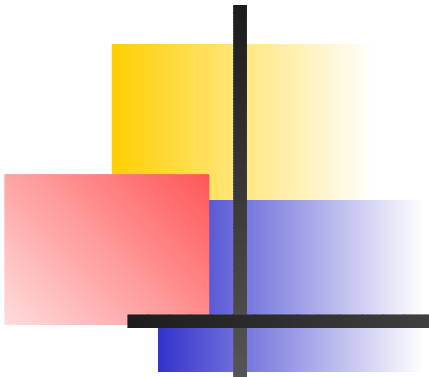


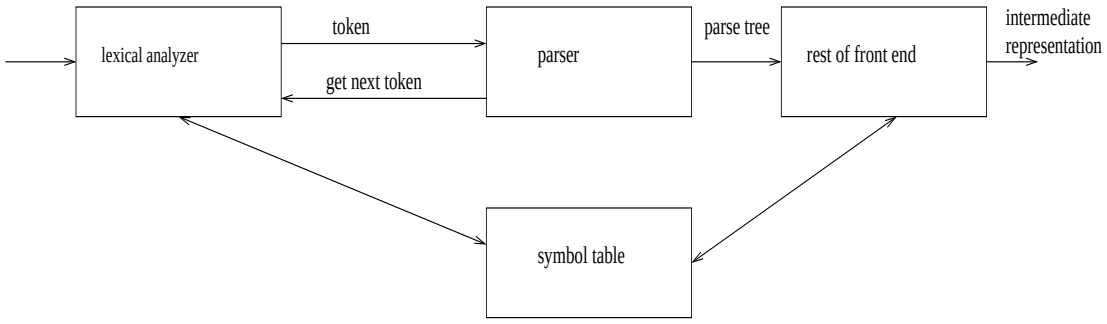
Compilers

Syntax Analysis

SITE : http://www.info.univ-tours.fr/~mirian/

The Role of the Parser

- The parser obtains a string of tokens from the lexical analyser and verifies that the string can be generated by the grammar for the source language
- The parser finds a derivation of a given sentence using the grammar or reports that none exists.
- The parser should
  - report any syntax errors in an intelligible fashion
  - recover from commonly occurring errors so that it can continue processing the reminder of its input
- Output of the parser: some representation of a parse tree for the stream of tokens produced by the lexical analyser



The parsing problem

- **Consists of finding a derivation (if one exists) of a particular sentence using a given grammar**
- Picture:
  Considering that we have:
  - a sentence symbol (*e.g.*, at the top of a sheet of paper) and
  - a sentence to be analysed (at the bottom of the sheet of paper)

  the parsing problem consists of drawing a syntax tree (parse tree) to join the sentence symbol and the sentence.

Types of parsers

- A general algorithm for syntax analysis of CFG has a high cost in terms of time complexity: $O(n^3)$
- We need grammar classes allowing syntax analysis to be **linear**
- In this context, there are two ways to build parse trees:

1. **Top-down**: build the parse trees from the top (root) to the bottom (leaves)
   - We have to decide which rule $A \longrightarrow \beta$ should be applied to a node labelled A
   - Expanding A results in new nodes (children of A) labelled by symbols in β
2. **Bottom-up**: start from the leaves and work up to the root
   - We have to decide when rule $A \longrightarrow \beta$ should be applied and we should find neighbour nodes labelled by symbols in β
   - Reducing rule $A \longrightarrow \beta$ results in adding a node A to the tree. A's children are the nodes labelled by the symbols in β.

In both cases, the input to the parser is scanned from left to right, one symbol at a time.

Examples

We consider the grammar $G_1 = (\{E, F, T\}, \{a, (,), *, +\}, P, E)$ where the productions are:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
E & \rightarrow & T \\
T & \rightarrow & T * F \\
T & \rightarrow & F \\
F & \rightarrow & (E) \\
F & \rightarrow & a
\end{array}
$$

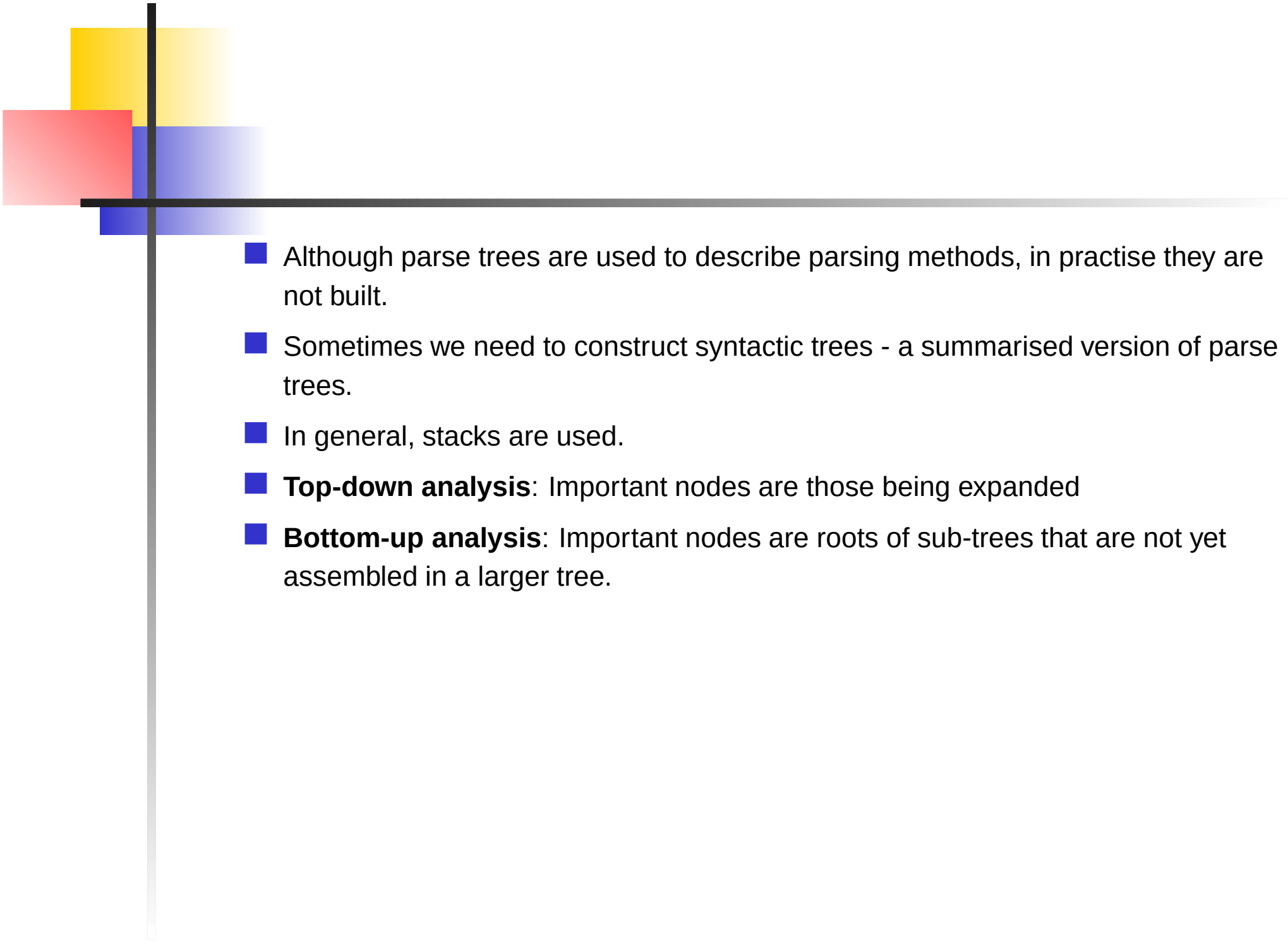
and the string $x = a + a * a$

Top-down method: rules are considered in the same order as a leftmost derivation

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow a + T \Rightarrow a + T * F \Rightarrow a + F * F \Rightarrow a + a * F \Rightarrow a + a * a$

Bottom-up method: rules are considered in the same order as a *reverse* rightmost derivation

$a + a * a \Leftarrow F + a * a \Leftarrow T + a * a \Leftarrow E + a * a \Leftarrow E + F * a \Leftarrow E + T * a \Leftarrow$
$E + T * F \Leftarrow E + T \Leftarrow E$



- Although parse trees are used to describe parsing methods, in practise they are not built.
- Sometimes we need to construct syntactic trees - a summarised version of parse trees.
- In general, stacks are used.
- **Top-down analysis**: Important nodes are those being expanded
- **Bottom-up analysis**: Important nodes are roots of sub-trees that are not yet assembled in a larger tree.

Top-down analysis: the use of stacks

- The process is represented by configurations (α, y) where
  α is the content of the stack and
  y is the rest of the input, not analysed yet.
- The top of the stack is on the left
- There are two kinds of transitions from one configuration to another
  1. Expanding a non terminal by using production $A \longrightarrow \beta$
     Changes configuration $(A\alpha, y)$ to $(\beta\alpha, y)$
  2. Verifying a terminal a
     Changes configuration $(a\alpha, ay)$ to (α, y)
     Used to *pop* an element from the stack (to find the next non terminal to be expanded)
- Initial configuration: (S, x) for an input string x
- Final configuration: (ϵ, ϵ).
  The stack is empty and the input has been completely considered

Example: How to choose the production rule to be applied?

Stack	Rest of the input	Leftmost derivation
E	$a + a * a$	E
$E + T$	$a + a * a$	$\Rightarrow E + T$
$T + T$	$a + a * a$	$\Rightarrow T + T$
$F + T$	$a + a * a$	$\Rightarrow F + T$
$a + T$	$a + a * a$	$\Rightarrow a + T$
$+T$	$+a * a$	
T	$a * a$	
$T * F$	$a * a$	$\Rightarrow a + T * F$
$F * F$	$a * a$	$\Rightarrow a + F * F$
$a * F$	$a * a$	$\Rightarrow a + a * F$
$*F$	$*a$	
F	a	
a	a	$\Rightarrow a + a * a$
ϵ	ϵ	

Bottom-up analysis: the use of stacks

- The process is represented by configurations (α, y) where
  α is the content of the stack and
  y is the rest of the input, not analysed yet.
- The top of the stack is on the right
- There are two kinds of transitions from one configuration to another
  1. Reduction by using production $A \rightarrow \beta$
     Changes configuration $(\alpha\beta, y)$ to $(\alpha A, y)$
  2. Putting terminal a in the stack
     Changes configuration (α, ay) to $(\alpha a, y)$
     Used to *push* an element to the stack (allowing them to take part in the reductions that take place on the top of the stack)
- Initial configuration: (ϵ, x) for an input string x
- Final configuration: (S, ϵ).
  Indicates that all the input has been read and reduced for S

Example: How to choose the production rule to be applied?

Stack	Rest of the input	Rightmost derivation (in reverse)
	$a + a * a$	$a + a * a$
a	$+a * a$	
F	$+a * a$	$\Leftarrow F + a * a$
T	$+a * a$	$\Leftarrow T + a * a$
E	$+a * a$	$\Leftarrow E + a * a$
$E+$	$a * a$	
$E + a$	$*a$	
$E + F$	$*a$	$\Leftarrow E + F * a$
$E + T$	$*a$	$\Leftarrow E + T * a$
$E + T*$	a	
$E + T * a$	ϵ	
$E + T * F$	ϵ	$\Leftarrow E + T * F$
$E + T$	ϵ	$\Leftarrow E + T$
E	ϵ	$\Leftarrow E$

Writing a Grammar

- Grammars are capable of describing most, but not all, of the syntax of programming languages
- Certain constraints on the input, such as the requirement that identifiers be declared before being used, cannot be described by a CFG
- Because each parsing method can handle grammars only of a certain form, the initial grammar may have to be rewritten to make it parsable by the method chosen
  - Eliminate ambiguity
  - Recursion
    1. Left recursive grammar: It has a non terminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α
    2. Right recursive grammar: It has a non terminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} \alpha A$ for some string α

    Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed.

Backtracking and Predictive Analysers

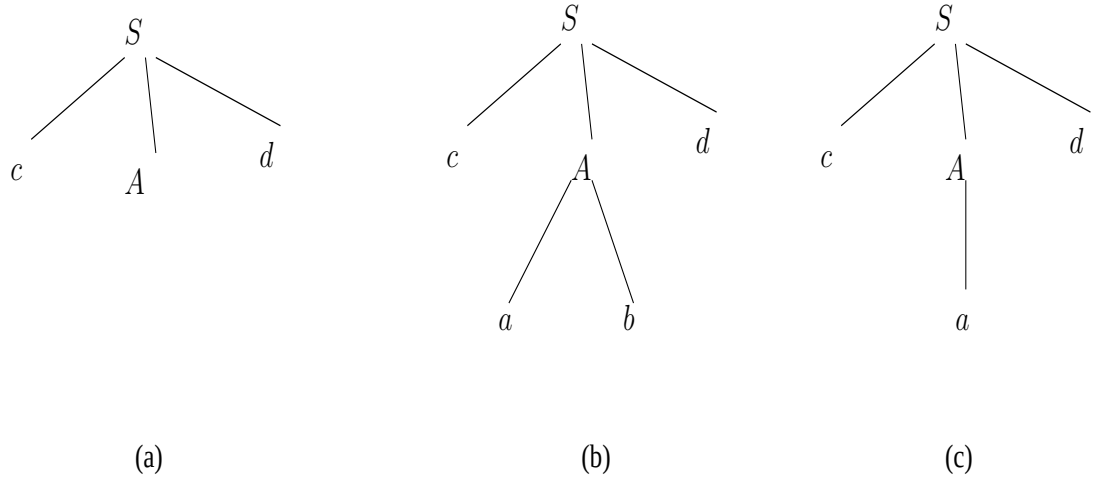
- Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string
- **Recursive descent**: a general form of top-down parsing that may involve *backtracking*, *i.e.*, making repeated scans of the input
  Backtracking parser are not seen frequently
- Backtracking is required in some cases

Example

Consider the grammar G_2 below and the string $w = cad$

$$S \longrightarrow cAd$$

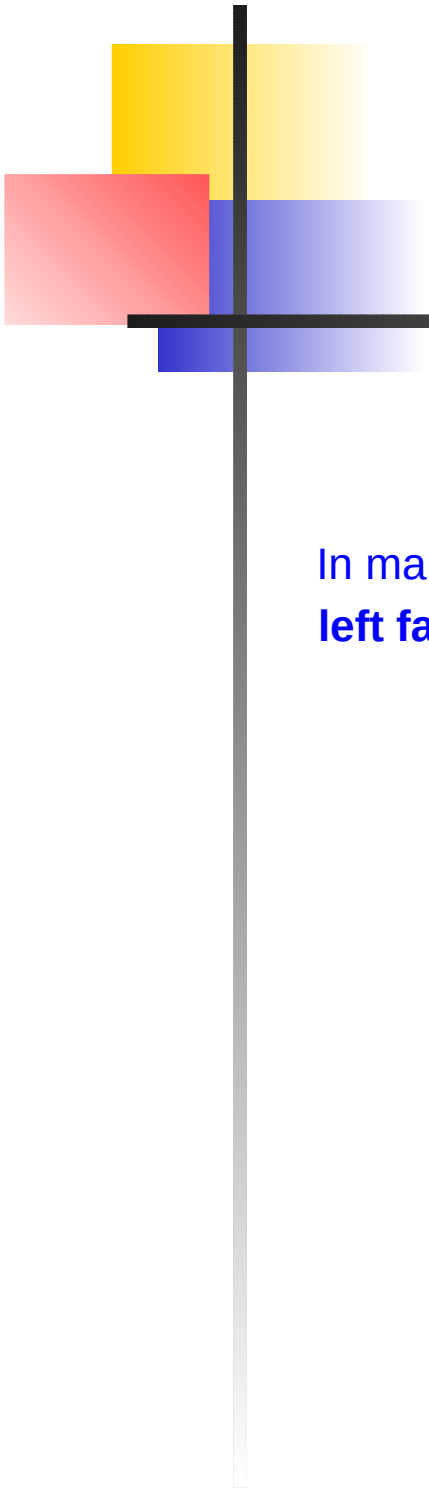
$$A \longrightarrow ab \mid a$$



(a) (b) (c)

Example: To build the parse tree for w top-down

1. Create a tree consisting of a single node S
2. An input pointer points to c (1st symbol of w)
3. Use the 1st production for S to expand the tree (figure (a))
4. Leftmost leaf (labelled c) matches the 1st symbol of w. Thus advance the input pointer to a
5. Expand A with its first alternative (figure (b))
6. We have a match for the second input symbol and we advance the input pointer to d
7. Compare the third input symbol d against the next leaf b. NO MATCH!
8. Report the failure and go back to A (looking for another alternative of expansion)
9. In going back to A, we must reset the input pointer in position 2
10. Try the second alternative for A: the leaf a matches the second input symbol of w and the leaf d matches the third symbol (figure (c)).



Re-writting grammars

In many cases, by carefully writting a grammar, **eliminating left recursion from it, and left factoring the resulting grammar**, we can obtain a grammar that can be parsed by **a recursive parser that needs no backtracking**

Simple left recursion

- The pair of production

$$A \rightarrow A\alpha \mid \beta$$

can be replaced by the non-left-recursive productions

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

- No matter how many A-production there are, we can eliminate immediate left recursion:
We group the A productions as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \ldots \mid \beta_n$$

where no β_i begins with an A.
Then we replace the A-productions by

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \alpha_m A' \mid \epsilon \end{aligned}$$

Left Factoring

- When it is not clear which of the two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice

  stmt $\rightarrow$ **if** expr **then** stmt **else** stmt

  if expr **then** stmt

- In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input begins with a non empty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$.

- We may defer the decision by expanding A to $\alpha A'$. Then after seeing the input derived from α, we expand A' to β_1 or to β_2

  $A \rightarrow \quad \alpha A'$

  $A' \rightarrow \quad \beta_1 \mid \beta_2$

Discussion

- A left-recursive grammar can cause a recursive-descent parser (even one without backtracking) to go to infinite loops
  When we try to expand A, we may eventually find ourselves again trying to expand A without having consumed any input
- A general type of analysis, one capable of treating all kinds of grammars, is not efficient
- By carefully writing a grammar, we can obtain a grammar that can be parsed by a recursive-descent parser that needs no backtracking
- *Predictive parser*: to build one, we must know:
  - Given the current input symbol a and the non terminal A to be expanded, the proper alternative (the good production rule) must be detectable by looking at only the first symbol it derives
- To have linear time complexity we cannot use backtracking
- The grammars that can be analysed by a top-down predictive parser are called LL(1)

Top-down and Bottom-up analysis

NOTATION

- LL(n) Analysis
- LR(n) Analysis

where
L: Left to right. The input string is analysed from left to right
L: Leftmost. Uses the leftmost derivation
R: Rightmost. Uses the rightmost derivation
n : the number of input symbols we need to know in order to perform the analysis
Example: LL(2) is a grammar having the following characteristics:

- strings are analysed from the left to the right
- derivation of the leftmost non terminal in the parse tree
- knowledge of two tokens in order to choose the production rule to apply

Getting information about grammars

To decide which production rules to use during the syntax analysis, three kinds of information about non terminals of CFG are requested:
For a non terminal A we want to know:

1. Whether A generates the *empty string*
2. The set of terminal symbols *starting* strings generated from A
   If $A \stackrel{*}{\Rightarrow} \alpha$ which terminals can appear as first symbol of α?
3. The set of terminal symbols *following* A
   If $S \stackrel{*}{\Rightarrow} \alpha A \beta$ which terminals can appear as first symbols of β?

Remark: The algorithms we are going to present can be used to grammars that do not have useless symbols or rules

Non terminals that derive the empty string

Input: A grammar G

Output: Non terminals that generate ϵ are marked *yes*;
otherwise they are marked *no*

Algorithm 1:

Non terminals that derive the empty string

If G does not have any production of the form $A \rightarrow \epsilon$ (for some non terminal A)
then all non terminals are marked with *no*
else apply the following steps until no new information can be generated

1. $L =$ list of all productions of G but those having one terminal at right-hand side
   Productions whose right-hand side have a terminal do not derive ϵ
2. For each non terminal A without productions, mark it with *no*
3. While there is a production $A \rightarrow \epsilon$:
   - delete from L all productions with A at the left-hand side;
   - delete every occurrence of A in the right-hand side of the productions in L;
   - mark A with *yes*

   Remark: The production $B \rightarrow C$ is replaced by $B \rightarrow \epsilon$ if the occurrence of C in its right-hand side is deleted.

Computing $FIRST(A)$ for non terminal A - Starting terminal symbols

Let $G = (V, T, P, S)$ be a CFG. Formally, we define the set of starting terminal symbols of a non terminal $A \in V$ as:

$$FIRST(A) = \{a \mid a \text{ is a terminal and } A \stackrel{*}{\Rightarrow} a\alpha, \text{ for some string } \alpha \in (V \cup T)^*\}$$

Computing $FIRST(A)$: Basic points

- If there is a production $A \rightarrow a\ \alpha$
  then $a \in FIRST(A)$.
  The corresponding derivation is $A \Rightarrow a\alpha$
- If there is a production $A \rightarrow B_1 B_2 \ldots B_m\ a\ \alpha$ and $B_i \stackrel{*}{\Rightarrow} \epsilon$ for $1 \leq i \leq m$
  then $a \in FIRST(A)$.
  In this case, a becomes the first symbol after the replacement of all B_i by ϵ.
  The corresponding derivation is $A \Rightarrow B_1 B_2 \ldots B_m a\alpha \stackrel{*}{\Rightarrow} a\alpha$
- If there is a production $A \rightarrow B\ \alpha$ and if $a \in FIRST(B)$
  then $a \in FIRST(A)$.
  If $a \in FIRST(B)$, we have $B \stackrel{*}{\Rightarrow} a\beta$ and the corresponding derivation is
  $A \Rightarrow B\alpha \stackrel{*}{\Rightarrow} a\beta\alpha$
- If there is a production $A \rightarrow B_1 B_2 \ldots B_m\ C\ \alpha$ and $B_i \stackrel{*}{\Rightarrow} \epsilon$ for $1 \leq i \leq m$
  then if $a \in FIRST(C)$
  then we also have $a \in FIRST(A)$.
  In this case, all B_i are replaced by ϵ.
  If $C \stackrel{*}{\Rightarrow} a\beta$, the corresponding derivation is $A \Rightarrow B_1 B_2 \ldots B_m C\alpha \Rightarrow C\alpha \stackrel{*}{\Rightarrow} a\beta\alpha$

Computing $FIRST(A)$: Algorithm

Input: A grammar G

Output: The set of starting terminal symbols for all non terminals of G

Algorithm 2:

Computation of $FIRST(A)$ for all non terminal A

1. For all non terminals A of G, $FIRST(A) = \emptyset$
2. Apply the following rules until no more terminals can be added to any $FIRST$ set:
   (a) For each rule
   $A \rightarrow a\alpha$
   put a in $FIRST(A)$
   (b) For each rule
   $A \rightarrow B_1 \ldots B_m$
   If for some i ($1 \leq i \leq m$) we have $a \in FIRST(B_i)$ and $B_1, \ldots, B_{i-1} \stackrel{*}{\Rightarrow} \epsilon$ then put a in $FIRST(A)$
   For example, everything in $FIRST(B_1)$ is surely in $FIRST(A)$. If B_1 does not derive ϵ, then we add nothing more to $FIRST(A)$, but if $B_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $FIRST(B_2)$ and so on.

Starting symbols of a string

Given a grammar G, we also need to introduce the concept of the set of starting terminal symbols for a string $\alpha \in (V \cup T)^*$, $FIRST(\alpha)$:

- If $\alpha = \epsilon$
  then $FIRST(\alpha) = FIRST(\epsilon) = \emptyset$
- If α is a terminal a
  then $FIRST(\alpha) = FIRST(a) = \{a\}$
- If α is a non terminal
  then $FIRST(\alpha)$ computed by Algorithm 2
- If α is a string $A\beta$ and A is a non terminal that derives ϵ
  then $FIRST(\alpha) = FIRST(A\beta) = FIRST(A) \cup FIRST(\beta)$
- If α is a string $A\beta$ and A is a non terminal that does not derive ϵ
  then $FIRST(\alpha) = FIRST(A\beta) = FIRST(A)$
- If α is a string $a\beta$ where a is a terminal
  then $FIRST(\alpha) = FIRST(a\beta) = \{a\}$

Computing $FOLLOW(A)$ for non terminal A - Following terminal symbols

Let $G = (V, T, P, S)$ be a CFG.

- A non terminal can appear at the end of a string and, in this case, it does not have a following symbol
- Due to this fact and to treat this case as the others, we introduce a new terminal symbol $\$$ to indicate the end of a string
- $\$$ is a right endmarker and corresponds to the end of a file
- This symbol is introduced as a production of S

Formally, we define the following symbols of a non terminal $A \in V$ as:

$$FOLLOW(A) = \{a \mid a \text{ is a terminal and } S \stackrel{*}{\Rightarrow} \alpha\ A\ a\ \beta, \text{ for strings } \alpha, \beta \in (V \cup T)^*\}$$

Computing $FOLLOW(A)$: Basic points

We suppose that $S\$ \stackrel{*}{\Rightarrow} \delta A \psi$. Put $\$$ in FOLLOW(S)

- If there is a production $A \rightarrow \alpha\ B\ \gamma\ a\ \beta$
  where $\gamma = B_1, \ldots, B_m$ is a non terminal string deriving ϵ, *i.e.*, $\gamma \stackrel{*}{\Rightarrow} \epsilon$
  then $a \in FOLLOW(B)$.
  The corresponding derivation is: $S\$ \stackrel{*}{\Rightarrow} \delta\ \underline{A}\ \psi \stackrel{*}{\Rightarrow} \delta\ \underline{\alpha\ B\ \gamma\ a\ \beta}\ \psi \stackrel{*}{\Rightarrow} \delta\ \alpha\ B\ a\ \beta\ \psi$
- If there is a production $A \rightarrow \alpha\ B\ \gamma\ C\ \beta$
  where $\gamma = B_1, \ldots, B_m$ is a non terminal string deriving ϵ, *i.e.*, $\gamma \stackrel{*}{\Rightarrow} \epsilon$
  and if $a \in FIRST(C)$
  then we have $a \in FOLLOW(B)$
  In this case, $C \stackrel{*}{\Rightarrow} a\mu$ and thus, the corresponding derivation is:
  $S\$ \stackrel{*}{\Rightarrow} \delta\ \underline{A}\ \psi \stackrel{*}{\Rightarrow} \delta\ \underline{\alpha\ B\ \gamma\ C\ \beta}\ \psi \stackrel{*}{\Rightarrow} \delta\ \alpha\ B\ \underline{C}\ \beta\ \psi \stackrel{*}{\Rightarrow} \delta\ \alpha\ B\ \underline{a\ \mu}\ \beta\ \psi$
- If there is a production $A \rightarrow \alpha\ B\ \gamma$
  where $\gamma = B_1, \ldots, B_m$ is a non terminal string deriving ϵ, *i.e.*, $\gamma \stackrel{*}{\Rightarrow} \epsilon$
  and if $a \in FOLLOW(A)$
  then we have $a \in FOLLOW(B)$
  In this case, $S\$ \stackrel{*}{\Rightarrow} \delta\ A\ a\ \psi$ and thus, the corresponding derivation is:
  $S\$ \stackrel{*}{\Rightarrow} \delta\ \underline{A}\ a\ \psi \stackrel{*}{\Rightarrow} \delta\ \underline{\alpha\ B\ \gamma}\ a\ \psi \stackrel{*}{\Rightarrow} \delta\ \alpha\ B\ a\ \psi$

$FOLLOW(A)$: algorithm

Input: A grammar G

Output: The set of following terminal symbols for all non terminals of G

Algorithm 3:

Computation of $FOLLOW(A)$ for all non terminal A

1. For all non terminals $A \neq S$ of G, $FOLLOW(A) = \emptyset$. $FOLLOW(S) = \{\$\}$
2. Apply the following rules until nothing can be added to any FOLLOW set
   (a) For each production rule $A \rightarrow \alpha B \beta$
   Put every terminal of $FIRST(\beta)$ in $FOLLOW(B)$
   (b) For each production rule $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \gamma$ such that $\gamma = B_1 \ldots B_m \stackrel{*}{\Rightarrow} \epsilon$
   Put every terminal of $FOLLOW(A)$ in $FOLLOW(B)$

Construction of Predictive Parsing Tables

- We can consider the problem of how to choose the production to be used during the syntax analysis
- To this end we use a *predictive parsing table* for a given grammar G
- To build this table we use the following algorithm whose main ideas are:
  1. Suppose $A \rightarrow \alpha$ is a production with a in $FIRST(\alpha)$.
     Then, the parser will expand A by α when the current input symbol is a
  2. Complication occurs when $\alpha = \epsilon$ or $\alpha \stackrel{*}{\Rightarrow} \epsilon$. In this case, we should again expand A by α if the current input symbol is in $FOLLOW(A)$

Algorithm: Construction of a predictive parsing table

Input: Grammar G

Output: Parsing table M

Algorithm: Construction of a predictive parsing table

1. For each production $A \rightarrow \alpha$ of G, do steps 2 and 3
2. For each terminal a in $FIRST(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
3. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $FOLLOW(A)$
   If $\alpha \stackrel{*}{\Rightarrow} \epsilon$ and \$ is in $FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$
4. Make each undefined entry of M be **error**

Example : Syntax Analysis

Grammar $G_3 = (\{E, E', T, T', F\}, \{a, *, +, (,)\}, P, E)$ with the set of productions P:

$$
\begin{array}{lcl@{\qquad}lcl}
E & \rightarrow & TE' & T & \rightarrow & FT' \\
F & \rightarrow & (E) & E' & \rightarrow & +TE' \\
T' & \rightarrow & *FT' & F & \rightarrow & a \\
E' & \rightarrow & \epsilon & T' & \rightarrow & \epsilon
\end{array}
$$

Parsing table M:

	$($	a	$+$	$*$	$)$	$\$$
E	$E \rightarrow TE'$	$E \rightarrow TE'$				
T	$T \rightarrow FT'$	$T \rightarrow FT'$				
F	$F \rightarrow (E)$	$F \rightarrow a$				
E'			$E' \rightarrow +TE'$		$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T'			$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$

Analyse the string $a + a * a$

Stack	Rest of the input	Chosen rule
E	$a + a * a$	$M[E, a] = E \rightarrow TE'$
TE'	$a + a * a$	$M[T, a] = T \rightarrow FT'$
$FT'E'$	$a + a * a$	$M[F, a] = F \rightarrow a$
$aT'E'$	$a + a * a$	$-$
$T'E'$	$+a * a$	$M[T', +] = T' \rightarrow \epsilon$
E'	$+a * a$	$M[E', +] = E' \rightarrow +TE'$
$+TE'$	$+a * a$	$-$
TE'	$a * a$	$M[T, a] = T \rightarrow FT'$
FTE'	$a * a$	$M[F, a] = F \rightarrow a$
$aT'E'$	$a * a$	$-$

Analyse the string $a + a * a$ (cont.)

Stack	Rest of the input	Chosen rule
$T'\,E'$	$*a$	$M[T', *] = T' \rightarrow *FT'$
$*FT'\,E'$	$*a$	–
$FT'\,E'$	a	$M[F, a] = F \rightarrow a$
$aT'\,E'$	a	–
$T'\,E'$	ϵ	$M[T', \$] = T' \rightarrow \epsilon$
E'	ϵ	$M[E', \$] = E' \rightarrow \epsilon$
ϵ	ϵ	–

LL(1) Grammars

- The algorithm for constructing a predictive parsing table can be applied to any grammar G to produce a parsing table M
- For some grammars however, M may have some entries that are multiply-defined
- If G is left recursive or ambiguous, then M will have at least one multiply-defined entry
- A grammar whose parsing table has no multiply-defined entries is said to be LL(1)
- LL(1) grammars have several distinctive properties
  - No ambiguous or left-recursive grammar can be LL(1)
  - A grammar G is LL(1) iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G the following conditions hold:
    1. For no terminal a do both α and β derive strings with a
    2. At most one of α and β can derive the empty string
    3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with the terminal in $FOLLOW(A)$

Examples

- Grammar G_1 is not LL(1)
- Grammar $G_2 = (\{E, E', T, T', F\}, \{a, *, +, (,)\}, P, E)$ where P is the set of productions:

$$
\begin{array}{lcllcl}
E & \rightarrow & TE' & T & \rightarrow & FT' \\
F & \rightarrow & (E) & E' & \rightarrow & +TE' \\
T' & \rightarrow & *FT' & F & \rightarrow & a \\
E' & \rightarrow & \epsilon & T' & \rightarrow & \epsilon
\end{array}
$$

is LL(1)

Remarks

- What should be done when a parsing table has multiply-defined entries?
- One recourse is to transform the grammar by eliminating left recursion and then left factoring whenever possible, hoping to produce a grammar for which the parsing table has no multiply-defined entries
- There are grammars for which no amount of alteration will yield an LL(1) grammar

Bottom-Up Parsing

- Constructs a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top)
- Reduction of a string w to the start symbol of a grammar
- At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production
- If the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse
- **Handles:** A handle of a string is a substring that matches the right side of a production and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation

Grammar:

$S \rightarrow \quad aABe \qquad A \rightarrow \quad Abc \mid b \qquad B \rightarrow \quad d$

The sentence $abbcde$ can be reduced as follows:

$abbcde$

$aAbcde$

$aAde$

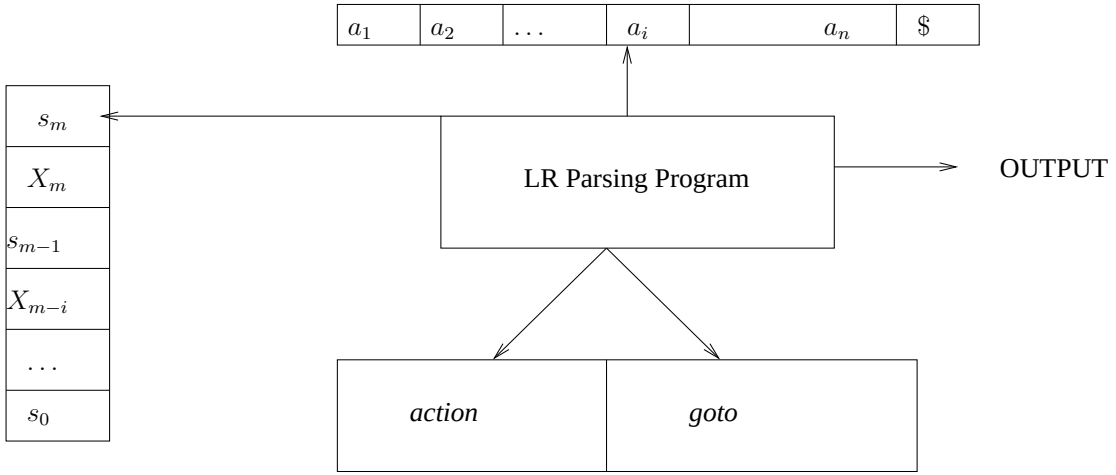
$aABe$

S

LR Parsers

- Technique that can be used to parse a large class of CFG
- Technique called LR(k) parsing
  - L: left-to-right scanning of the input
  - R: construction the rightmost derivation in reverse
  - k: number of input symbols of lookahead
- LR parsing is attractive for a variety of reasons:
  - LR parsers can be constructed to recognise virtually all programming language constructs for which CFG can be written
  - The class of grammars that can be parsed using LR methods is a **proper superset** of the class of grammars that can be parsed with predictive parsers
  - The LR parsing method can be implemented efficiently
- Drawback
  Too much complicate to construct an LR parser by hand for a typical programming language grammar
  One needs a specialised tools - an LR parser generator (as Yacc)

The LR Parsing Algorithm



- An LR parser consists of:
  1. An input
  2. An output
  3. A stack
  4. A driver program
  5. A parsing table that has two parts: *action* and *goto*

The LR Parsing Algorithm

- The driver program is the same for all LR parsers ; only the parsing table changes from one parser to another.
- The program uses a stack to store a string of the form

$$s_0\ X_1\ s_1\ X_2\ s_2 \ldots X_m\ s_m$$

where s_m is on the top

  - Each X_i is a grammar symbol
  - Each s_i is a state
    Each state summarises the information contained in the stack below it
    The combination of the state on the top of the stack and the current input symbol are used to index the parsing table and determine the shift-reduce parsing decision

The LR Parsing Algorithm

- The program driving the LR parser behaves as follows:
  1. It determines s_m (the state currently on the top of the stack) and a_i (the current input symbol)
  2. It consults *action*$[s_m, a_i]$ (the parsing action table entry for state s_m and input a_i) which can have one of the four values:
     - shift s, where s is a state
     - reduce by a grammar production $A \rightarrow \beta$
     - accept, and
     - error
- The function *goto* takes a state and a grammar symbol as arguments and produces a state.
- A *configuration* of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpected input $(s_0\ X_1\ s_1\ X_2 \ldots X_m\ s_m\ , a_i\ a_{i+1}\ a_n\ \$)$
- The next move of the parser is determined by reading a_i (the current input symbol) and s_m (the state on the top of the stack), and then consulting the parsing action table entry *action*$[s_m, a_i]$.

The LR Parsing Algorithm: configuration resulting after each of the four types or move

1. If *action*$[s_m, a_i] =$ shift s, then the parser executes a **shift move**, entering the configuration
   $(s_0\ X_1\ s_1\ X_2 \ldots X_m\ s_m\ a_i\ s\ , a_{i+1}\ a_n\ \$)$
   - The parser shifts both the current input symbol a_i and the next state s, which is given by *action*$[s_m, a_i]$, onto the stack.
   - a_{i+1} becomes the current input symbol
2. If *action*$[s_m, a_i] =$ reduce $A \rightarrow \beta$ then the parser executes a reduce move, entering the configuration $(s_0\ X_1\ s_1\ X_2 \ldots X_{m-r}\ s_{m-r}\ A\ s\ , a_i\ a_{i+1}\ a_n\ \$)$ where $s =$ *goto*$[s_{m-r}, A]$ and r **is the length of** β
   - The parser first popped $2r$ symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r}
   - The parser pushed both A (the left side of the production) and s (the entry *goto*$[s_{m-r}, A]$) onto the stack
   - The current input symbol is not changed in a reduce move
3. If *action*$[s_m, a_i] =$ accept, parsing is completed
4. If *action*$[s_m, a_i] =$ error, the parser has discovered an error

Algorithm: LR parsing

Input: An input string w and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If w is in $L(G)$, a bottom-up parse for w; otherwise, an error indication

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer.
The parser executes the following program until an accept or error action is encountered.

Algorithm: LR parsing

set ip to point to the first symbol of w;
repeat forever begin
    let s be the state on the top of the stack and
        a the symbol pointed by ip;
    if *action*$[s, a] =$ shift s' **then begin**
        push a then s' on top of the stack;
        advance ip to the next input symbol;
    end
    else if *action*$[s, a] =$ reduce $A \rightarrow \beta$ **then begin**
        pop $2* \mid \beta \mid$ symbols off the stack;
        let s' be the state now on top of the stack;
        push then *goto*$[s', A]$ on top of the stack;
        output the production $A \rightarrow \beta$;
    end
    else if *action*$[s, a] =$ accept **then return**
    else error()
end

Example

We consider the grammar $G_1 = (\{E, F, T\}, \{\textbf{id}, (,), *, +\}, P, E)$ where the productions are:

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow \textbf{id}$

and the input **id** $*$ **id** $+$ **id**

In the table:

s_i: shift and stack state i

r_j: reduce by production numbered j

acc: accept

blank means error

Example: grammar G_1 and input id $*$ id $+$ id

Parsing table for expression grammar

State	*action*						*goto*		
	id	$+$	$*$	$($	$)$	\$	E	T	F
0	s_5			s_4			1	2	3
1		s_6				acc			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4	r_4	r_4	8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5			s_4	r_4	r_4		9	3
7	s_5			s_4	r_4	r_4			10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Remarks

- The value of *goto*$[s, a]$ for terminal a is found in the action field connected with the shift action on input a for state s
- The goto fields gives *goto*$[s, A]$ for non terminals A
- We have not yet seen how the entries for the parsing table were selected

LR Parsing Methods

Three methods varying in their power of implementation

1. **SLR** : simple LR
   The weakest of the three in terms of number of grammars for which it succeeds, but is the easiest to implement
   - SLR table: parsing table constructed by this method
   - SLR parser
   - SLR grammar
2. **Canonical LR** : the most powerful and the most expensive
3. **LALR** (Lookahead LR): Intermediate in power and cost
   LALR method works on most programming-language grammars and, with some effort can be implemented efficiently

Building SLR Parsing Table

Definitions

LR(0) item or **item** (for short) of a grammar G:
A production of G with a dot at some position at the right side.

Example: Production $A \rightarrow XYZ$ yields the four items:
$A \rightarrow .XYZ$
$A \rightarrow X.YZ$
$A \rightarrow XY.Z$
$A \rightarrow XYZ.$
Production $A \rightarrow \epsilon$ generates one item $A \rightarrow .$

Intuitively: an item indicates how much of a production we have seen at a given point in the parsing process

Example:
$A \rightarrow X.YZ$
indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ

Augmented grammar

If G is a grammar with a start symbol S, then G', the augmented grammar for G is G with a new start symbol S' and production $S' \rightarrow S$.

Indicates to the parser when it should stop parsing and announce the acceptance of the input.

The Closure Operation

If I is a set of items for a grammar G, then $closure(I)$ is the set of items constructed from I by two rules:

1. Initially, every item in I is added to $closure(I)$
2. If $A \rightarrow \alpha.B\beta$ is in $closure(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow .\gamma$ to I (if it is not already there)
   We apply this rule until no more new items can be added to $closure(I)$

Why do we include $B \rightarrow .\gamma$ in $closure(I)$?

Intuitively:
(i) $A \rightarrow \alpha.B\beta$ in $closure(I)$ indicates that, at this point of the parsing, we think we might next see a substring derivable from $B\beta$ as input.
(ii) If $B \rightarrow \gamma$ is a production we expect we might see a substring derivable from γ at this point.

Example

We consider the augmented G_1 :

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \\
E & \rightarrow & T \\
T & \rightarrow & T * F \\
T & \rightarrow & F \\
F & \rightarrow & (E) \\
F & \rightarrow & \textbf{id}
\end{array}
$$

If I is the set of one item $\{[E' \rightarrow .E]\}$ then $closure(I)$ contains the items

$$
\begin{array}{lcl}
E' & \rightarrow & .E \\
E & \rightarrow & .E + T \\
E & \rightarrow & .T \\
T & \rightarrow & .T * F \\
T & \rightarrow & .F \\
F & \rightarrow & .(E) \\
F & \rightarrow & .\textbf{id}
\end{array}
$$

The *Goto* Operation

$goto(I, X)$, where I is a set of items and X is a grammar symbol
$goto(I, X)$ is the closure of the set of all items $[A \rightarrow \alpha X.\beta]$ such that $[A \rightarrow \alpha.X\beta]$ is in I

Example: If I is the set of two items $\{[E' \rightarrow E.], [E \rightarrow E. + T]\}$ then $goto(I, +)$ consists of

$$
\begin{array}{lcl}
E & \rightarrow & E + .T \\
T & \rightarrow & .T * F \\
T & \rightarrow & .F \\
F & \rightarrow & .(E) \\
F & \rightarrow & .\textbf{id}
\end{array}
$$

We compute $goto(I, +)$ by examining I for items with $+$ immediately to the right of the dot

The Sets-of-Items Construction

Construction of C, the canonical collection of sets of LR(0) items for an augmented grammar G'

procedure items(G');
begin
    $C := \{closure(\{[S' \rightarrow .S]\})\}$
    repeat
        for each set of items I in C and each grammar symbol X
            such that $goto(I, X)$ is not empty and not in C **do**
                add $goto(I, X)$ to C
    until no more sets of items can be added to C
end

Remarks

- For every grammar G, the *goto* function of the canonical collection of sets of items defines a deterministic finite state automaton D
  - The DFA D is obtained from a NFA N by the subset construction
  - States of N are the items
  - There is a transition from $A \rightarrow \alpha.X\beta$ to $A \rightarrow \alpha X.\beta$ labelled X, and there is a transition from $A \rightarrow \alpha.B\beta$ to $B \rightarrow .\gamma$ labelled ϵ
  - The $closure(I)$ for a set of items (states of N) I is the ϵ-closure of a set of states of NFA states
- *goto*(I, X) gives the transition from I on symbol X in the DFA constructed from N by the subset construction

SLR Parsing Tables

- We construct the SLR parsing action and goto functions
- The algorithm will not produce uniquely defined parsing action tables for all grammars, but it succeed on many grammars for programming languages
- Given a grammar G we augment G to produce G', and for G' we construct C, the canonical collection of sets of items for G'
- We construct *action* and *goto* from C using the following algorithm
- The algorithm requires us to know $FOLLOW(A)$ for each non terminal A of a grammar

Algorithm SLR: Constructing SLR parsing table

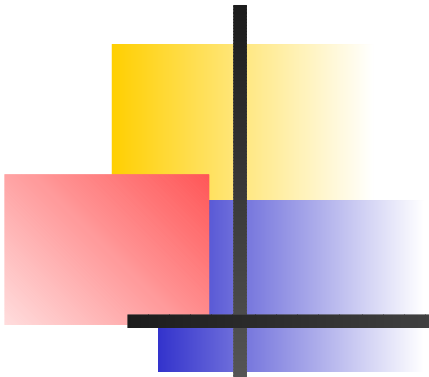
Input: An augmented grammar G'.

Output: The SLR parsing table functions *action* and *goto* for G'

Algorithm SLR: method

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'
2. State i is constructed from I_i. The parsing actions for state i are determined as follows:
   (a) If $[A \rightarrow \alpha.a\beta]$ is in I_i and *goto*$(I_i, a) = I_j$, then set *action*$[i, a]$ to "shift j". Here a must be a terminal.
   (b) If $[A \rightarrow \alpha.]$ is in I_i, then set *action*$[i, a]$ to reduce $A \rightarrow \alpha$ for all $a \in FOLLOW(A)$. Here A may not be S'
   (c) If $[S' \rightarrow S.]$ is in I_i, then set *action*$[i, \$]$ to accept

   If any conflicting actions are generated by the above rules, we say that the grammar is not SLR(1). *The algorithm fails to produce a parser in this case.*
3. The goto transitions for state i are constructed for all non terminals A using the rule:
   if *goto*$(I_i, A) = I_j$ then *goto*$[i, A] = j$
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing $[S \rightarrow .S']$



- SLR(1) Table for G: parsing table determined by the above algorithm
- SLR(1) Parser: an LR parser using the SLR(1) table for G
- SLR(1) Grammar: a grammar having an SLR(1) parsing table
- We usually omit the ”(1)” after SLR, since we shall not deal with parsers having more than one symbol of lookahead
- Every SLR grammar is unambiguous, but there are many unambiguous grammars that are not SLR

Constructing Canonical LR Parsing Tables

- In SLR method:
  State i calls for reduction by $A \rightarrow \alpha$ if:
  - the set of items I_i contains item $[A \rightarrow \alpha.]$ and
  - $a \in FOLLOW(A)$
- In some situations, when state i appears on the top of the stack, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by a in a right-sentential form
  Thus, the reduction $A \rightarrow \alpha$ would be invalid on input a

Example

We consider the grammar G_4 with productions:

$$\begin{array}{lcl} S & \rightarrow & L = R \\ S & \rightarrow & R \\ L & \rightarrow & *R \\ L & \rightarrow & \textbf{id} \\ R & \rightarrow & L \end{array}$$

In state 2: we have item $R \rightarrow L$.

- Corresponds to $A \rightarrow \alpha$: a is $=$ in $FOLLOW(R)$.
- SLR parser calls for reduction by $R \rightarrow L$ in state 2 with $=$ as the next input
- There is **no** right-sentential form of G_4 that begins with $R = \ \ldots$

State 2 (which is the state corresponding to viable prefix L only) should not call for reduction of that L to R

Remarks

Remarks...

- It is possible to carry more information in the state to rule out some invalid reductions
- By splitting states when necessary we can arrange to have each state of an LR parse indicate exactly which input symbols can follow α such that there is a possible reduction to A
- The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component
- **LR(1) item:** $[A \rightarrow \alpha.\beta, a]$ where $A \rightarrow \alpha.\beta$ is a production and a is a terminal or the right endmarker $\$$
- In LR(1), the 1 refers to the length of the second component (lookahead of the item)
- The lookahead has no effect in an item of the form $[A \rightarrow \alpha.\beta, a]$, where β is not ϵ
- An item of the form $[A \rightarrow \alpha., a]$ calls for reduction by $A \rightarrow \alpha$ only if the next input symbol is a
- Thus, we reduce by $A \rightarrow \alpha$ only on those input symbols a for which $[A \rightarrow \alpha., a]$ is in an LR(1) item in the state on the top of the stack
- The set of a's will always be a subset of $FOLLOW(A)$, but it could be a proper

- Formally:
  LR(1) item $[A \rightarrow \alpha.\beta, a]$ is valid for a viable prefix γ if there is a rightmost derivation:

  $$S \stackrel{*}{\Rightarrow} \delta A w \stackrel{*}{\Rightarrow} \delta\alpha\beta w$$

  where:

  1. $\gamma = \delta\alpha$ and
  2. either a is the first symbol of w, or
     w is ϵ and a is \$

- Method for constructing the collection of sets of valid LR(1) items is essentially the same as the way we built the canonical collection of sets of LR(0) items. We only need to modify the two procedures *closure* and *goto*

- The basis of the new *closure* operation:
  - Consider item $[A \rightarrow \alpha.B\beta, a]$ in the set of items valid for some viable prefix γ
  - Then there is a rightmost derivation $S \stackrel{*}{\Rightarrow} \delta Aax \stackrel{*}{\Rightarrow} \delta\alpha B\beta ax$ where $\gamma = \delta\alpha$
  - Suppose βax derives terminal string by
  - For each production of the for $B \rightarrow \mu$ (form some μ) we have

$$S \stackrel{*}{\Rightarrow} \gamma Bby \stackrel{*}{\Rightarrow} \gamma\mu by$$

  - Thus $[B \rightarrow .\mu, b]$ is valid for γ
  - Note:
    1. b can be the first terminal derived from β, or
    2. β can derive ϵ in the derivation $\beta ax \stackrel{*}{\Rightarrow} by$ (and b can therefore be a)
  - Thus, to summarise:
    b can be any terminal in $FIRST(\beta ax)$
    Remark: As x cannot contain the first terminal of by

$$FIRST(\beta ax) = FIRST(\beta a)$$

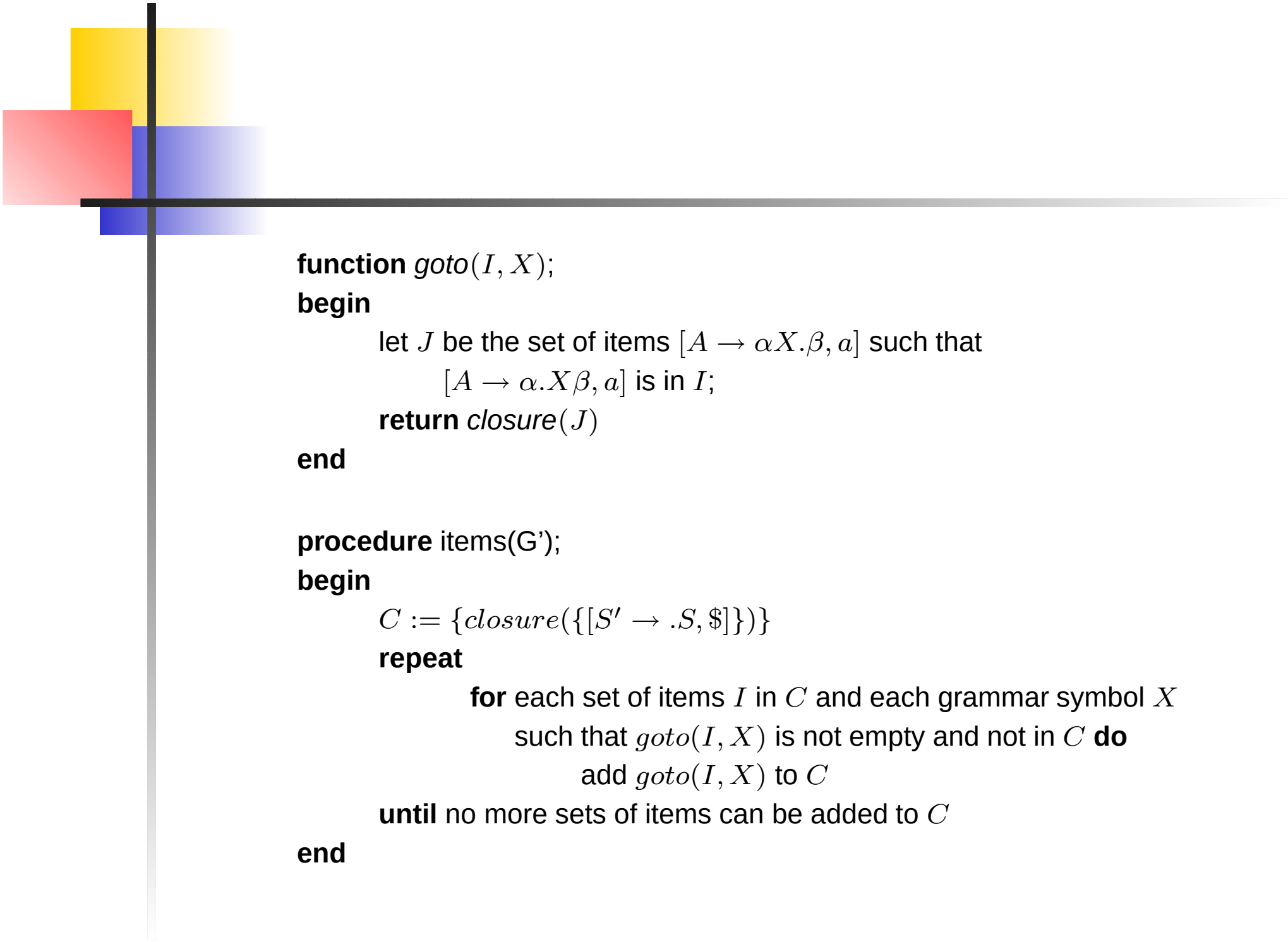
Algorithm: Construction of the sets of LR(1) items

Input: An augmented grammar G'.

Output: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G'

Method: The procedure *closure* and *goto* and the main routine **items** for constructing the sets of items are shown in the following

function *closure*(I);
begin
  repeat
    for each item $[A \rightarrow \alpha.B\beta, a]$ in I,
      each production $B \rightarrow \gamma$ in G', and
      each terminal $b \in FIRST(\beta a)$ such that $[B \rightarrow .\gamma, b] \notin I$ **do**
        add $[B \rightarrow .\gamma, b]$ to I;
  until no more items can be added to I;
return (I)
end



function *goto*(I, X);
begin
    let J be the set of items $[A \rightarrow \alpha X.\beta, a]$ such that
        $[A \rightarrow \alpha.X\beta, a]$ is in I;
    return *closure*(J)
end

procedure items(G');
begin
    $C := \{closure(\{[S' \rightarrow .S, \$]\})\}$
    repeat
        for each set of items I in C and each grammar symbol X
            such that $goto(I, X)$ is not empty and not in C **do**
                add $goto(I, X)$ to C
    until no more sets of items can be added to C
end

Algorithm: Canonical LR parsing table

Input: An augmented grammar G'.

Output: The canonical LR parsing table functions *action* and *goto* for G'

Algorithm: Method

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items for G'
2. State i is constructed from I_i. The parsing actions for state i are determined as follows:
   (a) If $[A \rightarrow \alpha.a\beta, b]$ is in I_i and *goto*$(I_i, a) = I_j$, then set *action*$[i, a]$ to "shift j". Here a must be a terminal.
   (b) If $[A \rightarrow \alpha., a]$ is in I_i, then set *action*$[i, a]$ to reduce $A \rightarrow \alpha$. Here A may not be S'
   (c) If $[S' \rightarrow S., \$]$ is in I_i, then set *action*$[i, \$]$ to accept

   If any conflicting actions are generated by the above rules, we say that the grammar is not LR(1). *The algorithm fails to produce a parser in this case.*
3. The goto transitions for state i are constructed for all non terminals A using the rule:
   if *goto*$(I_i, A) = I_j$ then *goto*$[i, A] = j$
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing $[S \rightarrow S', \$]$

Examples

We consider the grammar $G = (\{S, C\}, \{c, d\}, P, S)$ where the productions are:

(1) $S \rightarrow CC$ (2) $C \rightarrow cC$ (3) $C \rightarrow d$

The augmented grammar G has also rule $S' \rightarrow S$

Canonical parsing table for grammar

State	*action*			*goto*	
	c	d	\$	S	C
0	s_3	s_4		1	2
1			accept		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Construction LALR Parsing Tables

- *Lookahead*-LR technique
- Often used in practise because the tables obtained by it are considerably smaller that the canonical LR tables
- Common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar
- The same is **almost** true for SLR grammars - but there are a few constructs that cannot be conveniently handled by SLR techniques
- Parser size
  - SLR and LALR tables for a grammar: always have the same number of states
  - For a language like Pascal
    - SLR and LALR: several hundred states
    - LR: several thousand states

Consider $G = (\{S, C\}, \{\ c, d\}, P, S)$ with productions:

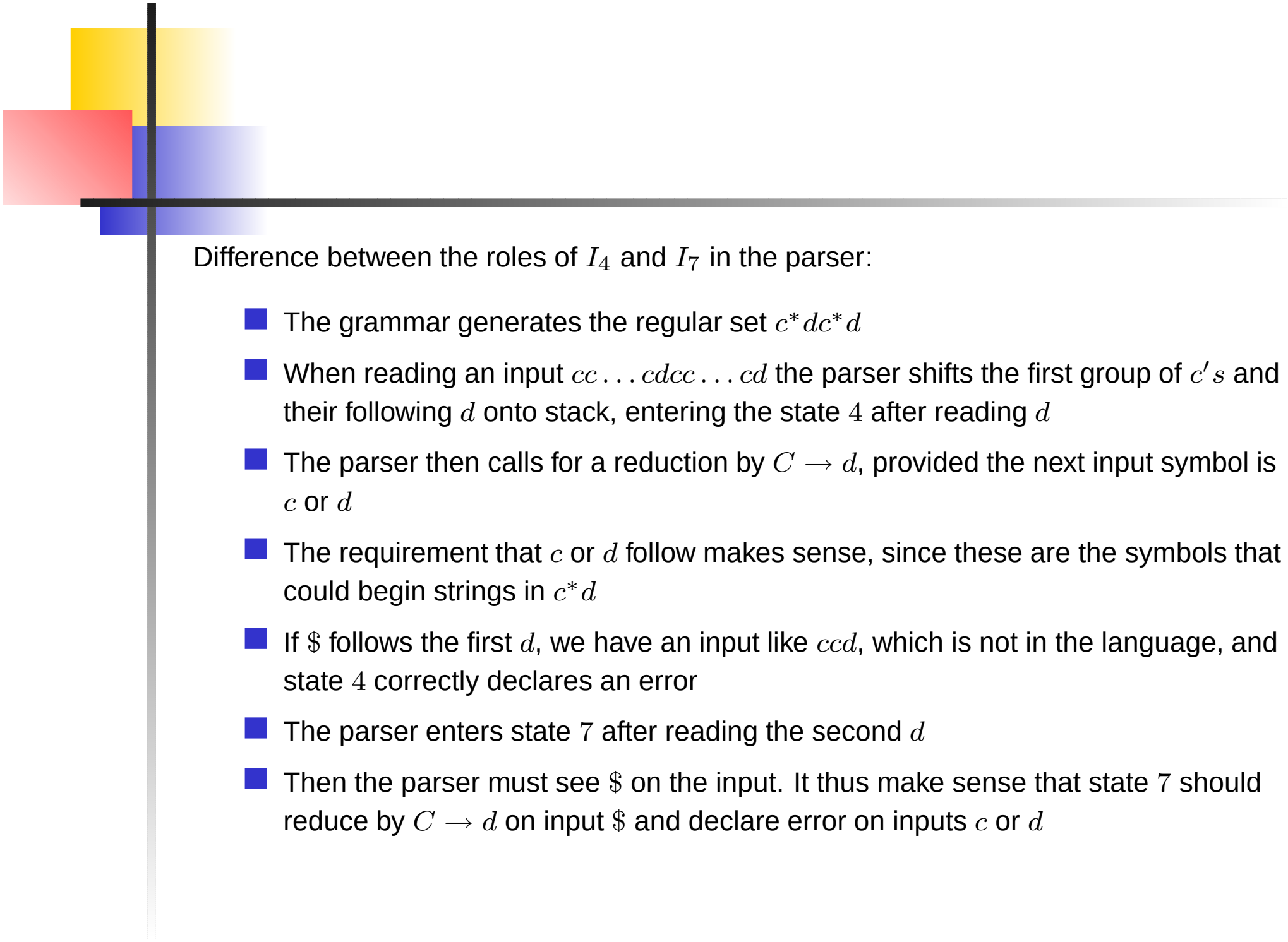
(1) $S \rightarrow CC$ (2) $C \rightarrow cC$ (3) $C \rightarrow d$

and its sets of LR(1) items. Take a pair of similar looking states such as I_4 and I_7:

$I_4 :\ \ C \rightarrow d.,\ c/d$ $I_7 :\ \ C \rightarrow d.,\ \$$

- Each of these states has only items with first component

$$C \rightarrow d.$$

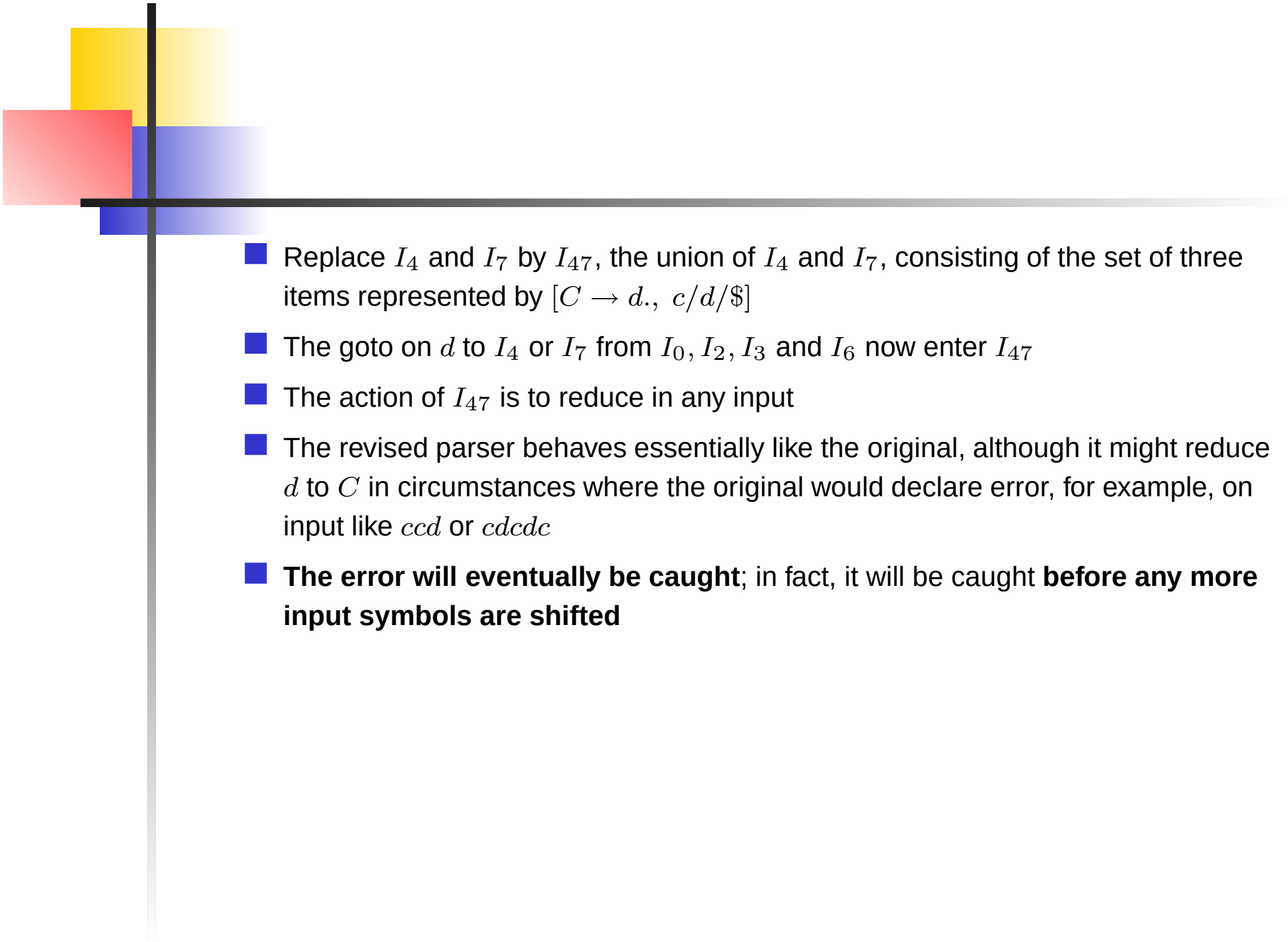


Difference between the roles of I_4 and I_7 in the parser:

- The grammar generates the regular set c^*dc^*d
- When reading an input $cc\ldots cdcc\ldots cd$ the parser shifts the first group of $c's$ and their following d onto stack, entering the state 4 after reading d
- The parser then calls for a reduction by $C \rightarrow d$, provided the next input symbol is c or d
- The requirement that c or d follow makes sense, since these are the symbols that could begin strings in c^*d
- If $\$$ follows the first d, we have an input like ccd, which is not in the language, and state 4 correctly declares an error
- The parser enters state 7 after reading the second d
- Then the parser must see $\$$ on the input. It thus make sense that state 7 should reduce by $C \rightarrow d$ on input $\$$ and declare error on inputs c or d

Stack	Input	Action
0	$cccdcccd\$$	$\mathit{action}[0, c] = s3$
$0c3$	$ccdccd\$$	$\mathit{action}[3, c] = s3$
$0c3c3$	$cdcc \ldots cd\$$	$\mathit{action}[3, c] = s3$
$0c3c3c3$	$dccd\$$	$\mathit{action}[3, d] = s4$
$0c3c3c3d4$	$ccd\$$	$\mathit{action}[4, c] = r3 \ \ (C \rightarrow d)$ $\mathit{goto}[3, C] = 8$
$0c3c3c3C8$	$ccd\$$	$\mathit{action}[8, c] = r2 \ \ (C \rightarrow cC)$ $\mathit{goto}[3, C] = 8$
$0c3c3C8$	$ccd\$$	$\mathit{action}[8, c] = r2 \ \ (C \rightarrow cC)$ $\mathit{goto}[3, C] = 8$
$0c3C8$	$ccd\$$	$\mathit{action}[8, c] = r2 \ \ (C \rightarrow cC)$ $\mathit{goto}[0, C] = 2$
$0C2$	$ccd\$$	$\mathit{action}[2, c] = s6$

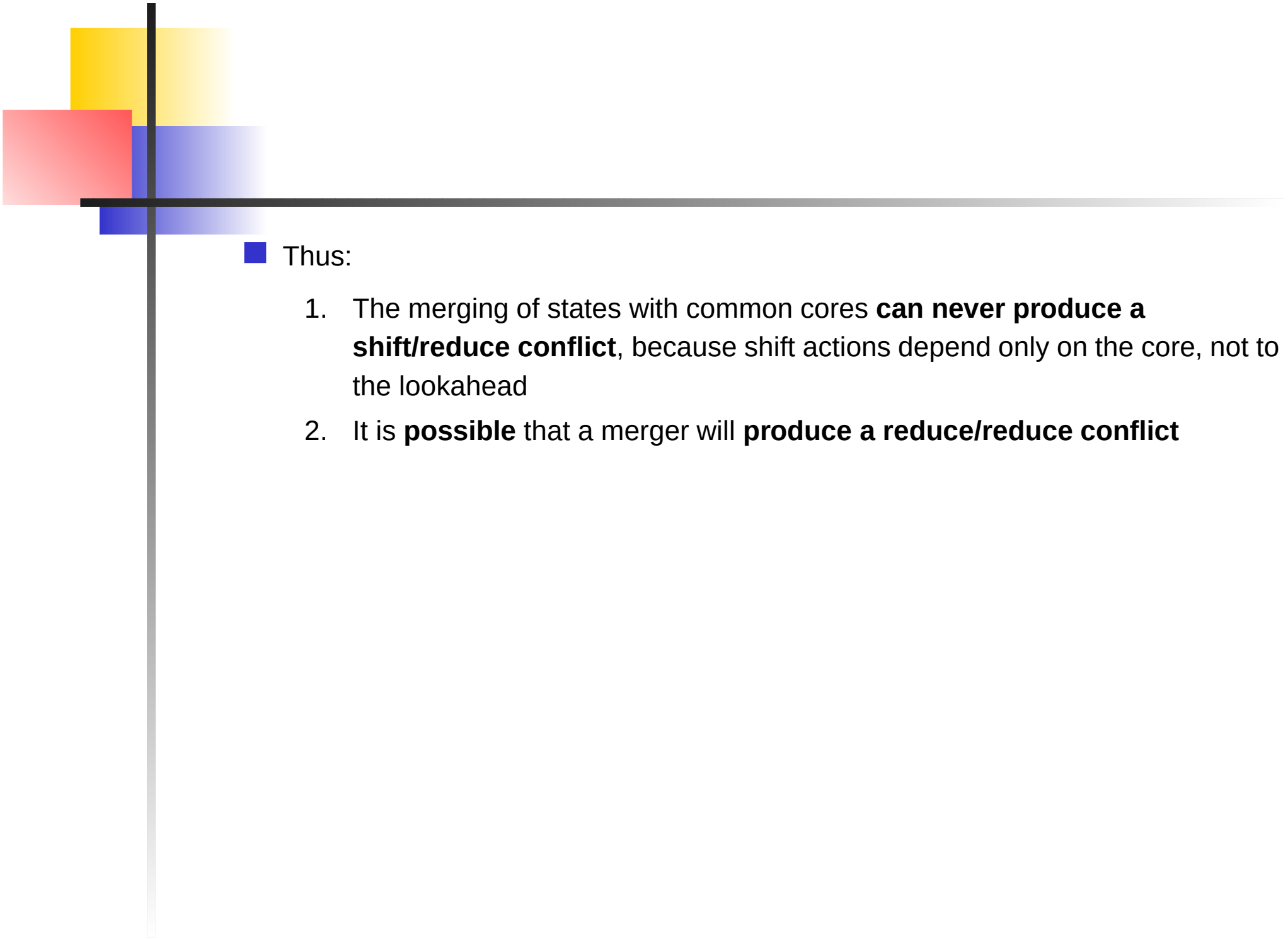
Stack	Input	Action
$0C2c6$	$cd\$$	$\mathit{action}[6, c] = s6$
$0C2c6c6$	$d\$$	$\mathit{action}[6, d] = s7$
$0C2c6c6d7$	$\$$	$\mathit{action}[7, \$] = r_3 \quad (C \rightarrow d)$ $\mathit{goto}[6, C] = 9$
$0C2c6c6C9$	$\$$	$\mathit{action}[9, \$] = r_2 \quad (C \rightarrow cC)$ $\mathit{goto}[6, C] = 9$
$0C2c6C9$	$\$$	$\mathit{action}[9, \$] = r_2 \quad (C \rightarrow cC)$ $\mathit{goto}[2, C] = 5$
$0C2C5$	$\$$	$\mathit{action}[5, \$] = r_1 \quad (S \rightarrow CC)$ $\mathit{goto}[0, S] = 1$
$0S1$	$\$$	$\mathit{action}[1, \$] = \mathit{accept}$



- Replace I_4 and I_7 by I_{47}, the union of I_4 and I_7, consisting of the set of three items represented by $[C \rightarrow d.,\ c/d/\$]$
- The goto on d to I_4 or I_7 from I_0, I_2, I_3 and I_6 now enter I_{47}
- The action of I_{47} is to reduce in any input
- The revised parser behaves essentially like the original, although it might reduce d to C in circumstances where the original would declare error, for example, on input like ccd or $cdcdc$
- **The error will eventually be caught**; in fact, it will be caught **before any more input symbols are shifted**

More generally ...

- We can look for sets of $LR(1)$ items having the same *core* (set of first components) and we may merge these sets
- A core is a set of LR(0) items for the grammar at hand, and an LR(1) grammar may produce more than two sets of items with the same core
- Suppose we have an LR(1) grammar (*i.e.*, one whose sets of LR(1) items produce no parsing action conflicts)
- If we replace all states having the same core with their union, it is possible that the resulting union will have a conflict
  - But a reduce/shift conflict is unlikely
    Suppose in the union a conflict on lookahead a:
    $[A \rightarrow \alpha., a]$ calling for a reduction by $A \rightarrow a$
    $[B \rightarrow \beta.a\gamma, b]$ calling for a shift
  - Then for some set of items from which the union was formed has item $[A \rightarrow \alpha., a]$, and since the cores of all these states are the same, it must have an item $[B \rightarrow \beta.a\gamma, c]$ for same c.
  - But then this state has the same shift/reduce conflict on a and the grammar was no LR(1) as we assumed



- Thus:
  1. The merging of states with common cores **can never produce a shift/reduce conflict**, because shift actions depend only on the core, not to the lookahead
  2. It is **possible** that a merger will **produce a reduce/reduce conflict**