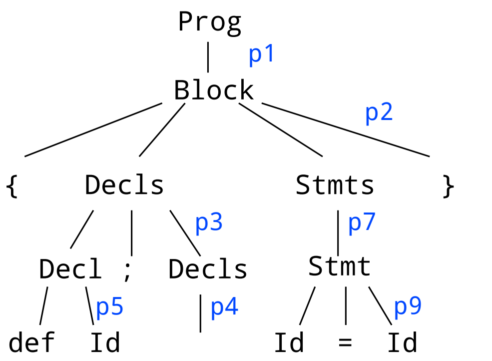
## LR Parsing

LR(k) parsing (Knuth, 1965) stands for reading from L eft-to-Right, constructing a R ightmost derivation in reverse using k input symbols of lookahead.

Only the cases k=0 and k=1 are of practical interest.




**Derivation Tree**

```
{ def Id ; Id = Id }       <==    p5
{ Decl ; Id = Id }         <==    p4
{ Decl ; Decls Id = Id }   <==    p3
{ Decls Id = Id }          <==    p9
{ Decls Stmt }             <==    p7
{ Decls Stmts }            <==    p2
Block                      <==    p1
Prog
```

**Bottom-Up, Reverse Rightmost Derivation**

The class of grammars that can be parsed using LR grammars is a proper superset of the class of grammars that can be parsed with LL methods.

Usually LR parsers are not constructed by hand but by using LR parser generators.





## LR(1) items

LR parsers are also called *Shift-Reduce Parsers* :

- they shift input symbols by pushing states onto the stack
- they reduce symbol sequences *uv* to Nonterminals *A* , according to productions *A ::= uv*

This process continues until an error is detected or the parser reduces to the start symbol and the input is empty.

LR states are represented by sets of so-called *items* consisting of

```
A ::= u.v    R
```

- a production.
- an analysis position, marked by a dot. If the dot is at the right end, the item is called *reduce item* .
- a right context R, a set of terminals which may follow in the input when the complete production is accepted.

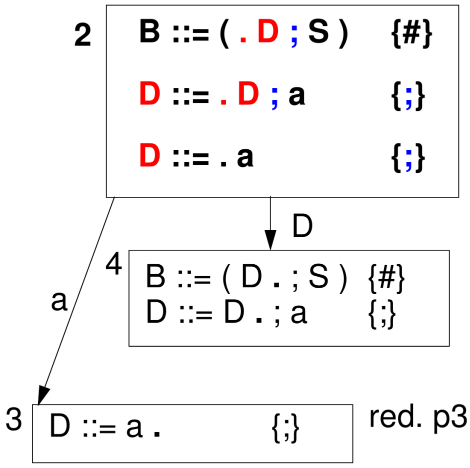
An item indicates how much has been seen of a production at a given point in the parsing process.

# 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 |

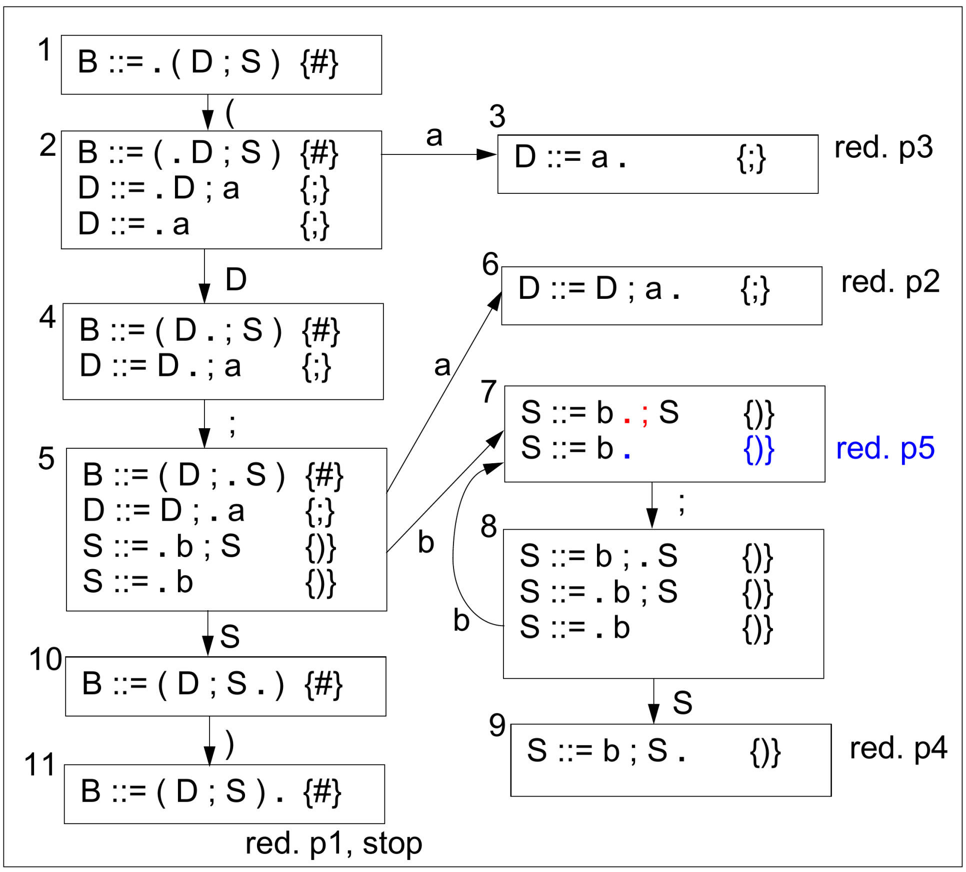




# Example LR(1) automaton

Grammar:

| | |
|---|---|
| p1 | B ::= ( D ; S ) |
| p2 | D ::= D ; a |
| p3 | D ::= a |
| p4 | S ::= b ; S |
| p5 | S ::= b |




In state 7 a decision is required on next input:

- if ; then shift
- if ) then reduce p5

In states 3, 6, 9, 11 a decision is not required:

- reduce on any input

## Operations of LR(1) Automata

**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:**
reduce 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 4 5 10 11 | # | p1 |
| 1 | # | |





## Table-driven Implementation of LR automata

LR Parser Tables

- Terminal Table "Action"
  - *shift* : si means shift and stack state i
  - *reduce* : rj means reduce by production j
  - *accept* : acc
  - *error* : blank entry
- Nonterminal Table "Goto"
  - n means push state n onto the stack

| STATE | ACTION | | | | | | GOTO | | |
|---|---|---|---|---|---|---|---|---|---|
| | **id** | + | * | ( | ) | $ | *E* | *T* | *F* |
| 0 | s5 | | | s4 | | | 1 | 2 | 3 |
| 1 | | s6 | | | | acc | | | |
| 2 | | r2 | s7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | s5 | | | s4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | s5 | | | s4 | | | | 9 | 3 |
| 7 | s5 | | | s4 | | | | | 10 |
| 8 | | s6 | | | s11 | | | | |
| 9 | | r1 | s7 | | r1 | r1 | | | |
| 10 | | r3 | r3 | | r3 | r3 | | | |
| 11 | | r5 | r5 | | r5 | r5 | | | |

**LR Parser for Expression Grammar (taken from ALSU, Compilers)**

```
p1: E ::= E '+' T        p4: T ::= F
p2: E ::= T              p5: F ::= '(' E ')'
p3: T ::= T '*' F        p6: F ::= id
```

## Construction of LR(1) Automata

**Algorithm**: 1. Create the start state.
2. For each created state compute the transitive closure of its items.
3. Create transitions and successor states as long as new ones can be created.

**Transitive closure** is to be applied to each state q:
Consider all items in q with the analysis position before a non-terminal B:
**[ $A_1$ ::= $u_1$ . B $v_1$ $R_1$ ] ... [ $A_n$ ::= $u_n$ . B $v_n$ $R_n$ ],**
then for each production **B ::= w**
**[ B ::= . w First ($v_1$ $R_1$)∪...∪First ($v_n$ $R_n$)]**
has to be added to state q.

*before:* 2

```
B ::= ( . D ; S )  {#}
```

*after:* 2

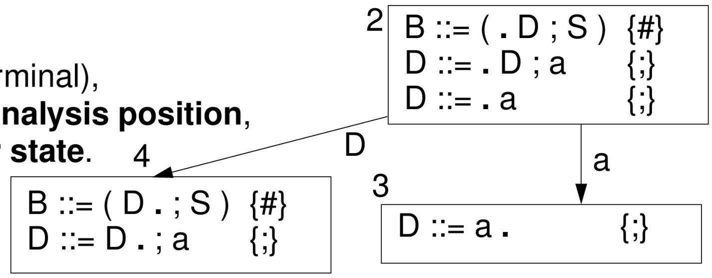
```
B ::= ( . D ; S )  {#}
D ::= . D ; a      {;}∪{;}
D ::= . a          {;}∪{;}
```

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

1

```
B ::= . ( D ; S )  {#}
```

**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 corresponding items with the **analysis position advanced behind the x** occurrence.








## 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
...
```

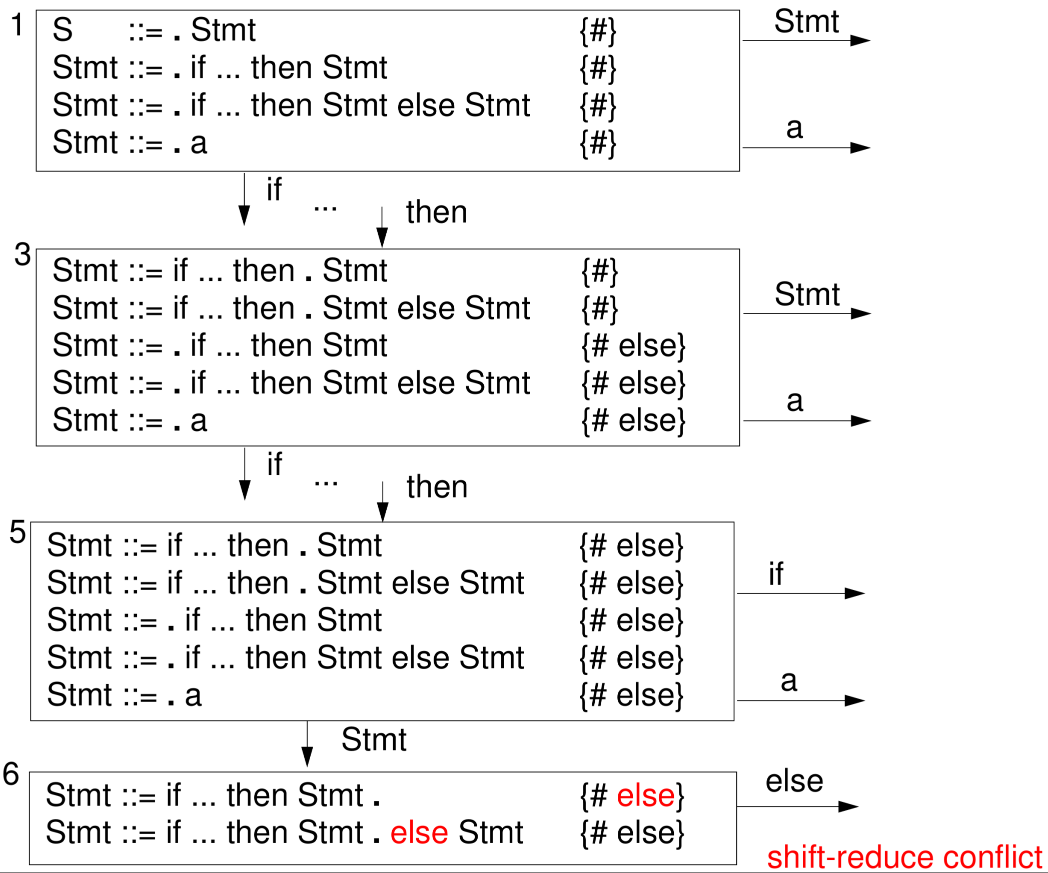
**R1, R2 not disjoint**

**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** ∈ **R2**

## Shift-Reduce Conflict for "Dangling Else" Ambiguity

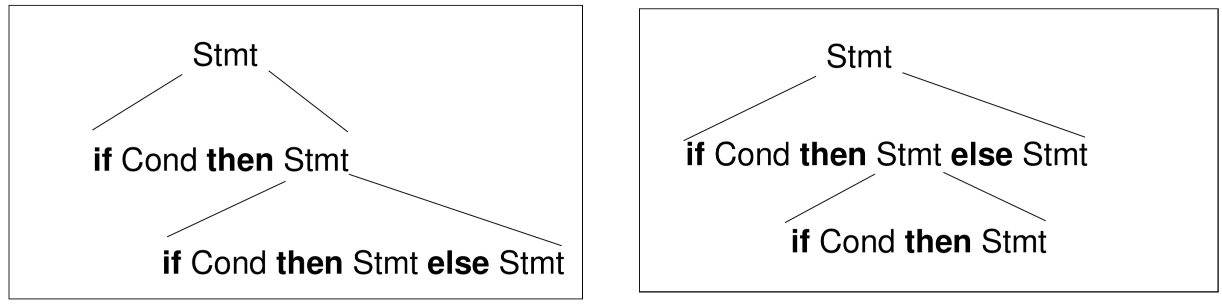




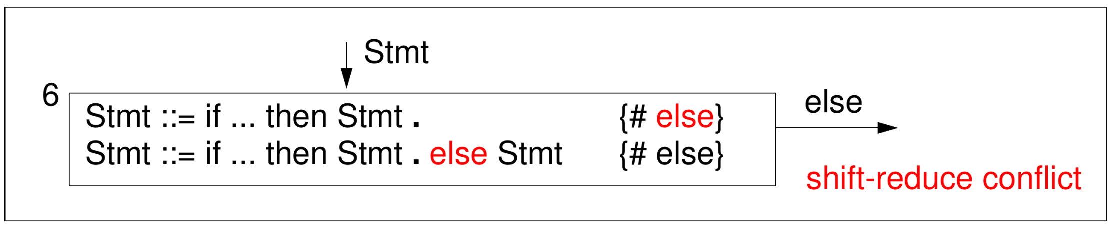



## Decision of Ambiguity

dangling else ambiguity:




desired solution for Pascal, C, C++, Java




State 6 of the automaton can be modified such that
an input token **else is shifted** (instead of causing a reduction);
yields the desired behaviour.

Some parser generators allow such modifications.

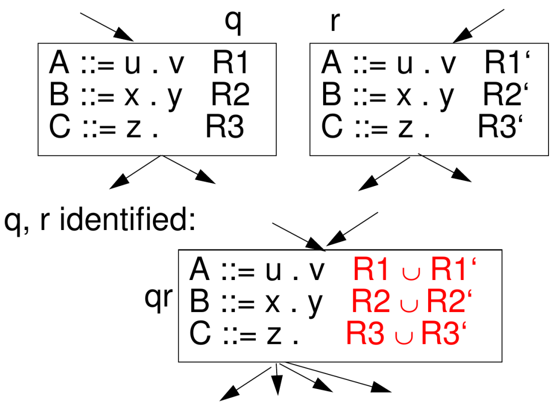
# Simplified LR Grammar Classes

**LR(1):**
**too many states** for practical use**,** because right-contexts distinguish many states.
**Strategy:** simplify right-contexts sets; **fewer states**; grammar classes less powerful

**LALR(1):**
construct LR(1) automaton,
**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)**




**SLR(1):**
**LR(0) states**; in reduce items
use larger right-context sets for decision:
**[ A ::= u . Follow (A) ]**

**A ::= u . v**
**B ::= x . y**
**C ::= z .** **Follow(C)**

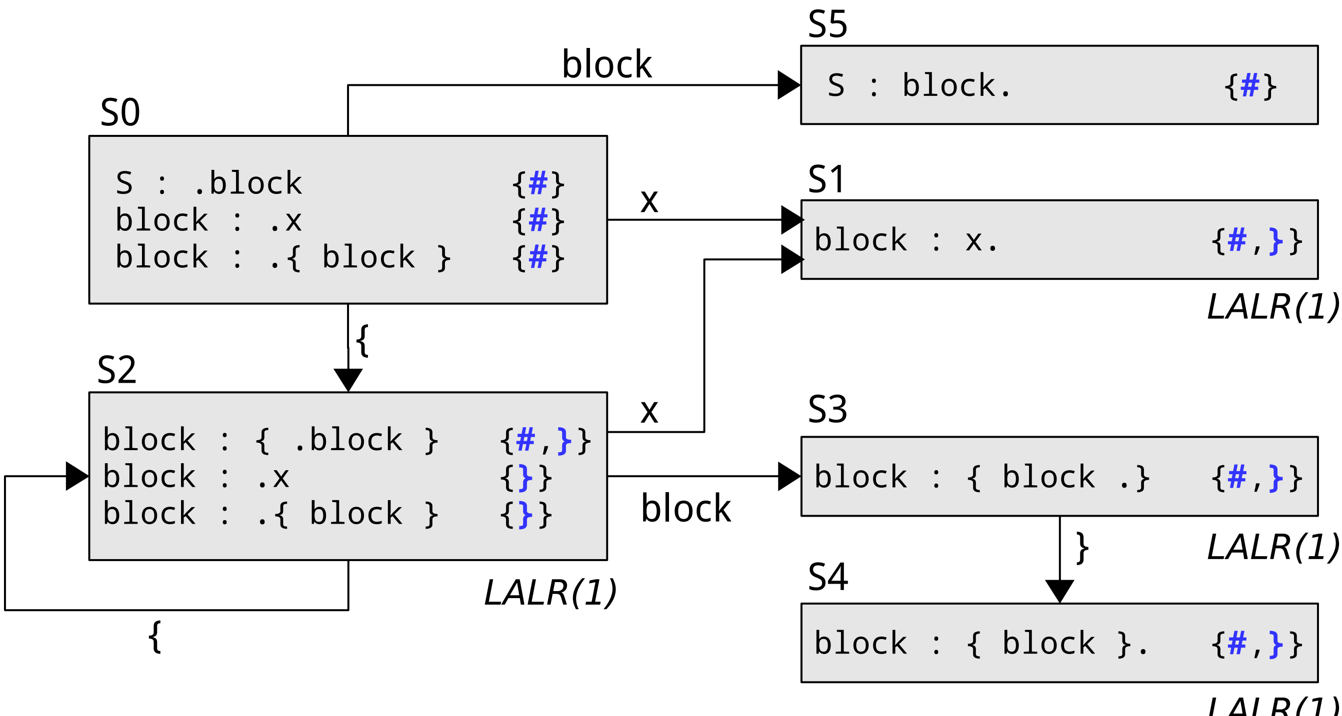
**LR(0):**
all items **without right-context**
**Consequence:** reduce items only in singleton sets

**C ::= z .**



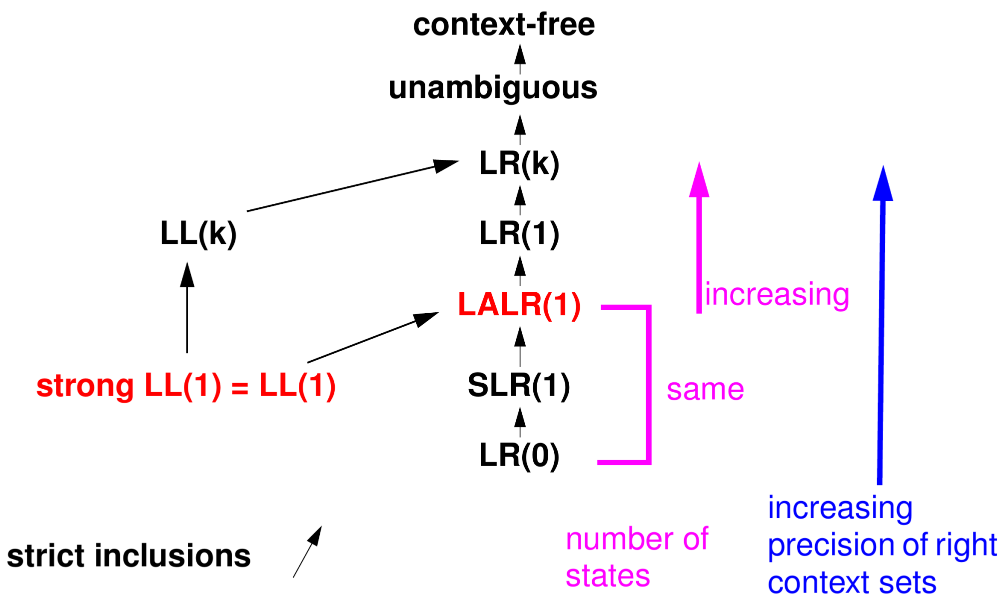


# LALR(1) Automaton for Nested Block Language




*States marked „LALR(1)“ show merged lookahead sets.*

## Grammar Class Hierarchy

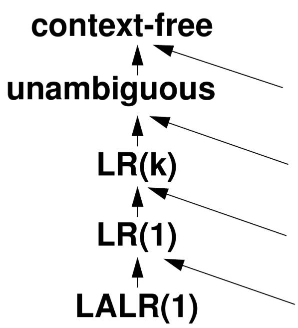







## Reasons for LALR(1) Conflicts

Grammar condition does not hold:




| | |
|---|---|
| **ambiguous** | most cases |
| **unbounded lookahead** needed | |
| **fixed length lookahead > 1** needed | |
| **merge of LR(1) states** introduces **conflicts** | rare cases |

LALR(1) parser generator can not distinguish these cases.

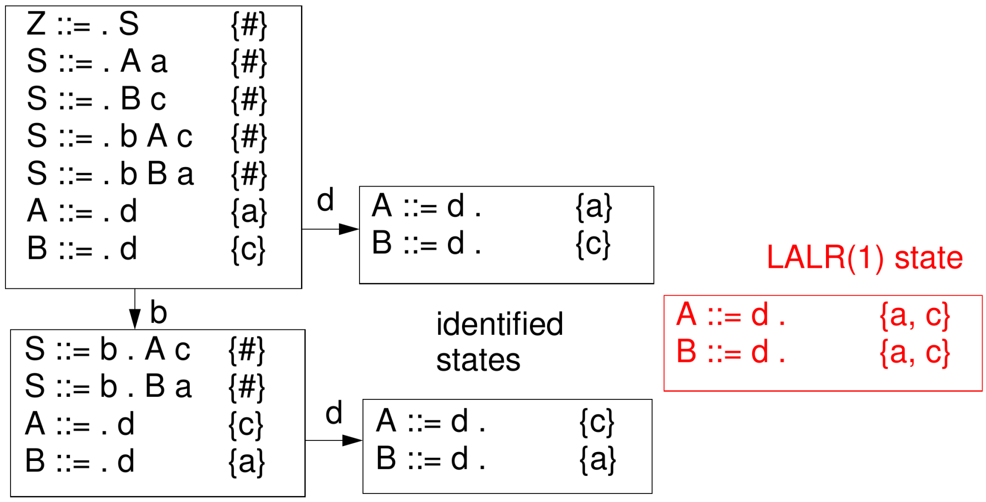
## LR(1) but not LALR(1)

**Identification of LR(1) states** causes non-disjoint right-context sets.

Artificial example:

Grammar:
Z ::= S
S ::= A a
S ::= B c
S ::= b A c
S ::= b B a
A ::= d.
B ::= d.

LR(1) states




Avoid the distinction between A and B - at least in one of the contexts.





## Syntax Error Handling

### General criteria

- **recognize error as early as possible**
  LL and LR can do that:
  no transitions after error position
- **report the symptom in terms of the source text**
  rather than in terms of the state of the parser
- **continue parsing short after the error position**
  analyze as much as possible
- **avoid avalanche errors**
- **build a tree that has a correct structure**
  later phases must not break
- **do not backtrack, do not undo actions,**
  not possible for semantic 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 $w \in T^*$ is a correct prefix in the language L(G), if there is an $u \in T^*$ such that **w u** $\in$ **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 $t \in T$ and $w, x \in T^*$, 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 _______________/ |
                                  t
```

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





# Error recovery

**Continuation point**:
A 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 $d \in T$ and $w, y, v, z \in T^*$.

error position

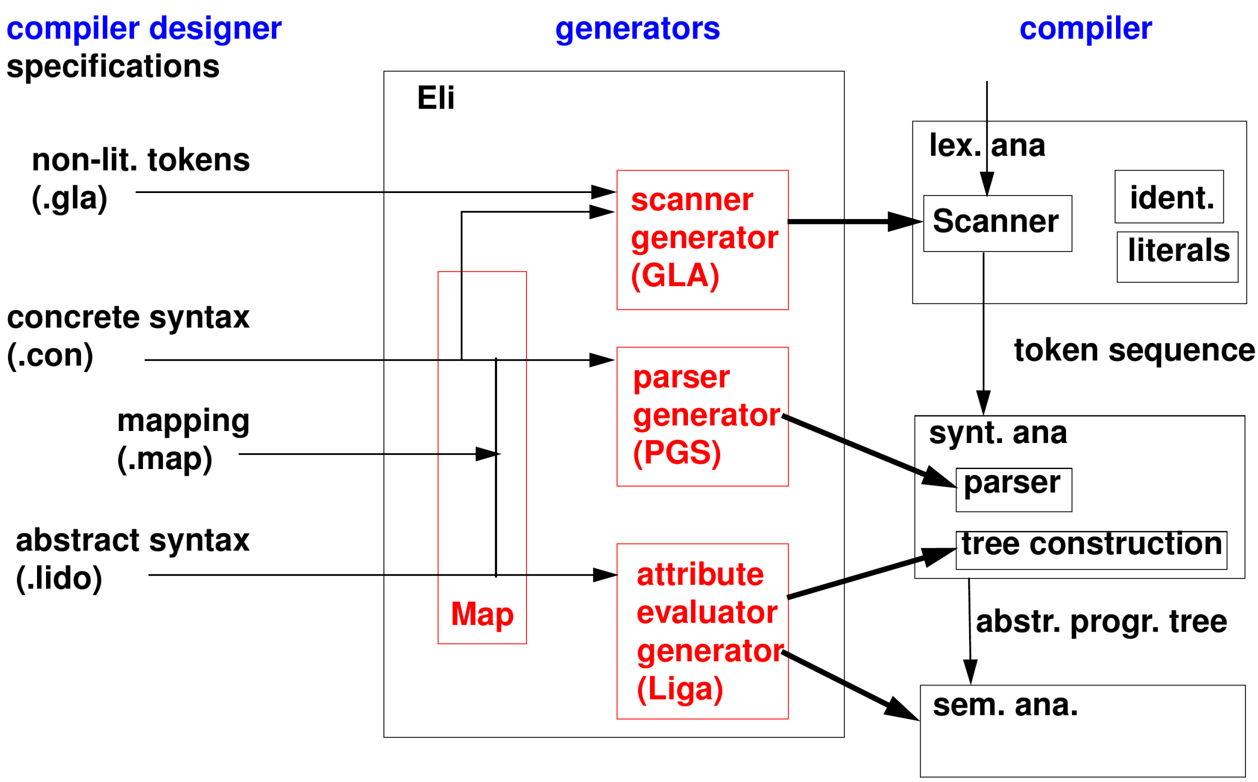
w t x =
w y d z
w v d z

continuation

**Examples:**

| w y d z | w yd z | w y d z |
|---|---|---|
| `a = i * / c;...` | `a = i * / c;...` | `a = i * / c;...` |
| `a = i *   c;...` | `a = i *e/ c;...` | `a = i * e  ;...` |
| **delete** / | **insert** error identifier e | **delete** / c and **insert** error id. e |

## Generating the Structuring Phase








## Parser Generators

Parser generators generate the central function of syntax analysis from the concrete syntax specification and support structure tree construction according to the abstract syntax, e.g. by adding *Semantic Actions* :

```
p9: Stmt ::= Id '=' Id &'mknode(p9)'
```

- YACC / Bison
  - standard Unix tool and its improved GNU version
  - LALR(1) parsers implemented in C/C++
  - Arbitrary C-Code as semantic actions
- PGS / Cola (Generator for Lexical Analysis)
  - University of Karlsruhe / Paderborn
  - Part of the Eli system, interfaces with other components
  - LALR(1) parsers implemented in C/C++
  - AST construction automatically provided by Eli
- Coco/R
  - University of Linz
  - LL(1) recursive descent parsers in C, Java, Pascal, Python, ...
- ANTLR v3/v4
  - University of San Francisco
  - LL(*), Adaptive LL(*) parsers in (mainly) Java
- Many, many others