Languages and Compilers
(SProg og Oversættere)

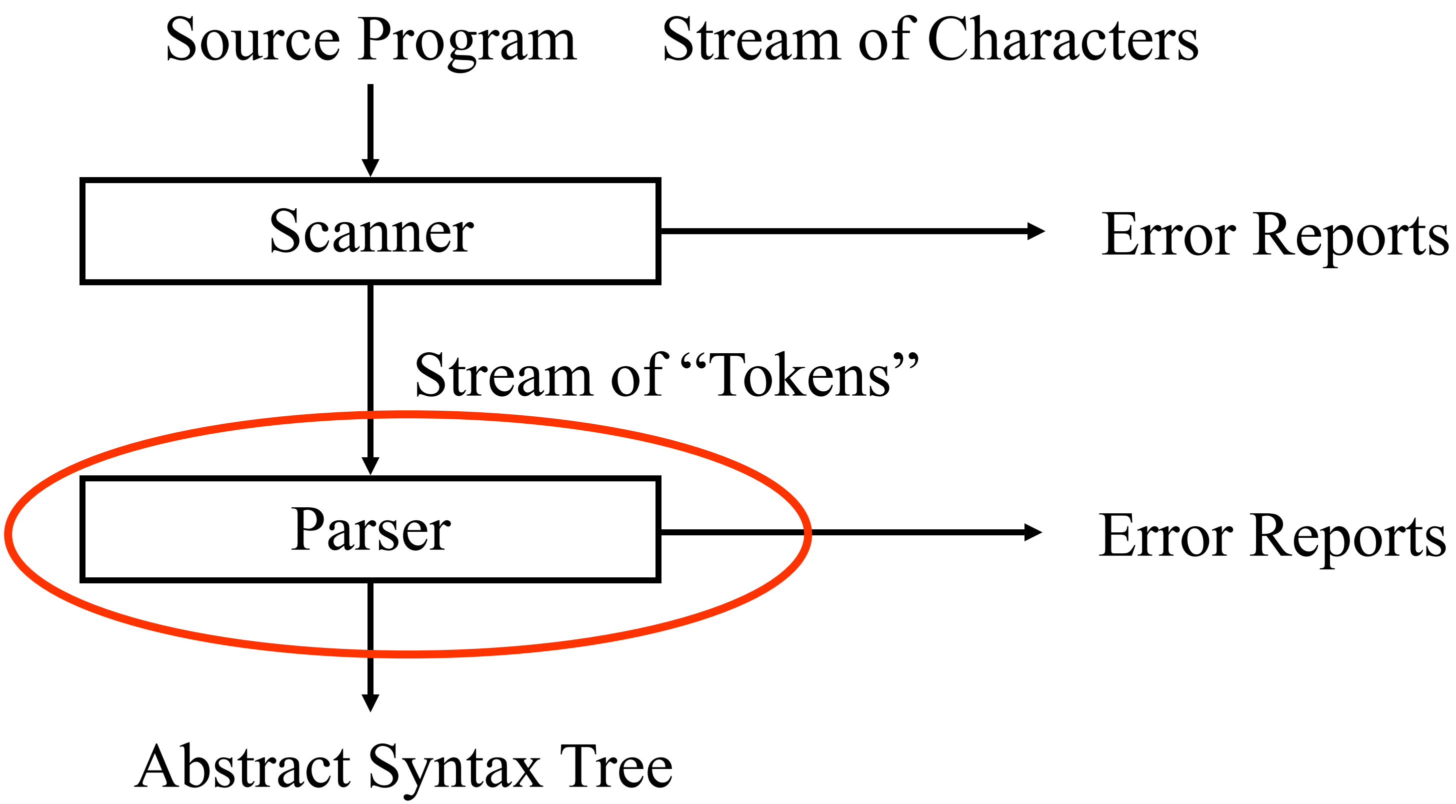
Parsing

Parsing

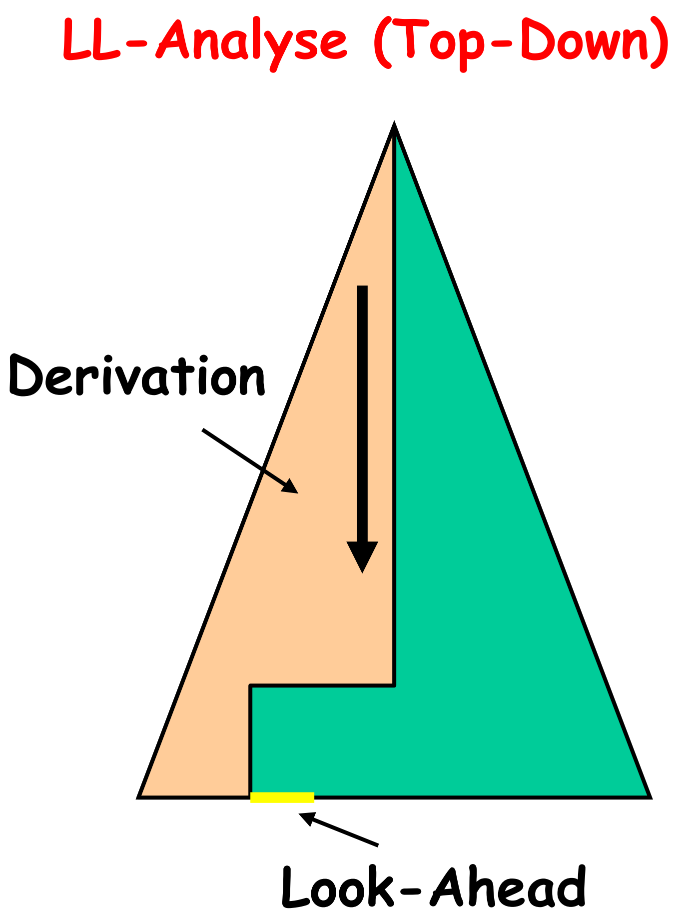
a. Describe the purpose of the parser
b. Discuss top down vs. bottom up parsing
c. Explain necessary conditions for construction of recursive decent parsers
d. Discuss the construction of an RD parser from a grammar
e. Discuss bottom Up/LR parsing
f. Discuss the dangling else problem

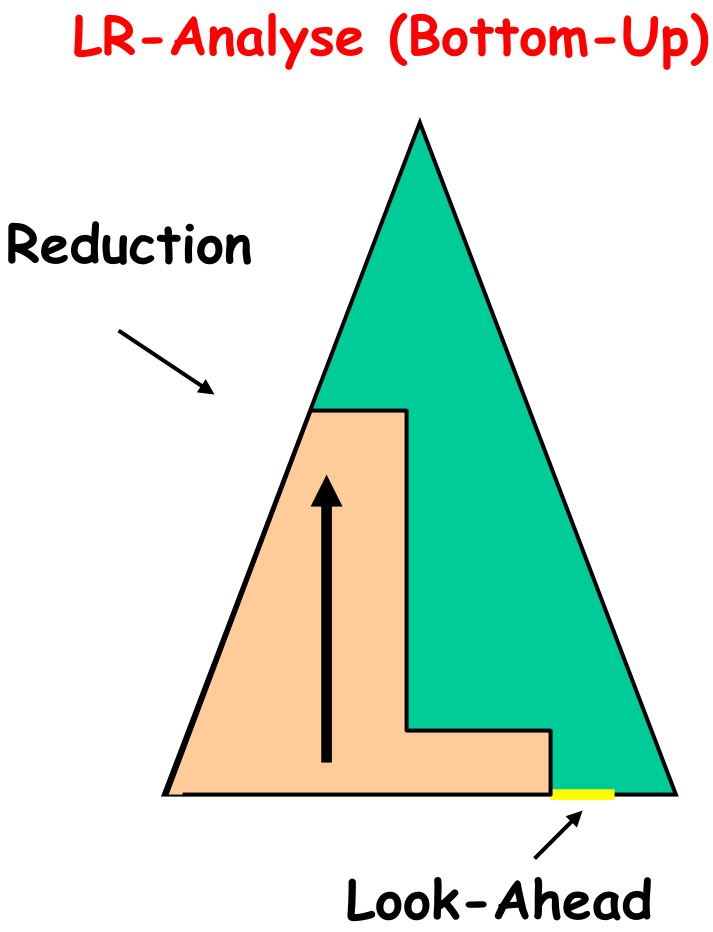
Syntax Analysis

Dataflow chart



Top-Down vs Bottom-Up parsing





Recursive Descent Parsing

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

Define a procedure parseN for each non-terminal N

```
private void parseSentence() ;
private void parseSubject();
private void parseObject();
private void parseNoun();
private void parseVerb();
```

Recursive Descent Parsing: Parsing Methods

```
Sentence  ::= Subject Verb Object .
```

```
private void parseSentence() {
  parseSubject();
  parseVerb();
  parseObject();
  accept('.');
}
```

Recursive Descent Parsing: Parsing Methods

```
Subject    ::= I  |  a Noun  |  the Noun
```

```
private void parseSubject() {
  if (currentTerminal matches 'I')
    accept('I');
  else if (currentTerminal matches 'a') {
    accept('a');
    parseNoun();
  }
  else if (currentTerminal matches 'the') {
    accept('the');
    parseNoun();
  }
  else
   report a syntax error
}
```

Formal definition of LL(1)

A grammar G is LL(1) iff
for each set of productions $X ::= X_1 \mid X_2 \mid \ldots \mid X_n$:

1. *starters*[X_1], *starters*[X_2], …, *starters*[X_n] are all pairwise disjoint
2. *If* $X_i \Rightarrow^* \varepsilon$ then *starters*[X_j]$\cap$ *follow*[X]=Ø, for $1 \leq j \leq n.i \neq j$

If G is ε-free then 1 is sufficient

NOTE: starters[X_1] is sometimes called *first*[X_1]

starters[X] = {t in Terminals | X =>* t β }
Follow[X] = {t in Terminals | S =>+ α X t β }

LL 1 Grammars

*parse X**

```
while (currentToken.kind is in starters[X]) {
  parse X
}
```

Condition: *starters*[*X*] must be disjoint from the set of tokens that can immediately follow *X* *

parse X|Y

```
switch (currentToken.kind) {
  cases in starters[X]:
    parse X
    break;
  cases in starters[Y]:
    parse Y
    break;
  default: report syntax error
}
```

Condition: *starters*[*X*] and *starters*[*Y*] must be disjoint sets.

```
function IsLL1(G) returns Boolean
    foreach A ∈ N do
        PredictSet ← ∅
        foreach p ∈ ProductionsFor(A) do
            if Predict(p) ∩ PredictSet ≠ ∅                                   (4)
            then return (false)
            PredictSet ← PredictSet ∪ Predict(p)
    return (true)
end
```

Figure 5.4: Algorithm to determine if a grammar G is LL(1).

```
function Predict(p : A → X_1 ... X_m) : Set
    ans ← First(X_1 ... X_m)                                                 (1)
    if RuleDerivesEmpty(p)                                                   (2)
    then
        ans ← ans ∪ Follow(A)                                                (3)
    return (ans)
end
```

Figure 5.1: Computation of Predict sets.

```
procedure A(ts)
    switch (...)
        case ts.PEEK() ∈ Predict(p_1)
            /*    Code for p_1                                   */
        case ts.PEEK() ∈ Predict(p_i)
            /*    Code for p_2                                   */
        /*    .                                                  */
        /*    .                                                  */
        /*    .                                                  */
        case ts.PEEK() ∈ Predict(p_n)
            /*    Code for p_n                                   */
        case default
            /*    Syntax error                                   */
end
```

Figure 5.6: A typical recursive-descent procedure. Successful LL(1) analysis ensures that only one of the **case** predicates is **true**.

```
procedure S( )
    switch (...)
        case ts.PEEK( ) ∈ { a, b, q, c, $ }
            call A( )
            call C( )
            call MATCH( $ )
end
procedure C( )
    switch (...)
        case ts.PEEK( ) ∈ { c }
            call MATCH( c )
        case ts.PEEK( ) ∈ { d, $ }
            return ()
end
procedure A( )
    switch (...)
        case ts.PEEK( ) ∈ { a }
            call MATCH( a )
            call B( )
            call C( )
            call MATCH( d )
        case ts.PEEK( ) ∈ { b, q, c, $ }
            call B( )
            call Q( )
end
procedure B( )
    switch (...)
        case ts.PEEK( ) ∈ { b }
            call MATCH( b )
            call B( )
        case ts.PEEK( ) ∈ { q, c, d, $ }
            return ()
end
procedure Q( )
    switch (...)
        case ts.PEEK( ) ∈ { q }
            call MATCH( q )
        case ts.PEEK( ) ∈ { c, $ }
            return ()
end
```

Figure 5.7: Recursive-descent code for the grammar shown in Figure 5.2. The variable *ts* denotes the token stream produced by the scanner.

1	S	→	A C $
2	C	→	c
3		|	λ
4	A	→	a B C d
5		|	B Q
6	B	→	b B
7		|	λ
8	Q	→	q
9		|	λ

```
procedure MATCH( ts, token )
    if ts.PEEK( ) = token
    then  call ts.ADVANCE( )
    else  call ERROR( Expected token )
end
```

Figure 5.5: Utility for matching tokens in an input stream.

Bottom Up Parsing/ LR Parsing

- The main task of a bottom-up parser is to find the leftmost node that has not yet been constructed but all of whose children have been constructed.
- The sequence of children is called the **handle**.
- Creating a parent node *N* and connecting the children in the handle to *N* is called **reducing** to *N*.

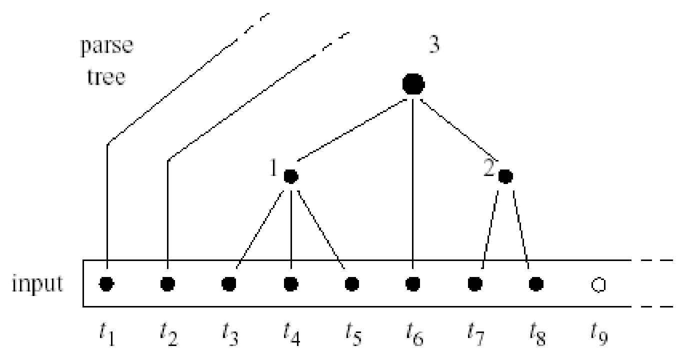


Figure 2.52 A bottom-up parser constructing its first, second, and third nodes.

(1,6,2) is a handle

Bottom Up Parsers/ shift-reduce

- All bottom up parsers have similar algorithm:
  - A loop with these parts:
    - try to find the leftmost node of the parse tree which has not yet been constructed, but all of whose children *have* been constructed.
      - This sequence of children is called a **handle**
      - **Shift** is the action of moving the next token to the top of the parse stack
    - construct a new parse tree node.
      - This is called **reducing**
- The difference between different algorithms is only in the way they find a handle.

1 Start → E $
2 E → plus E E
3 | num

Derivation

$$\text{Start} \Rightarrow_{rm} \text{E \$}$$
$$\Rightarrow_{rm} \text{plus E E \$}$$
$$\Rightarrow_{rm} \text{plus E num \$}$$
$$\Rightarrow_{rm} \text{plus num num \$}$$

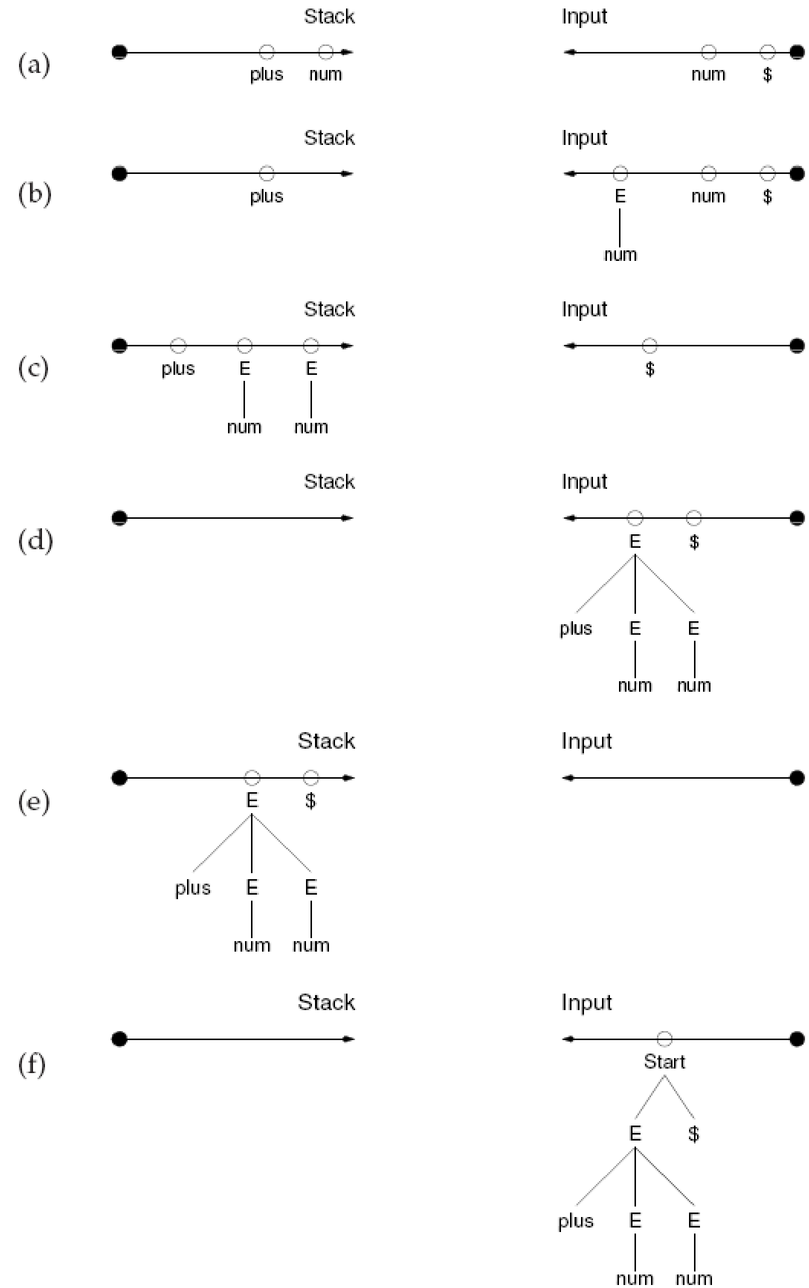


Figure 6.1: Bottom-up parsing resembles knitting.

Shifting and reducing

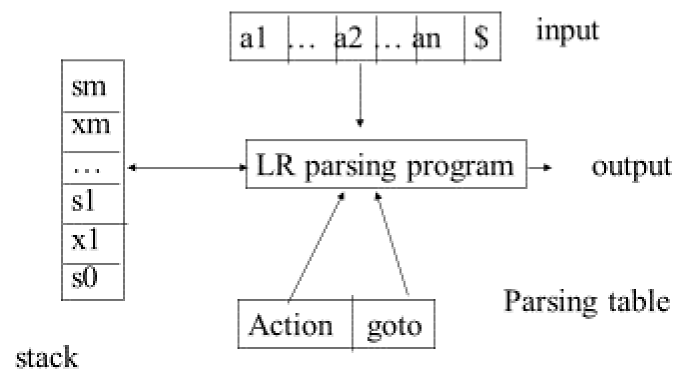
```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

```
Shift                            → ← the cat sees a rat .
Shift                        the → ← cat sees a rat .
Reduce                   the cat → ← sees a rat .
Shift                       the  → ← Noun sees a rat .
Reduce                  the Noun → ← sees a rat .
Reduce                           → ← Subject sees a rat .
Shift                    Subject → ← sees a rat .
Reduce              Subject sees → ← a rat .
Shift                   Subject  → ← Verb a rat .
Shift              Subject Verb  → ← a rat .
Shift             Subject Verb a → ← rat .
Reduce        Subject Verb a rat → ←.
Shift              Subject Verb  → ← Noun.
Reduce       Subject Verb a Noun → ←.
Shift               Subject Verb → ← Object.
Shift        Subject Verb Object → ←.
Shift      Subject Verb Object . → ←
Reduce                           → ← Sentence
Finish                  Sentence → ←
```

The LR-parse algorithm

- A finite automaton
  - With transitions and states
- A stack
  - with objects (symbol, state)
- A parse table

Model of an LR parser:



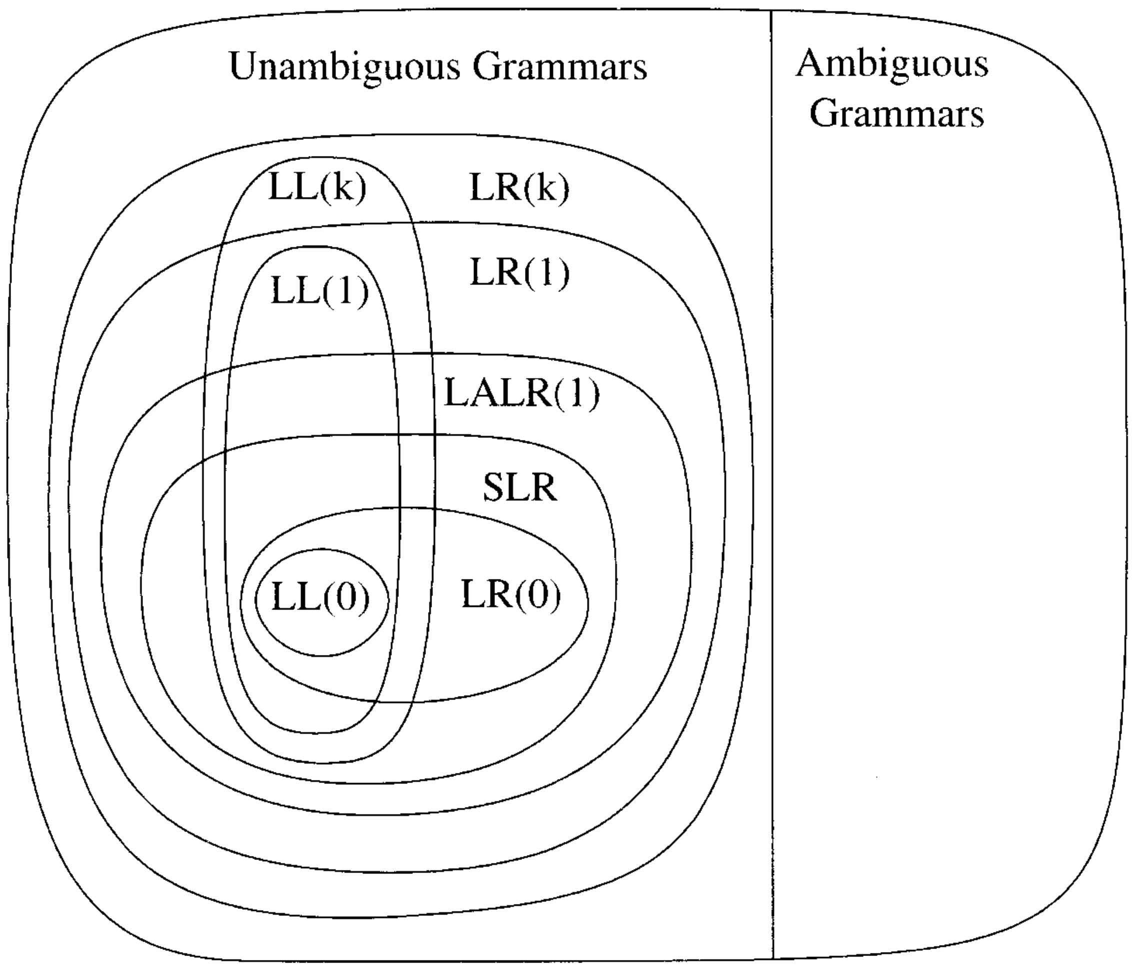
si is a state, xi is a grammar symbol

All LR parsers use the same algorithm, different grammars have different parsing tables.

```
call Stack.PUSH(StartState)
accepted ← false
while not accepted do
    action ← Table[Stack.TOS( )][InputStream.PEEK( )]        ①
    if action = shift s
    then
        call Stack.PUSH(s)                                   ②
        if s ∈ AcceptStates                                  ③
        then  accepted ← true
        else  call InputStream.ADVANCE( )
    else
        if action = reduce A → γ
        then
            call Stack.POP(|γ|)                              ④
            call InputStream.PREPEND(A)                      ⑤
        else
            call ERROR( )                                    ⑥
```

Figure 6.3: Driver for a bottom-up parser.

Hierarchy



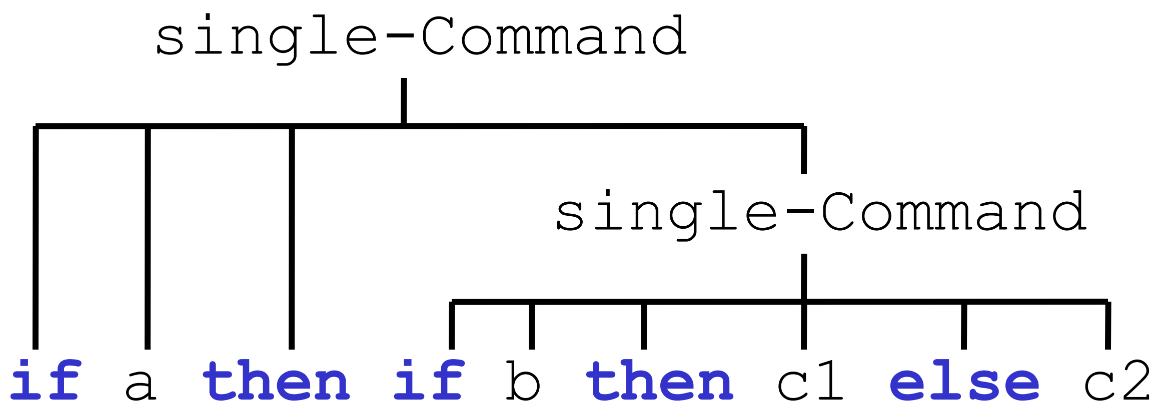
Unambiguous Grammars
Ambiguous Grammars
LL(k)
LR(k)
LL(1)
LR(1)
LALR(1)
SLR
LL(0)
LR(0)

Dangling Else Problem

Example: (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                        else single-Command
```

This parse tree?

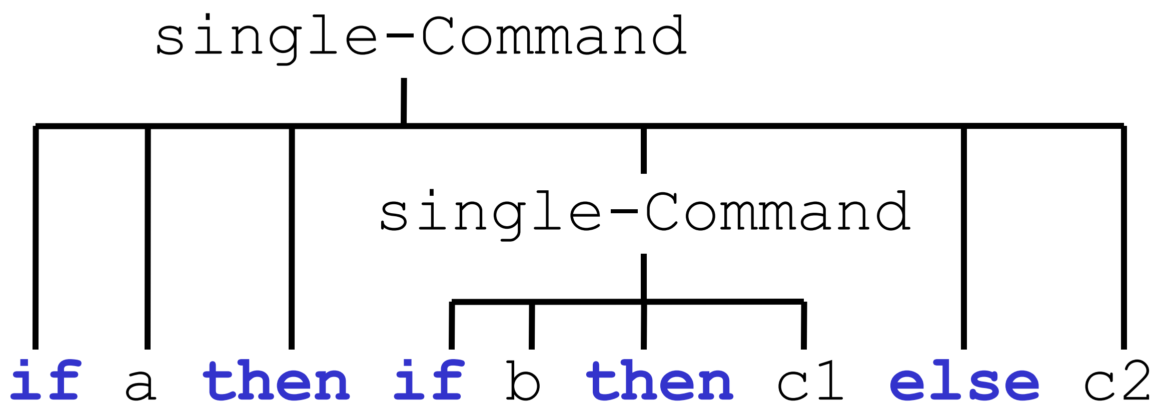


Dangling Else Problem

Example: (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                          else single-Command
```

or this one ?



Parser Conflict Resolution

Example: "dangling-else" problem (from Mini Triangle grammar)

```
            single-Command
    ::= if Expression then single-Command
      | if Expression then single-Command
                         else single-Command
```

LR(1) items (in some state of the parser)

```
 sC  ::= if E then sC • {... else ...}
sC  ::= if E then sC •  else sC {...}
```

Shift-reduce conflict!

Resolution rule: shift has priority over reduce.

Q: Does this resolution rule solve the conflict? What is its effect on the parse tree?

Dangling Else Problem

Example: "dangling-else" problem (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                          else single-Command
```

Rewrite Grammar:

```
sC ::= if E then sC endif
     |  if E then sC else sC endif
```