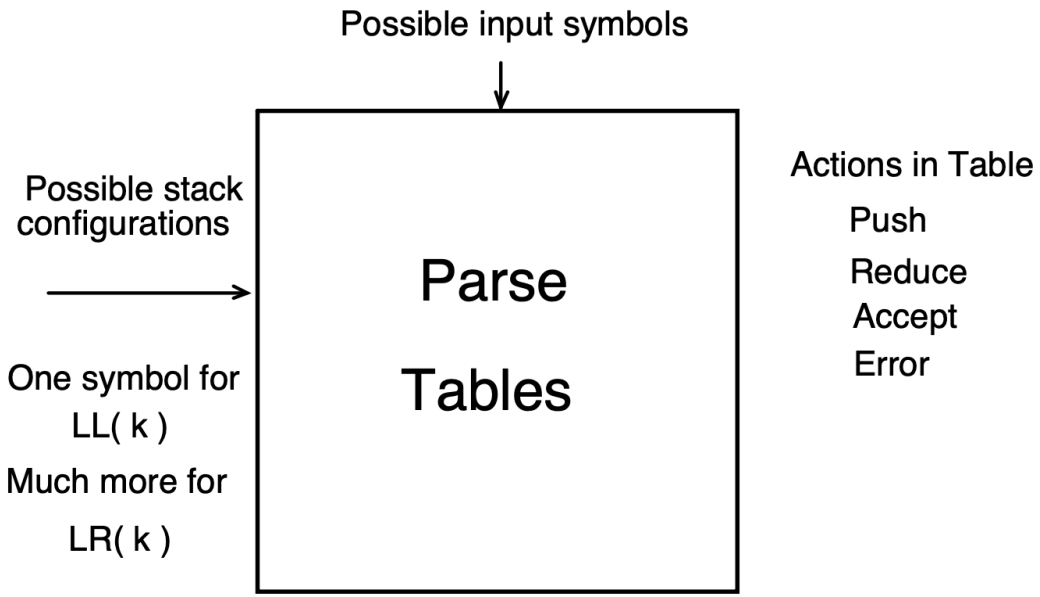
- For LL(*) parsers, the compiler grammar may try to eliminate left-recursion. ANTLR4 can do this automatically in most cases.
- For LALR(1) and SLR(1) parsers semantic and/or code generation actions can only be invoked when a reduction is applied. This means that actions can only occur at the right end of a rule.
- Reference Grammar:

  parameters → parameters ‘,’ parameters | var
- Compiler Grammar:

  parameters → var ACTION_NEW_LIST

  | parameters ‘,’ var ACTION_LIST_APPEND_VAR

Table-Driven Parsing



- Conceptually a table for each parser state. Allow multiple actions per state so tables can be merged to one table for all parser states.
- Final state is accepting or rejecting
- Top Down – stack represents what we expect to see
- Bottom Up – stack represents what we've seen so far

Practical Parsing Techniques

- **Recursive Descent** is a top down parsing technique with the potential for backtracking. Recursive descent parsers are usually written as a collection of interacting recursive functions. Recursive descent is useful for dealing with complicated languages (e.g. C (gcc), C++, Java, Fortran).
- **LL(1)** is a table–driven top down parsing technique.
  An LL(1) parser is a *deterministic push down automaton (DPDA)*
- **LR(1)** is a table–driven bottom up parsing technique.
  The parser uses a Shift/Reduce algorithm to record the state of its reduction. An LR(1) parser is a *deterministic push down automaton (DPDA)*
  SLR(1) and **LALR(1)** are practical versions of LR(1)

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.

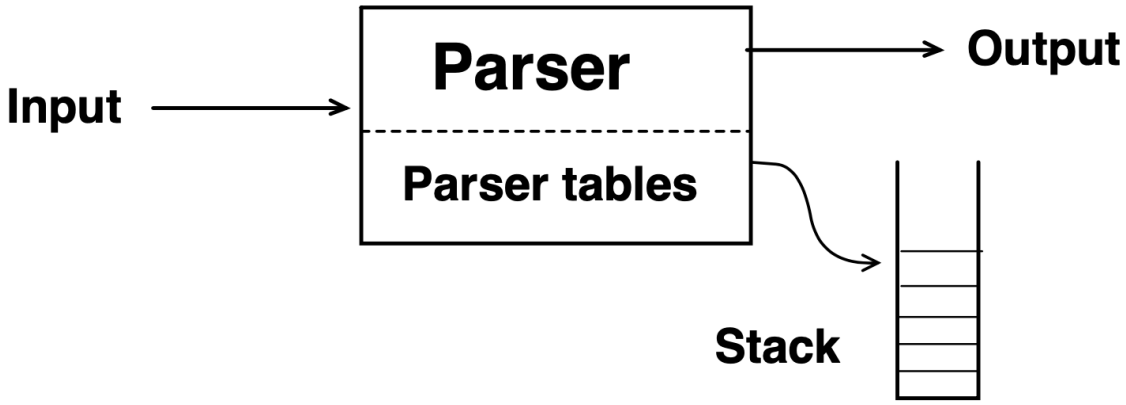
  $A \rightarrow \alpha_1$

  $\quad | \ \alpha_2 \ldots$

- 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

For the grammar:

S → A B

A → a A

| a

B → B b

| b

Leftmost derivation of a a a b b

S

→ A B

→ a A B

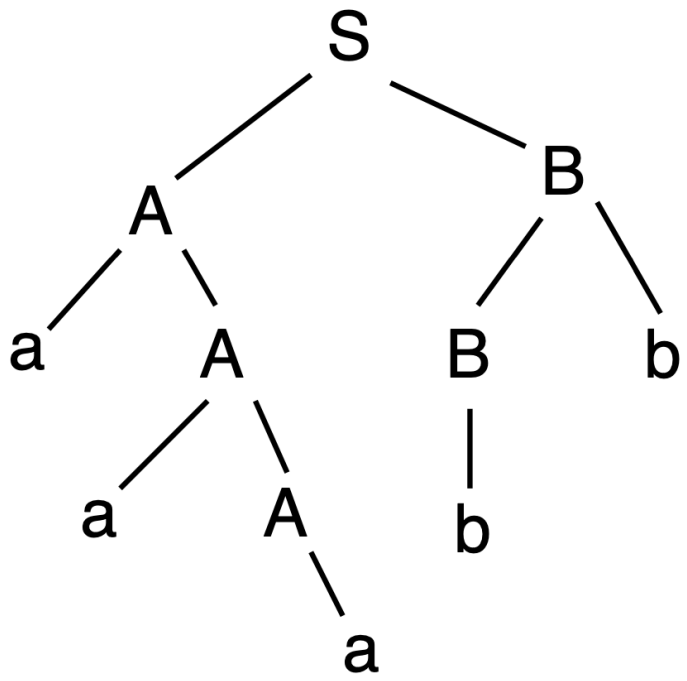
→ a a A B

→ a a a B

→ a a a B b

→ a a a b b

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.
- Predict set: given a nonterminal A with several alternative rules

  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

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

  $Predict(A \rightarrow \alpha_i) = First(\alpha_i)$ if α_i is not nullable

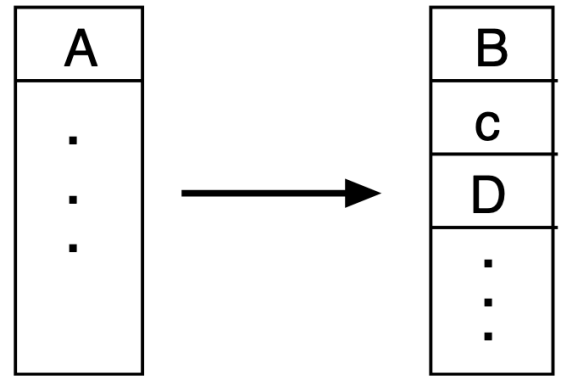
  $Predict(A \rightarrow \alpha_i) = First(\alpha_i) \cup Follow(A)$ α_i is nullable
- For each non-terminal in the grammar, the Predict sets for the definitions of the nonterminal **must be disjoint** for the language to be LL(1).
- LL(1) parsers must make a parsing decision at the beginning of each rule, i.e., select which αi to continue with.

LL(1) - Predict Sets

- 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 → α) then the rule A → α 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.
- Given a grammar rule: A → B c D

  and the next incoming symbol is

  in Predict (B c D) then:



parse stack

Nullability – Deriving the Empty String

- Nullable: α is nullable iff α can derive empty string λ
- Nullability is a significant issue for the construction of parsers. When the string α actually produces λ then it provides **no** information that the parser can use to make a parsing decision.
- A grammar can be processed to determine which non terminal symbols can potentially produce λ.
- Example:

$$A \rightarrow B\ C\ D$$
$$B \rightarrow \lambda$$
$$C \rightarrow B \mid c$$
$$D \rightarrow B\,C \mid d$$
$$A \Rightarrow BCD \Rightarrow CD \Rightarrow BD \Rightarrow D \Rightarrow BC \Rightarrow C \Rightarrow B \Rightarrow \lambda$$

First Sets

- First(α): the set of **terminal symbols** that can occur at the beginning of strings derived from α.
- First Sets can be defined recursively:

First(c β)	= { c }	c is a terminal
First(λ)	= { }	
First(B ɣ)	= First (B)	if B is not nullable
	= First (B) ∪ First (ɣ)	if B is nullable

- For small grammars First sets can often by determined by inspection *as long as nullability is taken into account*.

Follow Sets

- Follow(A): the set of **terminal symbols** that can occur **immediately follow** a non-terminal symbol A.

- For *small* grammars it is possible to compute Follow sets by inspection, **but** this is much harder than computing First sets, since all possible instances of nullability must be accounted for.

- The calculation of Follow sets can be automated, See *Fischer, Cytron, LeBlanc* Figure 4.11

First and Follow Set Example

Grammar:

$$
\begin{aligned}
A &\rightarrow B\ C\ c \\
&\rightarrow e\ D\ B \\
B &\rightarrow \lambda \\
&\rightarrow b\ C\ D\ E \\
C &\rightarrow D\ a\ B \\
&\rightarrow c\ a \\
D &\rightarrow \lambda \\
&\rightarrow d\ D \\
E &\rightarrow e\ A\ f \\
&\rightarrow c
\end{aligned}
$$

B and D are *nullable*

UNIVERSITY OF TORONTO

First Set (by inspection)

A → B C c

A → e D B

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

B → λ

B → b C D E

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

C → D a B

C → c a

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

D → λ

D → d D

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

E → e A f

E → c

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

Follow Set (by inspection)

A → B C c

E → e A f

$Follow(A) = \{ \$ \} \cup \{ f \} = \{ f, \$ \}$

A → B C c

A → e D B

C → D a B

$Follow(B) = First(C) \cup Follow(A) \cup Follow(C) = \{ a, c, d, e, f, \$ \}$

A → B C c

B → b C D E

$Follow(C) = \{ c \} \cup First(D) \cup First(E) = \{ c, d, e \}$

A → e D B

B → b C D E

C → D a B

D → d D

$Follow(D) = First(B) \cup Follow(A) \cup First(E) \cup \{ a \} = \{ a, b, c, e, f, \$ \}$

B → b C D E

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

LL(1) Parsing Example

Grammar

S → d S A
  → b A c
A → d A
  → 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
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

Issues for Top Down Parsers

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

  A → B C D x Y z

  A → B C D w U v

- The grammar must be rewritten for LL(k) parsing:

  A → Ahead Atail

  Ahead → B C D

  Atail → x Y z

  → w U v

- **Left recursive** grammar rule, example: A → 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.

Remove Left Recursion

- Usually left recursion can be removed by introducing new non-terminal symbols and factoring the rules so that the revised rules satisfy LL(1) property:
- Example:

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

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.)

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

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

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

LL(1) 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*

LL(1) Example – First & Follow Sets

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

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

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

LL(1) Example – Predict Set Calculation

$A \rightarrow B\,C\,c$

$First(\,B\,C\,c\,)$

$First(\,B\,) \cup First(\,C\,c\,)$ — B nullable

$\{\,b\,\} \cup First(\,C\,)$ — C not nullable

$\{\,b\,\} \cup \{\,a,\ c,\ d\ \}$

$B \rightarrow \lambda$

$First(\,\lambda\,) \cup Follow(\,B\,)$

$\{\ \} \cup \{\,a,\ c,\ d,\ e,\ f,\ \$\ \}$

$C \rightarrow D\,a\,B$

$First(\,D\,) \cup First(\,a\,B\,)$ — D nullable

$\{\,d\,\} \cup \{\,a\,\}$

$D \rightarrow \lambda$

$First(\,\lambda\,) \cup Follow(\,D\,)$

$\{\ \} \cup \{\,a,\ b,\ c,\ e,\ f,\ \$\ \}$

LL(1) Example – Predict Sets

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

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	

Parse “badeefcac$”

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

UNIVERSITY OF TORONTO

ANTLR: LL(*) Parser Generation

- 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, C#, and C++
- It can also automatically generate Abstract Syntax Trees and tree parsers to process such trees.
- ANTLRv4 can handle direct left recursion automatically.
- ANTLR has been used in a number of production systems including Twitter query processing, 2 billion queries/day.

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.
- Usually easy to write, convenient for semanitc analysis.
- 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., C++, Java, etc.

Recursive Descent Example Grammar

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

Recursive Descent Example 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.

Recursive Descent Example Parser

```
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 Stack Trace

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) $
	)
	$

Recursive Descent Backtracking Example

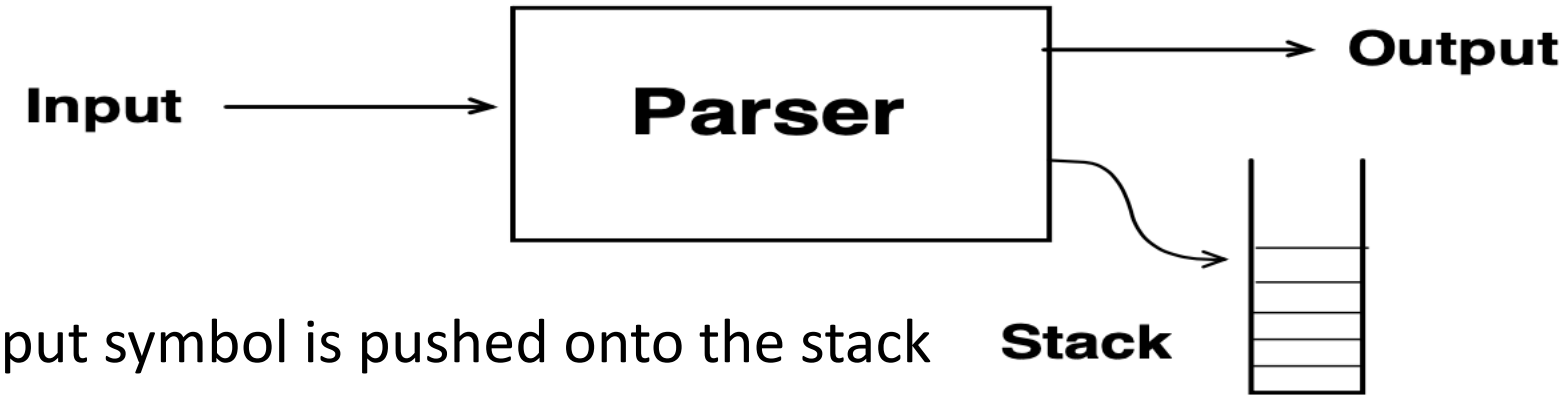
PL/I
```
DECLARE ( A, B, C, D) FIXED BINARY ;     /* Declaration */
DECLARE ( A, B, C, D) = 23 ;             /* Assignment */
```

```
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 ;
```

UNIVERSITY OF TORONTO

Shift Reduce Parsing – Bottom Up LR(k)

- Parser Model



- Parser Actions

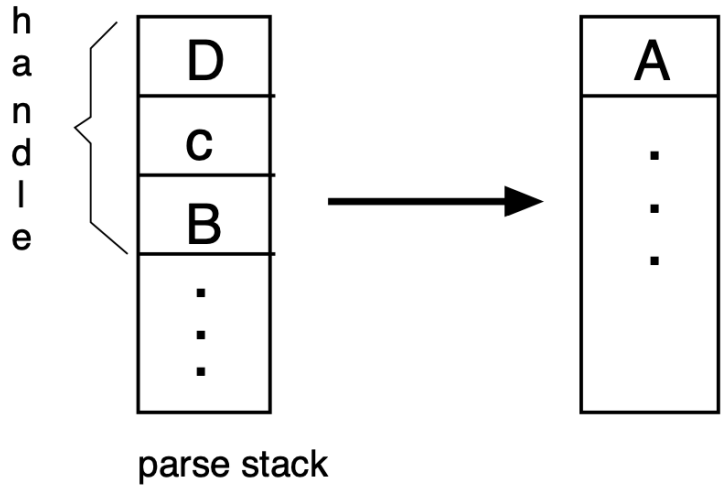
Shift	Next input symbol is pushed onto the stack
Reduce	A sequence of symbols (the handle) starting at the top of the stack is reduced using a production rule to replace the symbol with one non-terminal
Accept	Successful end of parse
Error	Call recovery routine to handle syntax error

- Choie of actions is based on the contents of the stack and the next *k* input tokens (k-symbol lookahead).

Shift Reduce Parsing – Bottom Up LR(k)

- A *handle* is the right hand side (RHS) of some rule in the grammar. Bottom up parsing allows more than one rule to have the same RHS iff the rules can be distinguished using the left context and k-symbol lookahead.
- Given a grammar rule: A → B c D

  a possible Reduce action would be:



Issue: efficiently detecting when a handle is present on top of the parse stack.

Issue: deciding which reduction to perform.

LR(k) Parsing

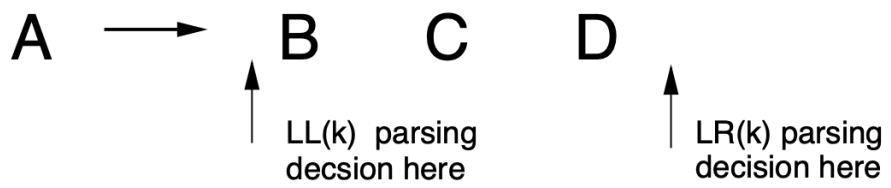
- The contents of the parser stack (left context) represents a string from which the past input can be derived.
- Inputs are stacked until the top elements in the stack (*the handle*) are a complete alternative (RHS) for some rule.
- When a handle is recognized, a reduction is performed and the handle on the stack is replaced by the nonterminal symbol (LHS) of the applicable rule.
- Initial parser stack is ▽ and parsing continues until the stack contains *S* ▽ and the next input is $.
- At each stage the the top elements in the stack represent the initial portion of *one or more* alternative rules.

LR(k) Parsing

- The next input symbol may narrow the number of possible alternatives. If the number of alternatives is narrowed to zero, a syntax error has occurred.
- Finally, an input symbol is stacked that completes one or more alternatives. If there is more than one alternative, the language is not LR(0).
- At this point the next *k* input symbols must provide enough information to distinguish among the alternatives. If it doesn't, the language isn't LR(k).
- For LR(k) we do not need to know which alternative to choose until we reach the end of a rule. *Then* the next k input symbols must be sufficient to decide if a reduction can be performed.

LR(k) Parsing

- Parsing decisions can be made *later* in an LR(k) parser than in an LL(k) parser. This is the reason that $L(\text{ LL(1) }) \subset L(\text{ LR(1) })$



- In contrast, for LL(k) we had to know at the start of an alternative, given k input symbols which alternative to choose.
- LR(k) parsers effectively perform a rightmost derivation in reverse

Rightmost Derivation Example

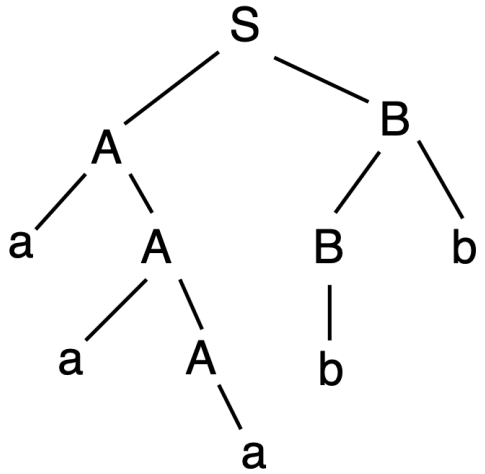
For the grammar:

S	→	A B
A	→	a A
	|	a
B	→	B b
	|	b

Rightmost derivation of a a a b b

0		S
1	→	A B
2	→	A B b
3	→	A b b
4	→	a A b b
5	→	a a A b b
6	→	a a a b b

Parse Tree



Rightmost derivation **in reverse**

6		a a a b b
5	→	a a A b b
4	→	a A b b
3	→	A b b
2	→	A B b
1	→	A B
0	→	S

LR(1) Example

Rule	Grammar
1	L → L , E
2	→ E
3	E → a
4	→ b

Input Tokens

a , b $

Note left recursion in grammar ⇒ not LL(k)

LR(1) Table

Stack Confg	a	b	,	$	St#
▽	Shift a Next	Shift b Next	Error	Error	0
a ▽	Error	Error	Reduce 3	Reduce 3	1
b ▽	Error	Error	Reduce 4	Reduce 4	2
E ▽	Error	Error	Reduce 2	Reduce 2	3
L ▽	Error	Error	Shift , Next	Accept	4
, L ▽	Shift a Next	Shift b Next	Error	Error	5
a , L ▽	Error	Error	Reduce 3	Reduce 3	6
b , L ▽	Error	Error	Reduce 4	Reduce 4	7
E , L ▽	Error	Error	Reduce 1	Reduce 1	8

Stack Snapshots

Parse	State	Input	Action
▽	▽	a	Next
a ▽	1 ▽	,	
E ▽	3 ▽	,	
L ▽	4 ▽	,	Next
, L ▽	5 4 ▽	b	Next
b , L ▽	7 5 4 ▽	$	
E , L ▽	8 5 4 ▽	$	
L ▽	4 ▽	$	Accept

Define: Reduce j , use grammar rule j to replace handle on top of parse stack.

LR(1), SLR(1), and LALR(1)

- For LR table construction techniques, see our textbook *Fischer, Cytron, LeBlanc.*
- LR(1) tables typically contain significant more states than LL(1) tables.
- SLR(1) combines states that share the same rule and use the follow set to select the right reduce action to apply.
- LALR(1) differs from LR(1) in that states with identical reductions but different lookahead sets are merged in LALR(1) and kept distinct in full LR(1).
  - Goal: merge non-essential LR(1) states

YACC, Bison, and JCup

- YACC is a widely available LALR(1) parser generator developed at Bell Labs. Bison is a freeware clone from GNU. Many other variants have been developed.
- YACC assumes input from Lex (or flex)lexical analyzer
- YACC allows the compiler writer to provide an arbitrary piece of C code for each rule in the grammar. This code gets executed just before the parser performs a reduction involving the rule. There is a convention for accessing information stored parallel to the parse stack. YACC does automatic rule splitting to handle code that is not at the end of a rule.
- YACC processes a grammar and produces tables that are used with a fixed parsing algorithm. The pieces of C code become the body of a giant switch statement.
- jflex, jcup – versions of flex and bison for Java.

Summary of Parsing

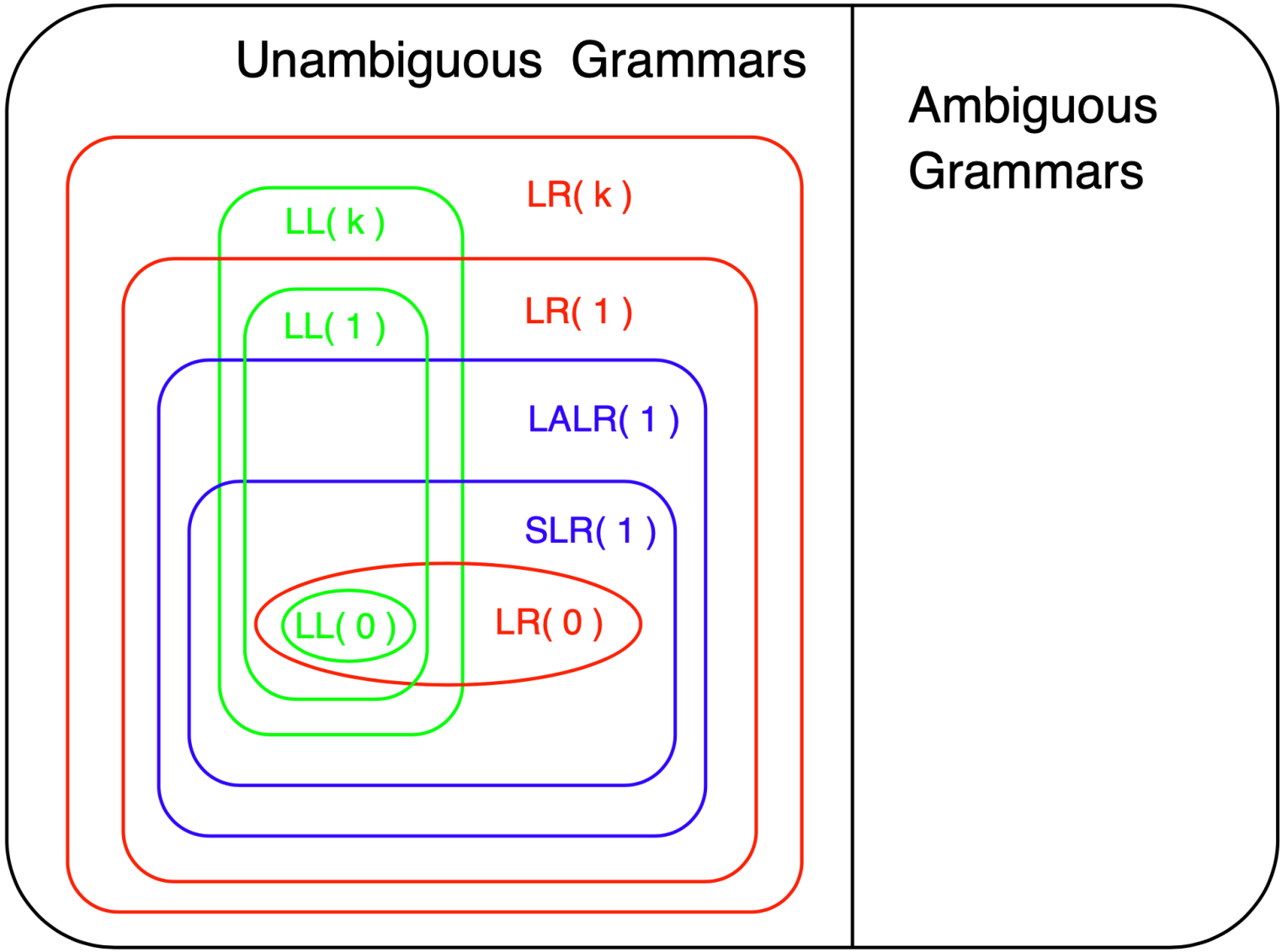
- A parser which can determine the correct derivation (reduction) at each stage as it scans the input stream in one direction is called *deterministic.*
- A grammar is LL(k) if a parser can parse strings scanning from left to right using left derivations with k-symbol lookahead deterministically.
- A grammar is LR(k) if a parser can parse strings scanning from left to right using right reductions with k-symbol lookahead deterministically.
- Recursive descent is the most versatile technique, but the parser must be written manually.

Relationship of Different Parsing Techniques

- LR(k) is the most general deterministic left-to-right language.
  - $L(\text{ LL(1) }) \subset L(\text{ LL(k) })$
  - $L(\text{ LR(1) }) \subset L(\text{ LR(k) })$
  - $L(\text{ LL(k) }) \subset L(\text{ LR(k) })$

- In practice, after eliminating left recursion, LL(k) parser can parse most context free grammars.
- Example: it is possible to parse C++ programs with ANTLRv4
  - https://github.com/antlr/grammars-v4/blob/master/cpp/CPP14Parser.g4
  - https://github.com/antlr/grammars-v4/blob/master/cpp/CPP14Lexer.g4

Heirachy of Grammar Classes



Q/A?