Syntactic analysis

Input: token sequence

Tasks:

Parsing: construct derivation according to **concrete syntax**,
Tree construction according to **abstract syntax**,
Error handling (detection, message, recovery)

Result: abstract program tree

Compiler module parser:

deterministic stack automaton, augmented by actions for tree construction

top-down parsers: leftmost derivation; tree construction top-down or bottom-up

bottom-up parsers: rightmost derivation backwards; tree construction bottom-up

Abstract program tree (condensed derivation tree):

represented by a
- **data structure in memory** for the translation phase to operate on,
- linear **sequence of nodes on a file** (costly in runtime),
- **sequence of calls** of functions of the translation phase.

Concrete and abstract syntax

concrete syntax	**abstract syntax**
context-free grammar	context-free grammar
defines the structure of source programs	defines abstract program trees
unambigous	usually ambiguous
specifies derivation and parser	translation phase is based on it
parser actions specify the --->	tree construction
some chain productions only for syntactic purpose `Expr ::= Fact` have no action	keep only semantically relevant ones no node created

symbols of syntactic chain productions comprised in symbol classes `Exp={Expr,Fact}`

same action at structural equivalent productions:

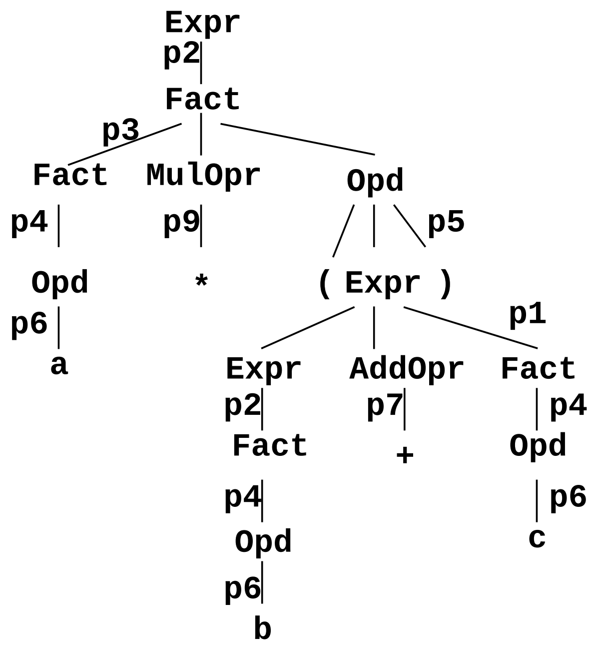
```
Expr ::= Expr AddOpr Fact &BinEx
Fact ::= Fact MulOpr Opd    &BinEx
```

terminal symbols	keep only semantically relevant ones as tree nodes
given the concrete syntax and the actions and	the symbol classes the abstract syntax can be generated

Example: concrete expression grammar

name production ***action***

```
p1:  Expr   ::= Expr AddOpr Fact BinEx
p2:  Expr   ::= Fact
p3:  Fact   ::= Fact MulOpr Opd  BinEx
p4:  Fact   ::= Opd
p5:  Opd    ::= '(' Expr ')'
p6:  Opd    ::= Ident            IdEx
p7:  AddOpr ::= '+'              PlusOpr
p8:  AddOpr ::= '-'              MinusOpr
p9:  MulOpr ::= '*'              TimesOpr
p10: MulOpr ::= '/'              DivOpr
```



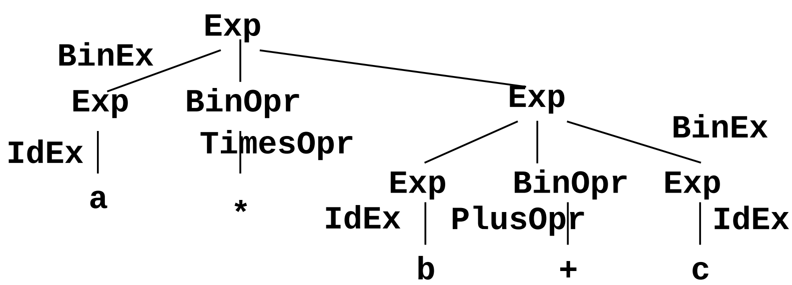
derivation tree for **a * (b + c)**

Example: abstract expression grammar

name **production**

```
BinEx:     Exp    ::= Exp BinOpr Exp
IdEx:      Exp    ::= Ident
PlusOpr:   BinOpr ::= '+'
MinusOpr:  BinOpr ::= '-'
TimesOpr:  BinOpr ::= '*'
DivOpr:    BinOpr ::= '/'
```

abstract program tree for **a * (b + c)**



symbol classes: `Exp = { Expr, Fact, Opd }, BinOpr = { AddOpr, MulOpr }`

Actions of the concrete syntax: **productions** of the abstract syntax to create tree nodes for
no action at a concrete chain production: **no tree node** is created

Recursive descent parser

top-down (construction of the **derivation** tree), **predictive** method

Sytematic transformation of a context-free grammar into a set of functions:

non-terminal symbol X	function X
alternative productions for X	branches in the function body
decision set of production pi	decision for branch pi
non-terminal occurrence X ::= ... Y ...	function call Y()
terminal occurrence X ::= ... t ...	accept a token t an read the next token

Example:

```
p1: Stmt ::= Variable ':=' Expr  p2: Stmt ::= 'while' Expr 'do' Stmt
```

Function:

```
void Stmt ()
{  switch (CurrSymbol)
   {
case decision set for p1:
   Variable();
   accept(assignSym);
   Expr();
   break;
case decision set for p2:
   accept(whileSym);
   Expr();
   accept(doSym);
   Stmt();
   break;

   default: Fehlerbehandlung();
}  }
```


Grammar conditions for recursive descent

A context-free grammar is **strong LL(1)**, if for any pair of productions that have the same symbol on their left-hand sides, the **decision sets are disjoint**:

productions:	A ::= u		A ::= v	
decision sets:	First (u Follow(A))	$\cap$	First (v Follow(A))	$= \emptyset$

First set and **follow set:**

First (u) := { t ∈ T | v ∈ V* exists and a derivation u ⇒* t v } and ε ∈ First (u) if u ⇒* ε exists

Follow (A) := { t ∈ T | u,v ∈ V* exist, A ∈ N and a derivation S ⇒* u A v such that t ∈ First (v) }

Example:

	production		**decision set**
p1:	Prog	::= Block #	begin
p2:	Block	::= begin Decls Stmts end	begin
p3:	Decls	::= Decl ; Decls	new
p4:	Decls	::=	Ident begin
p5:	Decls	::= new Ident	new
p6:	Stmts	::= Stmts ; Stmt	begin Ident
p7:	Stmts	::= Stmt	begin Ident
p8:	Stmt	::= Block	begin
p9:	Stmt	::= Ident := Ident	Ident

non-terminal X	**First(X)**	**Follow(X)**
Prog	begin	
Block	begin	# ; end
Decls	ε new	Ident begin
Decl	new	;
Stmts	begin Ident	; end
Stmt	begin Ident	; end

Grammar transformations for LL(1)

Consequences of strong LL(1) condition: A strong LL(1) grammar can not have

- **alternative productions that begin with the same symbols**
- **productions that are directly or indirectly left-recursive.**

Simple grammar transformations that keep the defined language invariant:

	non-LL(1) productions	transformed
• **left-factorization:** u, v, w ∈ V* X ∈ N does not occur in the original grammar	A ::= v u A ::= v w	A ::= v X X ::= u X ::= w
• **elimination of direct recursion :**	A ::= A u A ::= v	A ::= v X X ::= u X X ::=

EBNF constructs can avoid violation of strong LL(1) condition:

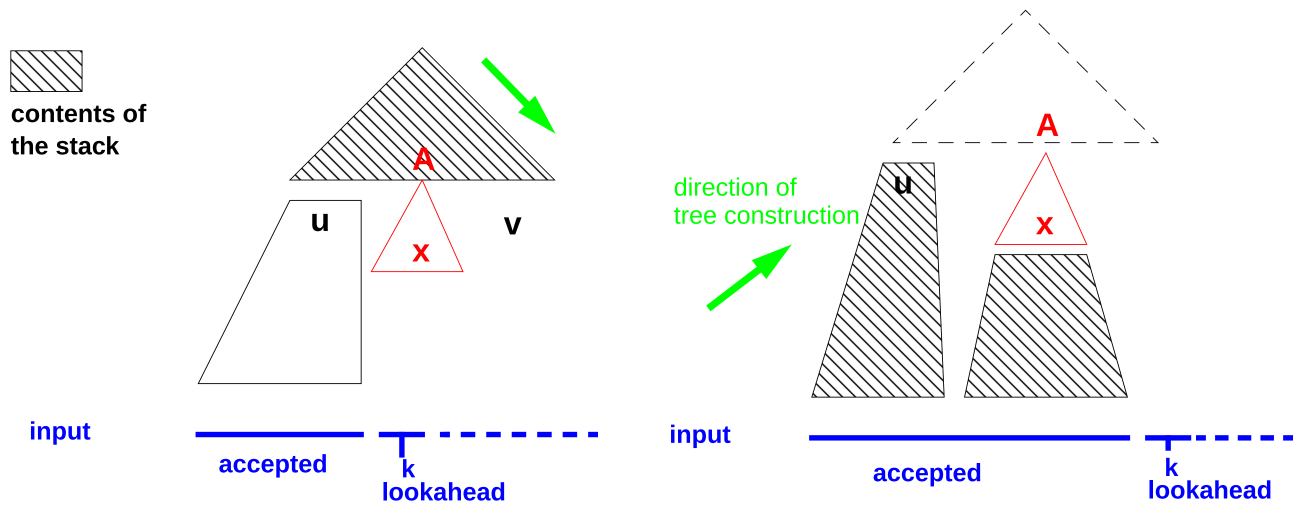
for example repetition of u: A ::= v (u)* w
additional condition: First(u) ∩ First(w Follow(A)) = ∅
branch in the function body: v **while (CurrToken in First(u)) { u }** w
correspondingly for EBNF constructs u^+, [u]

Comparison: top-down vs. bottom-up

Information a stack automata has when it decides to apply production A ::= x :

top-down, predictive
leftmost derivation

bottom-up
rightmost derivation backwards



A bottom-up parser has seen more of the input when it decides to apply a production.

Consequence: **bottom-up** parsers and their grammar classes are more **powerful**.

LR(1) automata

LR(k) grammars introduced 1965 by Donald Knuth; non-practical until subclasses were defined.

LR parsers construct the derivation tree **bottom-up,** a right-derivation backwards.

LR(k) grammar condition can not be checked directly, but
a context-free grammar is LR(k), iff the (canonical) **LR(k) automaton is deterministic**.

We consider only **1 token lookahead: LR(1).**

The **stacks** of LR(k) (and LL(k)) automata **contain states**.
The construction of LR and LL states is based on the notion of **items** (also called situations):

An **item** represents the progress of analysis with respect to one production:

[A ::= u . v R] z. B. **[B ::= (. D ; S) {#}]**

. position of analysis

R expected **right context,** i. e. a set of terminals which may follow after the application of the complete production.
(for general k: R contains terminal sequences not longer than k)

Reduce item:

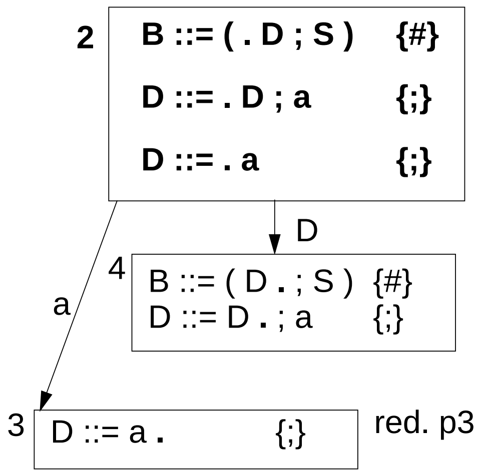
[A ::= u v . R] z. B. **[B ::= (D ; S) . {#}]**

characterizes a reduction using this production if the next input token is in R.

Each **state** of an automaton represents **LL: one item** **LR: a set of items**

LR(1) states and operations

A **state of an LR automaton represents a set of items**
Each item represents a way in which analysis may proceed from that state.



A shift transition is made under
a **token read** from input or
a **non-terminal** symbol
obtained from a **preceding reduction.**
The state is pushed.

A **reduction** is made according to a reduce item.
n states are popped for a production of length n.

Operations:	**shift**	read and push the next state on the stack
	reduce	reduce with a certain production, pop n states from the stack
	error	error recognized, report it, recover
	stop	input accepted

Example for a LR(1) automaton

Grammar:

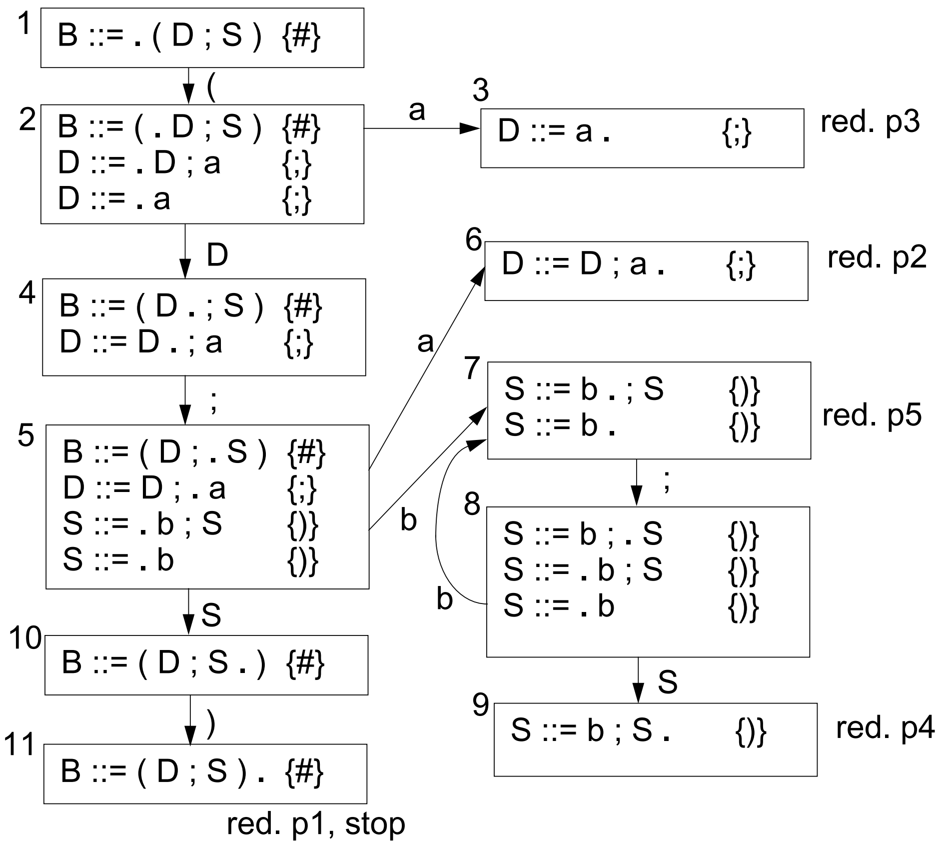
p1 B ::= (D ; S)

p2 D ::= D ; a

p3 D ::= a

p4 S ::= b ; S

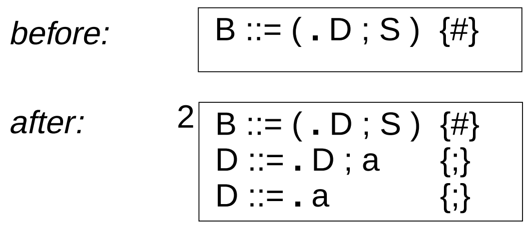
p5 S ::= b



Construction of LR(1) automata

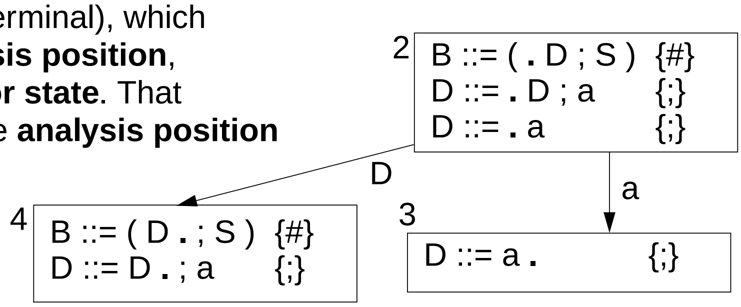
Create the start state; create transitions and states as long as new ones can be created.

Transitive closure is to be applied to each state:
If **[A ::= u . B v R]** is in state q,
with the analysis position before a non-terminal B,
then for each production **B ::= w**
[B ::= . w First (v R)]
has to be added to state q.



Start state:
Closure of **[S ::= . u {#}]**
S ::= u is the **unique start production**,
is an **artificial end symbol** (eof)

Successor states:
For each **symbol x** (terminal or non-terminal), which occurs in some items **after the analysis position**, a **transition** is created **to a successor state**. That contains a corresponding item with the **analysis position advanced behind the x** occurrence.



Operations of the LR(1) automaton

shift x (terminal or non-terminal):
from current state q
under x into the**successor state q‘** ,
push q‘

reduce p:
apply production p B ::= u ,
pop as many states,
as there are **symbols in u**, from the
new current state make a **shift with B**

error:
the current state has no transition
under the next input token,
issue a **message** and **recover**

stop:
recuce start production,
see # in the input

Example:

stack	input	reduction
1	(a ; a ; b ; b) #	
1 2	a ; a ; b ; b) #	
1 2 3	; a ; b ; b) #	p3
1 2	; a ; b ; b) #	
1 2 4	; a ; b ; b) #	
1 2 4 5	a ; b ; b) #	
1 2 4 5 6	; b ; b) #	p2
1 2	; b ; b) #	
1 2 4	; b ; b) #	
1 2 4 5	b ; b) #	
1 2 4 5 7	; b) #	
1 2 4 5 7 8	b) #	
1 2 4 5 7 8 7	) #	p5
1 2 4 5 7 8	) #	
1 2 4 5 7 8 9	) #	p4
1 2 4 5	) #	
1 2 4 5 10	) #	
1 2 3 5 10 11	#	p1
1	#	

LR conflicts

An **LR(1) automaton that has conflicts is not deterministic**. Its **grammar is not LR(1)**; correspondingly defined for any other LR class.

2 kinds of conflicts:

reduce-reduce conflict:
A state contains two reduce items, the
right context sets of which are **not disjoint**:

```
...
A ::= u .   R1
B ::= v .   R2
...
```

R1, R2 not disjoint

shift-reduce conflict:
A state contains
a **shift item** with the **analysis position in front of a t** and
a **reduce item with t in its right context set**.

```
...
A ::= u .t v   R1
B ::= w .      R2
...
```

t $\in$ **R2**

Shift-reduce conflict for „dangling else“ ambiguity

1

S ::= . Stmt	{#}
Stmt ::= . if ... then Stmt	{#}
Stmt ::= . if ... then Stmt else Stmt	{#}
Stmt ::= . a	{#}

Stmt →

a →

↓ if ... ↓ then

3

Stmt ::= if ... then . Stmt	{#}
Stmt ::= if ... then . Stmt else Stmt	{#}
Stmt ::= . if ... then Stmt	{# else}
Stmt ::= . if ... then Stmt else Stmt	{# else}
Stmt ::= . a	{# else}

Stmt →

a →

↓ if ... ↓ then

5

Stmt ::= if ... then . Stmt	{# else}
Stmt ::= if ... then . Stmt else Stmt	{# else}
Stmt ::= . if ... then Stmt	{# else}
Stmt ::= . if ... then Stmt else Stmt	{# else}
Stmt ::= . a	{# else}

if →

a →

↓ Stmt

6

Stmt ::= if ... then Stmt .	{# else}
Stmt ::= if ... then Stmt . else Stmt	{# else}

else →

shift-reduce conflict

Simplified LR grammar classes

LR(1):

too many states for practical use
Reason: right-contexts distinguish many states
Strategy: simplify right-contexts sets,
fewer states, grammar classes are less powerful

LR(0):

all items **without right-context**
Consequence: reduce items only in
singleton sets

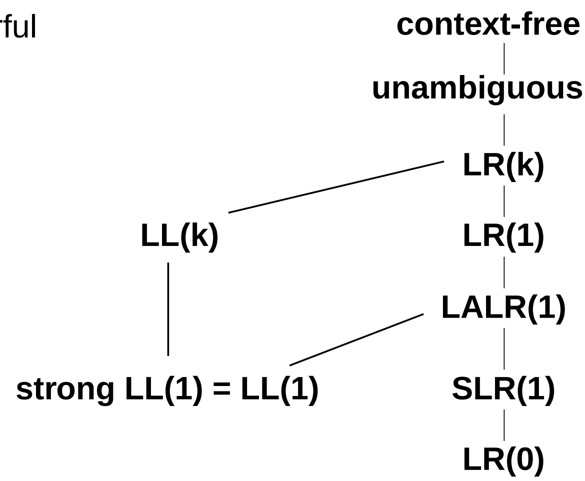
SLR(1):

LR(0) states; in reduce items
use larger right-context sets for decision:
[A ::= u . Follow (A)]

LALR(1):

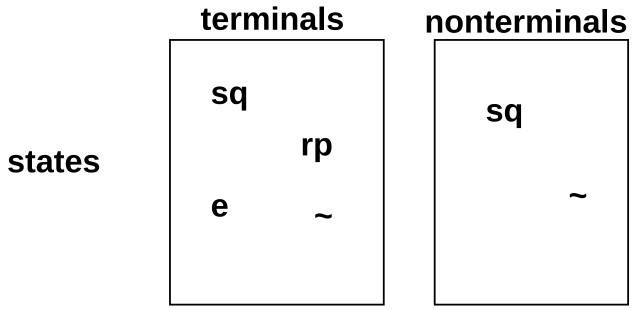
identify LR(1) states if their items differ only
in their right-context sets, unite the sets for those items;
yields the states of the **LR(0) automaton**
augmented by the "exact“ LR(1) right-context.
State-of-the-art parser generators accept LALR(1)

Grammar hierarchy:
(strict inclusions)



Implementation of LR automata

Table-driven:



sq: shift into state q

rp: reduce production p

e: error

~: never reached

Compress tables:

- **merge rows or columns** that differ only in irrelevant entries; method: graph coloring
- extract a **separate error matrix** (bit matrix); increases the chances for merging
- **normalize the values of rows or columns**; yields smaller domain; supports merging
- **eliminate LR(0) reduce states**; new operation in predecessor state: **shift-reduce** eliminates about 30% of the states in practical cases

About 10-20% of the original table sizes can be achieved!

Directly programmed LR-automata are possible **-** but usually too large.

Error handling: general criteria

- **recognize error as early as possible**
  LL and LR can do that
- **report the symptom in terms of the source text**
- **continue parsing short after the error position**
- **avoid avalanche errors**
- **build a tree that has a correct structure**
- **do not backtrack, do not undo actions**
- **no runtime penalty for correct programs**

Error position

Error recovery: Means that are taken by the parser after recognition of a syntactic error in order to continue parsing

Correct prefix: The token sequence w ∈ T* is a correct prefix in the language L(G), if there is an u ∈ T* such that **w u ∈ L(G)**; i. e. w can be extended to a sentence in L(G).

Error position: t is the (first) error position in the **input w t x** , where t ∈ T and w, x ∈ T*, if **w is a correct prefix** in L(G) and **w t is not a correct prefix**.

Example:

```
int compute (int i) { a = i * / c; return i;}
```

w (underlining `int compute (int i) { a = i *`), t (marking `/`)

LL and LR parsers recognize an error at the error position; they can not accept t in the current state.

Error recovery

Continuation point:
The token d at or behind the error position t such that **parsing of the input continues at d**.

Error repair
with respect to a consistent derivation - regardless the intension of the programmer!

Let the input be w t x with the error position at t and let w t x = w y d z, then the recovery (conceptually) **deletes y** and **inserts v**, such that **w v d is a correct prefix** in L(G), with d ∈ T and w, y, v, z ∈ T*.

Examples:

```
    w    y d  z
a = i *  / c;...
a = i *    c;...
```

delete /

```
    w    yd  z
a = i *  / c;...
a = i *e/ c;...
```

insert error id. e

```
    w     y d z
a = i *  / c;...
a = i * e  ;...
```

delete / c
and **insert** error id. **e**

Recovery method: simulated continuation

Problem: Determine a continuation point close to the error position and reach it.

Idea: Use parse stack to determine a set of tokens as potential continuation points.

Steps of the method:

1. **Save the contents of the parse stack** when an error is recognized. Skip the error token.
2. **Compute a set D $\subseteq$ T of tokens that may be used as continuation point** (**anchor set**)
   Let a modified parser run to completion:
   Instead of reading a token from input it is inserted into D; (modification given below)
3. **Find a continuation point d**: Skip input tokens until a token of D is found.
4. **Reach the continuation point d**:
   Restore the saved parser stack as the current stack.
   Perform dedicated transitions until d is acceptable.
   Instead of reading tokens (conceptually) insert tokens. Thus a correct prefix is constructed.
5. **Continue normal parsing**.

Augment parser construction for steps 2 and 4:
For each parser state select a transition and its token,
such that the parser empties its stack and terminates as fast as possible.
This selection can be **generated automatically**.
The quality of the recovery can be improved by influence on the computation of D.

Parser generators

PGS	Univ. Karlsruhe; in Eli	**LALR(1), table-driven**
Cola	Univ. Paderborn; in Eli	**LALR(1), optional: table-driven or directly programmed**
Lalr	Univ. / GMD Karlsruhe	**LALR(1), table-driven**
Yacc	Unix tool	**LALR(1), table-driven**
Bison	Gnu	**LALR(1), table-driven**
Llgen	Amsterdam Compiler Kit	**LL(1), recursive descent**
Deer	Univ. Colorado, Bouder	**LL(1), recursive descent**

Form of grammar specification:
EBNF: Cola, PGS, Lalr; **BNF**: Yacc, Bison

Error recovery:
simulated continuation, automatically generated: Cola, PGS, Lalr
error productions, hand-specified: Yacc, Bison

Actions:
statements in the implementation language
at the end of productions: Yacc, Bison
anywhere in productions: Cola, PGS, Lalr

Conflict resolution:
modification of states (reduce if ...) Cola, PGS, Lalr
order of productions: Yacc, Bison
rules for precedence and associativity: Yacc, Bison

Implementation languages:
C: Cola, Yacc, Bison **C, Pascal, Modula-2, Ada**: PGS, Lalr