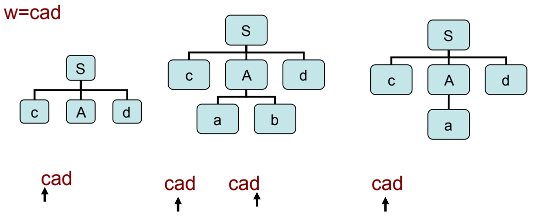
Top down parsing

- Types of parsers:
- Top down:
  - repeatedly rewrite the start symbol;
  - find a left-most derivation of the input string;
  - easy to implement;
  - not all context-free grammars are suitable.
- Bottom up:
  - start with tokens and combine them to form interior nodes of the parse tree;
  - find a right-most derivation of the input string;
  - accept when the start symbol is reached;
  - it is more prevalent.

Topdown parsing with backtracking

- S→cAd
- A→ab|a
- w=cad



Parsing trace

Expansion	Remaining input	Action
S	cad	Try S→cAd
cAd	cad	Match c
Ad	ad	Try A→ab
abd	ad	Match a
bd	d	Dead end, backtrack
ad	ad	Try A→a
ad	ad	Match a
d	d	Match d
Success		

S→cAd
A→ab|a

Parsing trace

Expansion	Remaining input	Action
S	cad	Try S→cAd
cAd	cad	Match c
Ad	ad	Try A→ab
abd	ad	Match a
bd	d	Dead end, backtrack
ad	ad	Try A→a
ad	ad	Match a
d	d	Match d
Success		

- S→cAd
- A→ab|a

Another example

Expansion	Remaining input	Action
S	aabb	Try S→AB
AB	aabb	Try A→aA
aAB	aabb	match a
AB	abb	A→aA
aAB	abb	match a
AB	bb	A→epsilon
B	bb	B→bB
bB	bb	match b
B	b	B→b
b	b	match

S→AB
A→aA|ε
B→bB|b

S➔AB
➔aAB
➔aaAB
➔aaεB
➔aabB
➔aabb

Top down vs. bottom up parsing

- Given the rules
  - S→AB
  - A→aA|ε
  - B→bB|b
- How to parse aabb ?
- Topdown approach

  S➔AB
  ➔aAB
  ➔aaAB
  ➔aaεB
  ➔aabB
  ➔aabb

`Note that it is a left most derivation`

- Bottom up approach

  aabb
  🡐a abb
  🡐aa bb
  🡐aaε bb
  🡐aaA bb
  🡐aA bb
  🡐A bb
  🡐Ab b
  🡐Abb
  🡐AbB
  🡐AB
  🡐S

- If read backwards, the derivation is right most
- In both topdown and bottom up approaches, the input is scanned from left to right

Recursive descent parsing

- **Each method corresponds to a non-terminal**

```
static boolean checkS() {
  int savedPointer = pointer;
  if (checkA() && checkB())
    return true;
  pointer = savedPointer;
  return false;
}
```

S→AB

```
static boolean checkA() {
  int savedPointer = pointer;
  if (nextToken().equals('a') && checkA())
    return true;
  pointer = savedPointer;
  return true;
}
```

A→aA|ε

```
//B→bB|b

static boolean checkB() {
  int savedPointer = pointer;
  if (nextToken().equals('b') && checkB())
        return true;
  pointer = savedPointer;
  if(nextToken().equals('b')) return true;
  pointer = savedPointer;
  return false;
}
```

Left recursion

- What if the grammar is changed to
  S→AB
  A→Aa|ε
  B→b|bB
- The corresponding methods are

```
static boolean checkA() {
  int savedPointer = pointer;
  if (checkA() && nextToken().equals('a'))
   return true;
  pointer = savedPointer;
  return true;
}
static boolean checkB() {
  int savedPointer = pointer;
  if (nextToken().equals('b'))
   return true;
  pointer = savedPointer;
  if(nextToken().equals('b') && checkB()) return true;
  return false;
  pointer = savedPointer;
}
```

Recursive descent parsing (a complete example)

- Grammar
  program→ statement program | statement
  statement→ assignment
  assignment→ ID EQUAL expr
  … …
- Task:
  - Write a java program that can judge whether a program is syntactically correct.
  - This time we will write the parser manually.
  - We can use the scanner
- How to do it?

RecursiveDescent.java outline

```
static int pointer=-1;
static ArrayList tokens=new ArrayList();
static Symbol nextToken() {  }
public static void main(String[] args) {
  Calc3Scanner scanner=new
       Calc3Scanner(new FileReader("calc2.input"));
  Symbol token;
  while(token=scanner.yylex().sym!=Calc2Symbol.EOF)
       tokens.add(token);
  boolean legal= program() && nextToken()==null;
  System.out.println(legal);
}

static boolean program() throws Exception {…}
static boolean statement() throws Exception {…}
static boolean assignment () throws Exception {…}
static boolean expr() {…}
```

One of the methods

```
/**  program-->statement program
     program-->statement
*/
static boolean program() throws Exception {
   int savedPointer = pointer;
   if (statement() && program()) return true;
   pointer = savedPointer;
   if (statement()) return true;
   pointer = savedPointer;
   return false;
}
```

Recursive Descent parsing

- Recursive descent parsing is an easy, natural way to code top-down parsers.
  - All non terminals become procedure calls that return true or false;
  - all terminals become matches against the input stream.
- Example:

```
/**  assignment--> ID=exp **/
static boolean assignment() throws Exception{
    int savePointer= pointer;
    if (  nextToken().sym==Calc2Symbol.ID
        && nextToken().sym==Calc2Symbol.EQUAL
        && expr())
                       return true;
    pointer = savePointer;
    return false;
}
```

Summary of recursive descent parser

- Simple enough that it can easily be constructed by hand;
- Not efficient;
- Limitations:

```
/** E→ E+T | T **/
static boolean expr() throws Exception {
   int savePointer = pointer;
   if ( expr()
          && nextToken().sym==Calc2Symbol.PLUS
          && term())
     return true;
   pointer = savePointer;
   if (term()) return true;
   pointer = savePointer;
   return false;
}
```

- A recursive descent parser can enter into infinite loop.

Left recursion

- Definition
  - A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \Rightarrow^+ A\alpha$
  - There are tow kinds of left recursion:
    - direct left recursive : $A \Rightarrow A\alpha$
    - in-direct left-recursive: $A \Rightarrow^+ A\alpha$, but not $A \Rightarrow A\alpha$
- Example:

  E→E+T|T   is direct left recursive
- Indirect left-recursive

  S→Aa|b

  A→Ac|Sd|ε

  Is S left recursive?

  $S \Rightarrow Aa$

  $\Rightarrow Sda$

  $S \Rightarrow^+ Sda$

Left recursion has to be removed for recursive descent parser

Look at the previous example that works:

```
E→ T+E | T
static boolean expr() throws Exception {
    int savePointer = pointer;
    if (term() && nextToken().sym == Calc2Symbol.PLUS && expr())  return true;
    pointer = savePointer;
    if (term()) return true;
    pointer = savePointer;
    return false;
}
```

What if the grammar is left recursive?

```
E→ E+T | T
static boolean expr() throws Exception {
    int savePointer = pointer;
    if (expr() && nextToken().sym == Calc2Symbol.PLUS && term ())  return true;
    pointer = savePointer;
    if (term()) return true;
    pointer = savePointer;
    return false;
}
```

There will be infinite loop!

Remove left recursion

- Direct left recursion

  A→Aα|β

  expanded form: A→β α α ... α

  Left recursion removed:

  A→β Z

  Z→α Z|ε

- Example:

  E→E+T |T

  expanded form: E→T +T +T ... +T

  E→TZ

  Z→+TZ| ε

Remove left recursion

- In general, for a production

  $A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

  where no β_i begins with A.

  It can be replaced by:

  $A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$

  $A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$

Predictive parsing

- Predictive parser is a special case of top-down parsing when no backtracking is required;
- At each non-terminal node, the action to undertake is unambiguous;

```
STAT→if ...
     | while ...
     | for ...
```

- Not general enough to handle real programming languages;
- Grammar must be left factored;

```
IFSTAT→if EXPR then STAT
       | if EXPR then STAT else STAT
```

  - A predictive parser must choose the correct version of the IFSTAT before seeing the entire input
  - The solution is to factor out common terms:

```
IFSTAT→if EXPR then STAT IFREST
IFREST→else STAT | ε
```

- Consider another familiar example:

  E→T+E | T

Left factoring

- General method

  For a production $A \rightarrow \alpha\ \beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$

  where γ represents all alternatives that do not begin with α,

  it can be replaced by

  $A \rightarrow \alpha\ B \mid \gamma$

  $B \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

- Example

  E→T+E | T

  Can be transformed into:

  E→T E'

  E'→+E| ε

Predictive parsing

- The recursive descent parser is not efficient because of the backtracking and recursive calls.
- a *predictive parser* does not require backtracking.
  - able to choose the production to apply solely on the basis of the next input symbol and the current nonterminal being processed
- To enable this, the grammar must be *LL(1)*.
  - The first "L" means we scan the input from left to right;
  - the second "L" means we create a leftmost derivation;
  - the 1 means one input symbol of lookahead.

More on LL(1)

- LL(1) grammar has no left-recursive productions and has been left factored.
  - left factored grammar with no left recursion may not be LL(1)
- there are grammars that cannot be modified to become LL(1).
- In such cases, another parsing technique must be employed, or special rules must be embedded into the predictive parser.

First() set--motivation

- Navigating through two choices seemed simple enough, however, what happens where we have many alternatives on the right side?
  - statement → assignment | returnStatement | ifStatement | whileStatement | blockStatement
- When implementing the statement() method, how are we going to be able to determine which of the 5 options to match for any given input?
- Remember, we are trying to do this without backtracking, and just one token of lookahead, so we have to be able to make immediate decision with minimal information— this can be a challenge!
- Fortunately, many production rules starts with terminals, which can help in deciding which rule to use.
  - For example, if the input token is 'while', the program should know that the whileStatement rule will be used.

Fisrt(): motivating example

- On many cases, rules starts with non-terminals

```
S→Ab|Bc
A→Df|CA
B→gA|e
C→dC|c
D→h|i
```

  How to parse "gchfc"?

```
S ⇒Bc
  ⇒gAc
  ⇒gCAc
  ⇒gcAc
  ⇒gcDfc
  ⇒gchfc
```

```
                 ⇒ Dfb   ⇒hfb
                /          ⇒ifb
         ⇒Ab
        /    \
       /      ⇒ CAb ⇒ dCAb ⇒ ….
   S                 ⇒ cAb ⇒ …
     \
      \
        ⇒ Bc  ⇒ gAc  ⇒ …
              ⇒ ec
```

If the next token is h, i, d, or c, alternative Ab should be selected.

If the next token is g or e, alternative Bc should be selected.

In this way, by looking at the next token, the parser is able to decide which rule to use without exhaustive searching.

First(): Definition

- The *First set* of a sequence of symbols α, written as First (α), is the set of terminals which start the sequences of symbols derivable from α.
  - If α =>* aβ, then a is in First(α).
  - If α =>* ε , then ε is in First(α).
- Given a production with a number of alternatives:
  - A → α1 | α2 | ...,
  - we can write a predicative parser only if all the sets First(αi) are disjoint.

First() algorithm

- First(): compute the set of terminals that can begin a rule
  1. if **a** is a terminal, then first(a) is {a}.
  2. if A is a non-terminal and A→aα is a production, then add **a** to first(A).
     if A→ε is a production, add ε to first(A).
  3. if A→α_1 α_2 ... α_m is a production, add Fisrt(α1)-ε to First(A).
     If α_1 can derive ε, add First(α_2)-ε to First(A).
     If both α_1 and α_2 derives ε, add First(α_3)-ε to First(A). and so on.
     If α_1 α_2 ... α_m =>*ε , add ε to First(A).
- Example

  S → Aa | b
  A → bdZ | eZ
  Z → cZ | adZ | ε

  First(A) = {First(b), First(e)}= {b, e} (by rule 2, rule 1)
  First(Z) = {a, c, ε } (by rule 2, rule 1)
  First(S)
  = {First(A), First(b)} (by rule 3)
  = {First(A), b} (by rule 1)
  = {b, e, b} = {b, e} (by rule 2)

A slightly modified example

S → Aa | b
A → bdZ | eZ| ε
Z → cZ | adZ | ε

First(S)
= {First(A), First(b)} (by rule 3)
= {First(A), b} (by rule 1)
= {b, e, b} = {b, e} (by rule 2) ?

First(S) = {First(A), First(a), b} = { a, b, e, ε } ?

Answer: First(S) = { a, b, e}

Follow()– motivation

- Consider
  - S➔*aaAb
  - Where A→ε|aA .
- When can A →ε be used? What is the next token expected?
- In general, when A is nullable, what is the next token we expect to see?
  - A non-terminal A is nullable if ε in First(A), or
  - A➔*ε
- the next token would be the first token of the symbol following A in the sentence being parsed.

Follow()

- Follow(): find the set of terminals that can immediately follow a non-terminal
  1. $(end of input) is in Follow(S), where S is the start symbol;
  2. For the productions of the form A→αBβ then everything in First(β) but ε is in Follow(B).
  3. For productions of the form A→αB or A→αBβ where First(β) contains ε, then everything in Follow(A) is in Follow(B).
     - aAb => aαBb
- Example

  S→Aa | b
  A→ bdZ |eZ
  Z→ cZ | adZ | ε
  Follow(S)
  = {$} (by rule 1)
  Follow(A)
  = {a} (by rule 2)
  Follow(Z)
  = {Follow(A)} = {a} (by rule 3)

Compute First() and Follow()

1. E→TE'
2. E'→+TE'|ε
3. T→FT'
4. T'→*FT'|ε
5. F→(E)|id

First (E)
= First(T)
= First(F)
={ (, id }
First (E')
= {+, ε }
First (T')= { *, ε }

Follow (E)
= {), $ }
=Follow(E')
Follow(T)
= Follow(T')
= { +,), $ } First(E') except ε plus Follow(E)
Follow (F)
= { *, +,), $} First(T') except ε plus Follow(T')

The use of First() and Follow()

- If we want to expand S in this grammar:
  - S → A ... | B ...
  - A → a...
  - B → b ... | a...
- If the next input character is b, we should rewrite S with A... or B ....?
  - since First(B) ={a, b}, and First(A)= {a}, we know to rewrite S with B;
  - First and Follow gives us information about the next characters expected in the grammar.
- If the next input character is a, how to rewrite S?
  - ***a*** is in both First(A) and First(B);
  - The grammar is not suitable for predictive parsing.

LL(1) parse table construction

- Construct a parse table (PT) with one axis the set of terminals, and the other the set of non-terminals.
- For all productions of the form A→α
  - Add A→α to entry PT[A,b] for each token b in First(α);
  - add A→α to entry PT[A,b] for each token b in Follow(A) if First(α) contains ε;
  - add A→α to entry PT[A,$] if First(α) contains ε and Follow(A) contains $.

S → Aa | b
A → b d Z | eZ
Z → cZ | a d Z | ε

	a	b	c	d	e	$
S		S→Aa S→b			S→Aa	
A		A→bdZ			A→eZ	
Z	Z→ε Z→adZ		Z→cZ			

		First	Follow
S	Aa	b, e	$
	b	b	
A	bdZ	b	a
	eZ	e	
Z	cZ	c	a
	adZ	a	
	ε	ε	

Construct the parsing table

- if A→α, which column we place A→α in row A?
  - in the column of t, if t can start a string derived from α, i.e., t in First(α).
  - what if α is empty? put A→α in the column of t if t can follow an A, i.e., t in Follow(A).

	a	b	c	d	e	$
S		S→Aa S→b			S→Aa	
A		A→bdZ			A→eZ	
Z	Z→ε Z→adZ		Z→cZ			

S → Aa | b
A → b d Z | eZ
Z → cZ | a d Z | ε

Stack	**RemainingInput**	**Action**
S$	bda$	Predict S→Aa or S→b? suppose Aa is used
Aa$	bda$	Predict A→bdZ
bdZa$	bda$	match
dZa$	da$	match
Za$	a$	Predict Z→ε
a$	a$	match
$	$	accept

- Note that it is not LL(1) because there are more than one rule can be selected.
- The correspondent (leftmost) derivation
  S→Aa→bdZa→bdε a
- Note when Z→ε rule is used.

LL(1) grammar

- If the table entries are unique, the grammar is said to be LL(1):
  - Scan the input from **L**eft to right;
  - performing a **L**eftmost derivation.
- LL(1) grammars can have all their parser decisions made using one token look ahead.
- In principle, can have LL(k) parsers with k>1.
- Properties of LL(1)
  - Ambiguous grammar is never LL(1);
  - Grammar with left recursion is never LL(1);
- A grammar G is LL(1) iff whenever A –> α | β are two distinct productions of G, the following conditions hold:
  - For no terminal a do both α and β derive strings beginning with a (i.e., First sets are disjoint);
  - At most one of α and β can derive the empty string
  - If β =>* ε then α does not derive any string beginning with a terminal in Follow(A)

A complete example for LL(1) parsing

S→P
P→ { D; C}
D→ d, D | d
C→ c, C| c

- The above grammar corresponds loosely to the structure of programs. (program is a sequence of declarations followed by a sequence of commands).
- Need to left factor the grammar first.

S→P
P→ { D; C}
D→ d D2
D2→ , D | ε
C→c C2
C2 → , C | ε

	First	Follow
S	{	$
P	{	$
D	d	;
D2	, ε	;
C	c	}
C2	, ε	}

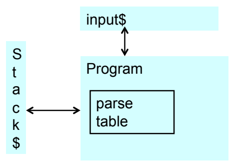
Construct LL(1) parse table

	{	}	;	,	c	d	$
S	S→P$						
P	P→{D;C}						
D						D→dD2	
D2			D2→ε	D2→,D			
C					C→cC2		
C2		C2→ε		C2→,C			

S→P
P→ { D; C}
D→ d D2
D2→ , D | ε
C→c C2
C2 → , C | ε

	First	Follow
S	{	$
P	{	$
D	d	;
D2	, ε	;
C	c	}
C2	, ε	}

LL(1) parse program



- Stack: contain the current rewrite of the start symbol.
- Input: left to right scan of input.
- Parse table: contain the LL(k) parse table.

LL(1) parsing algorithm

- Use the stack, input, and parse table with the following rules:
  - *Accept*: if the symbol on the top of the stack is $ and the input symbol is $, successful parse
  - *match*: if the symbol on the top of the stack is the same as the next input token, pop the stack and advance the input
  - *predict*: if the top of the stack is a non-terminal M and the next input token is ***a***, remove M from the stack and push entry PT[M,a] on to the stack in reverse order
  - *Error*: Anything else is a syntax error

Running LL(1) parser

S→P
P→ { D; C}
D→ d D2
D2→ , D | ε
C→c C2
C2 → , C | ε

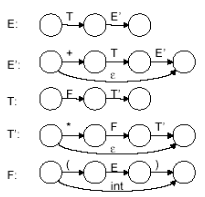
Stack	**Remaining Input**	**Action**
S	{d,d;c}$	predict S→P$
P$	{d,d;c}$	predict P→{D;C}
{ D ; C } $	{d,d;c}$	match {
D ; C } $	d,d;c}$	predict D→d D2
d D2 ; C } $	d,d;c}$	match d
D2 ; C } $	,d;c}$	predict D2→,D
, D ; C } $	,d;c}$	match ,
D ; C } $	d;c}$	predict D→d D2
d D2 ; C } $	d;c}$	match d
D2 ; C } $	;c}$	predict D2→ε
ε ; C } $	;c} $	match ;
C } $	c}$	predict C→c C2
c C2 } $	c}$	match c
C2 } $	}$	predict C2→ε
} $	}$	match }
$	$	accept

Derivation
S→P$
→{ D;C}$
→{d D2 ; C } $
→{d, D ; C } $
→{d,d D2; C } $
→{d,d; C} $
→{d,d; c C2}$
→{d,d; c} $

Note that it is leftmost derivation

The expression example

1. E→TE'
2. E'→+TE'|ε
3. T→FT'
4. T'→*FT'|ε
5. F→(E)|int



	+	*	(	)	int	$
E			E→TE'		E→TE'	
E'	E'→+TE'			E'→ε		E'→ε
T			T→FT'		T→FT'	
T'	T'→ε	T'→*FT'		T'→ε		T'→ε
F			F→(E)		F→int	

Parsing int*int

Stack	**Remaining input**	**Action**
E$	int*int $	predicate E→TE'
TE'$	int*int $	predicate T→FT'
FT'E' $	int*int $	predicate F→int
int T' E' $	int*int $	match int
T' E' $	* int $	predicate T'→*FT'
* F T' E' $	* int $	match *
F T' E' $	int $	predicate F→int
int T' E' $	int $	match int
T' E' $	$	predicate T'→ε
E' $	$	predicate E'→ε
$	$	match $. success.

Parsing the wrong expression int*]+int

Stack	Remaining input	Action
E$	int*]+int $	predicate E→TE’
TE’$	int*]+int $	predicate T→FT’
FT’E’ $	int*]+int $	predicate F→int
int T’ E’ $	int*]+int $	match int
T’ E’ $	*]+int $	predicate T’→*FT’
* F T’ E’ $	*]+ int $	match *
F T’ E’ $	]+int $	error, skip]
F T’ E’$	+ int $	PT[F, +] is sync, pop F
T’E’$	+int$	predicate T’→ ε
E’ $	+int$	predicate E’→+TE’
... ...		

It is easy for LL(1) parsers to skip error

	+	*****	**(**	**)**	**int**	**]**	**$**
E			E→TE’		E→TE’	error	
E’	E’→+TE’			E’→ε		error	E’→ε
T			T→FT’		T→FT’	error	
T’	T’→ε	T’→*FT’		T’→ε		error	T’→ε
F	sync (Follow(F))	sync	F→(E)	sync	F→int	error	sync

Error handling

- There are three types of error processing:
  - report, recovery, repair
- general principles
  - try to determine that an error has occurred as soon as possible. Waiting too long before declaring an error can cause the parser to lose the actual location of the error.
  - Error report: A suitable and comprehensive message should be reported. “Missing semicolon on line 36” is helpful, “unable to shift in state 425” is not.
  - Error recovery: After an error has occurred, the parser must pick a likely place to resume the parse. Rather than giving up at the first problem, a parser should always try to parse as much of the code as possible in order to find as many real errors as possible during a single run.
  - A parser should avoid *cascading errors*, which is when one error generates a lengthy sequence of spurious error messages.

Error report

- report an error occurred and what and where possibly the error is;
  - Report expected vs found tokens by filling holes of parse table with error messages)

	+	*	(	)	int	$
E			E→TE’	Err, int or (expected in line …	E→TE’	
E’	E’→+TE’			E’→ε		E’→ε
T			T→FT’		T→FT’	
T’	T’→ε	T’→*FT’		T’→ε		T’→ε
F			F→(E)		F→int	

Error Recovery

- Error recovery: a single error won’t stop the whole parsing. Instead, the parser will be able to resume the parsing at certain place after the error;
  - Give up on current construct and restart later:
  - Delimiters help parser synch back up
  - Skip until find matching), end,], whatever
  - Use First and Follow to choose good synchronizing tokens
  - Example:
    - duoble d; use Follow(TYPE) D → TYPE ID SEMI
    - junk double d; use First(D)
- Error repair: Patch up simple errors and continue .
  - Insert missing token (;)
  - Add declaration for undeclared name

Types of errors

- Types of errors
  - Lexical: @+2
    - Captured by JLex
  - Syntactical: x=3+*4;
    - Captured by javacup
  - Semantic: boolean x; x = 3+4;
    - Captured by type checker, not implemented in parser generators
  - Logical: infinite loop
    - Not implemented in compilers

Summarize LL(1) parsing

- Massage the grammar
  - Remove ambiguity
  - Remove left recursion
  - Left factoring
- Construct the LL(1) parse table
  - First(), Follow()
  - Fill in the table entry
- Run the LL(1) parse program