Bottom-up analysis

- Shift-reduce parsing.
- Constructs the derivation tree from bottom to top.
- Reduces the string to the start symbol.
- Produces a reverse right derivation.

Example: G(E):

1. $E \rightarrow E + T$
2. $\quad | \ T$
3. $T \rightarrow T * F$
4. $\quad | \ F$
5. $F \rightarrow id$
6. $\quad | \ (E)$

Input: id + id

Reverse right derivation (the art of finding handles).

The handles are underlined:

$$\underline{id} + id \underset{rm}{\Leftarrow} \underline{F} + id \underset{rm}{\Leftarrow} \underline{T} + id \underset{rm}{\Leftarrow} E + \underline{id} \underset{rm}{\Leftarrow}$$

$$E + \underline{F} \underset{rm}{\Leftarrow} \underline{E + T} \underset{rm}{\Leftarrow} E \quad \text{(start symbol} \Rightarrow \text{accept)}$$

A shift-reduce parser finds the rules in the order:
5, 4, 2, 5, 4, 1.

The transformation to right derivation is called the *canonical reduction sequence*.

This sequence is reached by so-called *"handle pruning"* (handle elimination).

How is this sequence arrived at?

$$w = \gamma_n \underset{rm}{\Leftarrow} \gamma_{n-1} \underset{rm}{\Leftarrow} \cdots \underset{rm}{\Leftarrow} \gamma_1 \underset{rm}{\Leftarrow} \gamma_0 = S$$

where

w: statement
$\gamma_n, \gamma_{n-1}, ..., \gamma_1$: sentential forms
γ_0: start symbol

Method:

1. Localize the handle β_n in the sentential form γ_n.
2. Replace the handle β_n with A.
   (There is a rule $A \rightarrow \beta_n$). Then we get γ_{n-1}.
3. Localize the handle β_{n-1} in sentential form γ_{n-1}.
4. etc. until the target symbol is reached.

- A **stack** is needed to implement a shift-reduce parser.
- New symbols:

  $ (or _!_, or ⊢) marks the end of the string

  ⊣ marks the start (the stack is empty)

Stack	Input	
⊣	w⊢	Configuration at the start
⊣S	⊢	Configuration at the end

- The analyser works by *shifting* symbols on the stack (from input) to a handle (i.e., a complete right side of some rule) is on the top of the stack (never inside).
- Then the handle is reduced to the corresponding left side (i.e., nonterminal).

Example: Parse `id + id` according to G(E):

Step	Stack	Input	Event	Handle	Rule
1	⊣	id+id⊢	Shift		
2	⊣id	+id⊢	Reduce	$F \rightarrow id$	5
3	⊣F	+id⊢	Reduce	$T \rightarrow F$	4
4	⊣T	+id⊢	Reduce	$E \rightarrow T$	2
5	⊣E	+id⊢	Shift		
6	⊣E+	id⊢	Shift		
7	⊣E+id	⊢	Reduce	$F \rightarrow id$	5
8	⊣E+F	⊢	Reduce	$T \rightarrow F$	4
9	⊣E+T	⊢	Reduce	$E \rightarrow E + T$	1
10	⊣E	⊢	Accept		

Accept only if we only have the target symbol on the stack and there is no more input (otherwise an *error* has occurred*).*

Problems:

1. How do we find the handle?
2. How is reduction accomplished when we have several rules which have the same right side?

An LR-parser uses *lookahead*.

LR-parser — Automatic construction

Characteristics:
+ An LR analyser can analyze almost all languages which can be described with a CFG.

+ More generally: Rewriting seldom needed, left recursion, right recursion, several rules with the same symbols at the beginning are not a problem.
+ As fast as hand-written.
+ Discovers the error as soon as it is possible.
- The semantics can not be introduced so easily.
- Difficult to generate by hand.

LR classes	
LR(0)	Too weak (no lookahead)
SLR(1)	*Simple* LR, 1 token lookahead
LALR(1)	Most common, 1 token lookahead
LR(1)	1 token lookahead - tables far too big
LR(k)	k tokens lookahead

Example of table size:
Pascal: LALR(1) 20 pages, 2 bytes per entry.

Differences:
- Table size varies widely.
- Errors not discovered as quickly.

Limitations in the language definition.

Augmented grammar (extended grammar)

Add a rule which includes the target symbol <S> and the end symbol ⊢—

<SYS> → <S> ⊢—

Example of parsing using the LR method

(See the following pages for grammar and parse tables)

Input: a, b

Step	Stack	Input	Table entries
1	**—⊢0**	**a,b⊢—**	**ACTION[0, a] = S4**
2	**—⊢0a4**	**,b⊢—**	**ACTION[4, ,] = R3 (E → a)**
	—⊢0E	**,b⊢—**	**GOTO[0, E] = 6**
3	**—⊢0E6**	**,b⊢—**	**ACTION[6, ,] = R2 (L → E)**
	—⊢0L	**,b⊢—**	**GOTO[0, L] = 1**
4	**—⊢0L1**	**,b⊢—**	**ACTION[1, ,] = S2**
5	**—⊢0L1,2**	**b⊢—**	**ACTION[2, b] = S5**
6	**—⊢0L1,2b5**	**⊢—**	**ACTION[5, ⊢—] = R4 (E → b)**
	—⊢0L1,2E	**⊢—**	**GOTO[2, E] = 3**
7	**—⊢0L1,2E3**	**⊢—**	**ACTION[3, ⊢—] = R1 (L→ L,E)**
	—⊢0L	**⊢—**	**GOTO[0, L] = 1**
8	**—⊢0L1**	**⊢—**	**ACTION[1, ⊢—] = A (accept)**

Example of an SLR(1) grammar

```
          T H E   V O C A B U L A R Y

T E R M I N A L S          N O N T E R M I N A L S

  1.  _!_                    5.  <list>
  2.  ,                      6.  <element>
  3.  a
  4.  b

  GOAL SYMBOL IS:  <list>



          T H E   P R O D U C T I O N S

 1.  <list> ::= <list> , <element>
 2.           ! <element>

 3.  <element> ::= a
 4.              ! b
```



```
          T H E   P A R S E R   A C T I O N   T A B L E

    TOP OF STACK                    INPUT SYMBOL
                                     -!- ,  a b
  STATE       NAME                !  1  2  3  4
  --------------------------------!------------
    0       <SYS>                 !  X  X S4 S5
    1       <list>                !  A S2  *  *
    2       ,                     !  X  X S4 S5
    3       <element>             ! R1 R1  *  *
    4       a                     ! R3 R3  X  X
    5       b                     ! R4 R4  X  X
    6       <element>             ! R2 R2  *  *


             T H E   G O T O   T A B L E

!  (Note: The GOTO table is not the same as the GOTO graph, which
!  represents both the ACTION table and the GOTO table)

    TOP OF STACK                    SYMBOL
                                    <list><element>
  STATE       NAME                !  5  6
  --------------------------------!------
    0       <SYS>                 !  1  6
    1       <list>                !  *  *
    2       ,                     !  *  3
    3       <element>             !  *  *
    4       a                     !  *  *
    5       b                     !  *  *
    6       <element>             !  *  *

!
!  NB:  the following abbreviations are used in the tables above
!
!           X = error
!
!           * = impossible to end up here!
```

Construction of parse tree (Bottom-up)

- *Shift* operations:

  Create a one-node tree containing the shifted symbol.

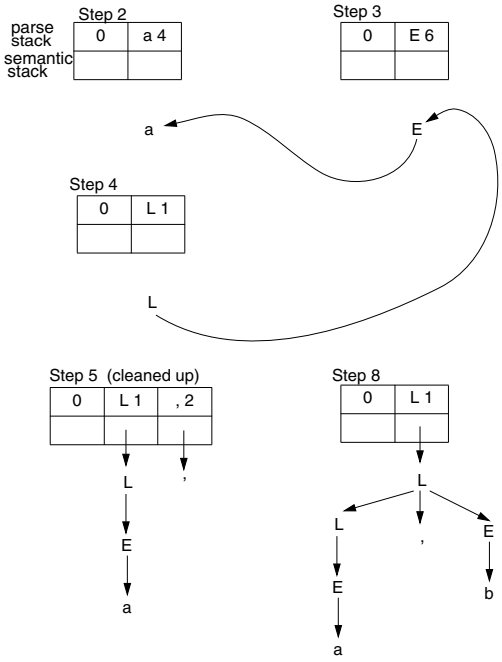
- *Reduce* operations:

  When reducing a handle β to A (as in $A \rightarrow \beta$), create a new node A whose "children" are those nodes that were created in the handle.

During the analysis we have a forest of sub-trees. Each entry in the stack points to its corresponding sub-tree. When we *accept,* the whole parse tree is completed.

Example.
Construction of a parse tree.

L → L, E
L → E
E → a
E → b



Definition: **LR-grammar:**

A grammar for which a unique LR-table can be constructed is called an *LR grammar* (LR(0), SLR(1), LALR(1), LR(1), ...).

- No ambiguous grammars are LR grammars.
- There are unambiguous grammars which are not LR grammars.
- The state at the top of the stack contains all the information we need.

Definition: **Viable prefix**

The prefixes of a right sentential form which do not contain any symbols to the right of a handle.

Example: (See next page for grammar and parse table)

1. <list> → <list> , <element>
2. | <element>
3. <element> → a
4. | b

Input: a, b, a

Right derivation (handles are underlined)

<list> $\underset{rm}{\Rightarrow}$ <u><list> , <element></u>

$\underset{rm}{\Rightarrow}$ <list> , <u>a</u>

$\underset{rm}{\Rightarrow}$ <u><list> , <element></u> , a

$\underset{rm}{\Rightarrow}$ <list> , <u>b</u> , a

$\underset{rm}{\Rightarrow}$ <u><element></u> , b, a

$\underset{rm}{\Rightarrow}$ <u>a</u> , b, a

Viable prefixes of the sentential form: <list> , b, a are

{ ε, <list>, <list> , , <list> , b }

A parser generator constructs a DFA which recognises all *viable prefixes*.

This DFA is then transformed into table form.

Automatic construction of the ACTION- and GOTO-table:

Definition: **LR(0) item**

An *LR(0) item* of a rule P is a rule with a dot ''•'' somewhere in the right side.

Example:
All LR(0) items of the production

1. <list> → <list> , <element>

are

<list> → • <list> , <element>
<list> → <list> • , <element>
<list> → <list> , • <element>
<list> → <list> , <element> •

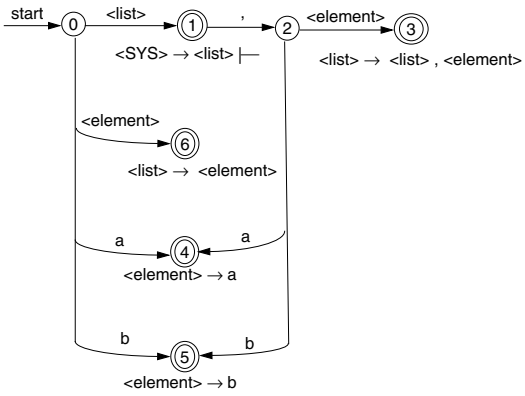
Intuitively an *item* is interpreted as how much of the rule we have found and how much remains.

***I**tems* are put together in sets which become the LR analyser's state.

We want to construct a DFA which recognises all *viable prefixes* of G(<SYS>):

Augmented grammar:

0. <SYS> → <list> ⊢
1. <list> → <list> , <element>
2. | <element>
3. <element> → a
4. | b



GOTO-graph

(A GOTO-graph is not the same as a GOTO-table but corresponds to an ACTION + GOTO-table. The graph discovers *viable prefixes*.)

Algorithm to construct a GOTO-graph from the set of LR(0)-items

(A detailed description is given in the textbook p. 221 ff)

0 <SYS> → • <list> ⊢	Kernel (Basis)
<list> → • <list> , <element> <list> → • <element> <element> → • a <element> → • b	Closure (of kernel items)

Each set corresponds to a node in the GOTO graph. When we have found the symbol behind ''•'' we move the dot over the symbol.

1 <SYS> → <list> • ⊢ <list> → <list> • , <element>	Kernel
(empty *closure* as ''•'' precedes terminals)	Closure

(shift-reduce conflict in the Kernel set!)

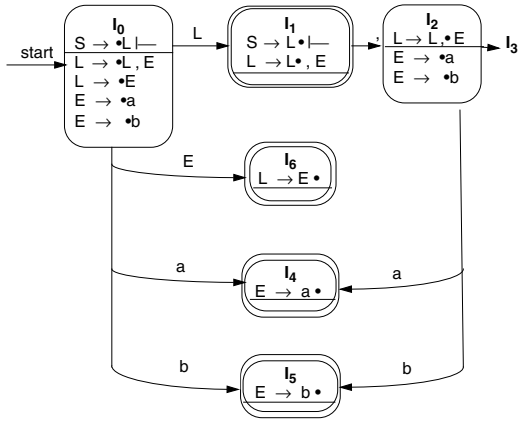
2 <list> → <list> , • <element>	Kernel
<element> → • a <element> → • b	Closure

etc., (see the final result below).

Based on the canonical collection of LR(0) items draw the GOTO graph.

The GOTO graph discovers those prefixes of right sentential forms which have (at most) one handle furthest to the right in the prefix.

GOTO graph with its LR(0) items:



Canonical collection of LR(0) items for the grammar G(<SYS>)

I_0	<SYS>	→	•<list> ⊢—
	<list>	→	•<list> , <element>
	<list>	→	•<element>
	<element>	→	•a
	<element>	→	•b
I_1	<SYS>	→	<list> • ⊢—
	<list>	→	<list> •, <element>
I_2	<list>	→	<list> , • <element>
	<element>	→	•a
	<element>	→	•b
I_3	<list>	→	<list> , <element> •
I_4	<element>	→	a •
I_5	<element>	→	b •
I_6	<list>	→	<element> •

L R (0) S E T S O F I T E M S

```
STATENR.      ITEMSET                          SUCCESSOR

0     <SYS> --> .<list>                          1
      <list> --> .<list> , <element>             1
      <list> --> .<element>                      6
      <element> --> .a                           4
      <element> --> .b                           5

1     <SYS> --> <list> .                       ==>0
      <list> --> <list> ., <element>             2

2     <list> --> <list> , .<element>             3
      <element> --> .a                           4
      <element> --> .b                           5

3     <list> --> <list> , <element> .          ==>1

4     <element> --> a .                        ==>3

5     <element> --> b .                        ==>4

6     <list> --> <element> .                   ==>2
```

T H E L O O K - A H E A D S E T S

```
NONTERMINAL SYMBOLS          LOOK-AHEAD SET

  <list>                       _!_ ,

  <element>                    _!_ ,
```


Fill in the ACTION table according to the GOTO graph

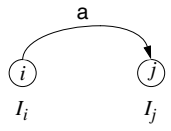
I_i : state i (line i, itemset i)

1. If there is an item

   $\langle A \rangle \rightarrow \alpha \bullet a\beta \in I_i$

   and

   GOTO(I_i , a) = I_j



Fill in *shift j* for row i and column for symbol a.

ACTION:

		a	
i		*shift j*	

2. If there is a complete item
   (i.e., ends in a dot "•"):

   $\langle A \rangle \rightarrow \alpha \bullet \in I_i$

   Fill in *reduce x* where *x* is the production number for

   x: $\langle A \rangle \rightarrow \alpha$

   In which column(s) should reduce x be written?

   LR(0) fills in for all input.

   SLR(1) fills in for all input in FOLLOW(<A>).

   LALR(1) fills in for all those that can follow a certain instance of <A>

3. If we have

   $\langle SYS \rangle \rightarrow \langle S \rangle \bullet$ ⊢—

   accept the symbol ⊢—

4. Otherwise *error.*

Fill in the GOTO table

<A> → α• ∈ I_i

If the GOTO graph(I_i , <A>) = I_j

fill in GOTO[i, <A>] = j

GOTO:

	<A>	
i	j	

Limitations of LR grammars

- No ambiguous grammar is LR(k)
- Shift-Reduce conflict
- Reduce-Reduce conflict

What is a conflict?

Given the current state and input symbol it is not possible to select an **action** (there are two or more entries in the action table).

Example. Reduce/Reduce-conflict

```
procid → ident
exp → ident
```

Example. Shift/Reduce-conflict

```
if ... then ....      else
```

Stack top Next token

Example of Reduce/Reduce conflict:

X → (A) | (B)

A→ REAL | INTEGER | IDENT

B → BOOL | CHAR | IDENT

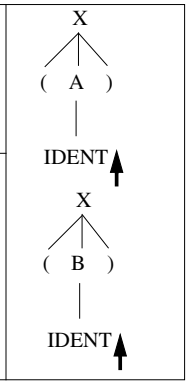
Factorised grammar:

X → (A) | (B) | (C)

A→ REAL | INTEGER

B → BOOL | CHAR

C → IDENT



Example of Shift/Reduce conflict: (One token lookahead: ( )

X → (A) | OPT_Y (B)

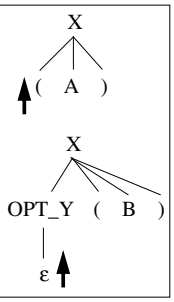
OPT_Y → ε | Y

Y → ... A → ... B → ...

Expanded grammar:

X → (A) | (B) | Y (B)

Y → ... A → ... B → ...



Conflict elimination by

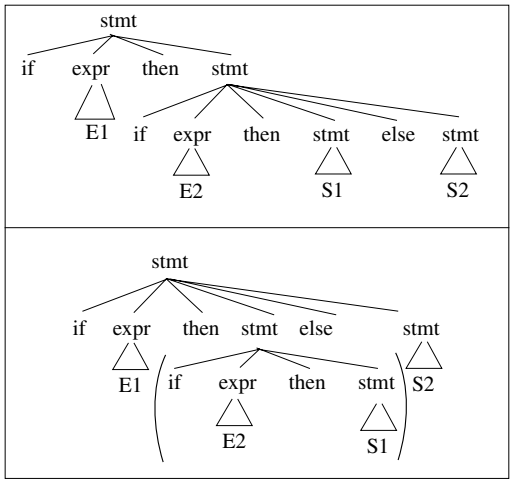
- factorising
- expansion
- rewriting

A concrete example from the book pp. 174, 175:
rewriting

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | other
```

Ambiguous grammar, as the following statement has two parse trees:

```
                    -alt 2
if E1 then if E2 then S1 else S2
                              -alt 1
```



Rewriting the grammar

stmt → matched_stmt
| unmatched_stmt

matched_stmt →
if expr then matched_stmt else matched_stmt
| other

unmatched_stmt →
if expr then stmt
| if expr then matched_stmt else unmatched_stmt

This grammar will create the first alternative of parse tree, i.e. try to match immediately if possible.

Shift-Reduce conflict

If both criterion (1) and (2) hold for a certain item-set:

<A> → α •
<B> → γ • aβ

Many conflicts are resolved in SLR(1) and LALR(1) by using lookahead. These methods construct FOLLOW-sets, e.g.:

FOLLOW(<list>) = { "⊢—", ",", }

This set specifies terminal symbols which can follow the nonterminal <list> during derivation, also called the *lookahead-set*.

All SLR(1) grammars are unambiguous but there are unambiguous grammars which are not SLR(1). The same is true for LR(k). For LR(k) the intersection of the lookahead sets must also be non-empty.

Reduce-Reduce conflict

If there are several complete items in an item-set, e.g.

1. <A> → α • (*reduce 1*)
2. <B> → β α• (*reduce 2*)

An LALR(1) grammar

run through the SLR(1) generator

```
                T H E   V O C A B U L A R Y

T E R M I N A L S                N O N T E R M I N A L S

    1.  _!_                          7.  <S>
    2.  B                            8.  <A>
    3.  C
    4.  A
    5.  E
    6.  D

GOAL SYMBOL IS:  <S>

              T H E   P R O D U C T I O N S

        1.  <S> ::= B <A> C
        2.          ! A E C
        3.          ! A <A> D

        4.  <A> ::= E
```

L R (0) S E T S O F I T E M S

STATENR.	ITEMSET	SUCCESSOR
0	<SYS> --> .<S>	1
	<S> --> .B <A> C	2
	<S> --> .A E C	6
	<S> --> .A <A> D	6
1	<SYS> --> <S> .	==>0
2	<S> --> B .<A> C	3
	<A> --> .E	5
3	<S> --> B <A> .C	4
4	<S> --> B <A> C .	==>1
5	<A> --> E .	==>4
6	<S> --> A .E C	7
	<S> --> A .<A> D	9
	<A> --> .E	7
7	<A> --> E .	==>4
	<S> --> A E .C	8
8	<S> --> A E C .	==>2
9	<S> --> A <A> .D	10
10	<S> --> A <A> D .	==>3

T H E L O O K - A H E A D S E T S

NONTERMINAL SYMBOLS	LOOK-AHEAD SET
<S>	_!_
<A>	C D

****** SHIFT-REDUCE CONFLICT IN STATE 7
PRODUCTIONS: 2 4

THIS GRAMMAR IS NOT SLR(1)

An LALR(1) grammar

run through the LALR(1) generator

T H E V O C A B U L A R Y

T E R M I N A L S N O N T E R M I N A L S

1. _!_ 7. <s>
2. b 8. <a>
3. c
4. a
5. e
6. d

Goal symbol is: <s>

T H E P R O D U C T I O N S

1. <s> ::= b <a> c
2. ! a e c
3. ! a <a> d

4. <a> ::= e

L R (0) S E T S O F I T E M S

statenr.	itemset	successor
0	<SYS> --> .<s>	1
	<s> --> .b <a> c	2
	<s> --> .a e c	6
	<s> --> .a <a> d	6
1	<SYS> --> <s> .	==>0
2	<s> --> b .<a> c	3
	<a> --> .e	5
3	<s> --> b <a> .c	4
4	<s> --> b <a> c .	==>1
5	<a> --> e .	==>4
6	<s> --> a .e c	7
	<s> --> a .<a> d	9
	<a> --> .e	7
7	<a> --> e .	==>4
	<s> --> a e .c	8
8	<s> --> a e c .	==>2
9	<s> --> a <a> .d	10
10	<s> --> a <a> d .	==>3

T H E L O O K - A H E A D S E T S

STATE	PRODUCTION	LOOK-AHEAD SET
1	0	_!_
4	1	_!_ b c a e d
5	4	_!_ b c a e d
7	4	d
8	2	_!_ b c a e d
10	3	_!_ b c a e d

T H E P A R S E R A C T I O N T A B L E

TOP OF STACK		INPUT SYMBOL					
STATE	NAME	! 1	2	3	4	5	6
0	<SYS>	! X	S2	X	S6	X	X
1	<s>	! A	X	X	X	X	X
2	b	! X	X	X	X	-4	X
3	<a>	! X	X	-1	X	X	X
6	a	! X	X	X	X	S7	X
7	e	! X	X	-2	X	X	R4
9	<a>	! *	*	*	*	*	-3

T H E G O T O T A B L E

TOP OF STACK		SYMBOL	
STATE	NAME	! 7	8
0	<SYS>	! 1	*
1	<s>	! *	*
2	b	! *	3
3	<a>	! *	*
6	a	! *	9
7	e	! *	*
9	<a>	! *	*

NB: The tables are optimised.
For example, **-4** in the action table stands for *shift-reduce* according to rule 4.

Methods for syntax error management

1. Panic mode
2. Coding of wrong entries in the **`ACTION`** table
3. Wrong productions
4. Language-independent methods:

   4a) *Continuation method*, Röchrich (1980)

   4b) *Automatic error recovery*, Burke & Fisher (1982)

1. Panic mode

c) Skip input until either

   i) Parsing can continue, or

   ii) An important symbol has been found (e.g. **`PROCEDURE, BEGIN, WHILE,...`**)

d) If parsing can not continue:

   Pop the stack until the important symbol is accepted.

   If you reach the bottom of the stack:

   `"Quit --Unrecoverable error."`

- Much input can be removed.
- Semantic info on the stack disappears.

+ Systematic, simple to implement.

+ Efficient, very fast and does not require extra memory.

(The other methods are dealt with later in the section on error management.)