Languages and Compilers
(SProg og Oversættere)

Lecture 8
Bottom Up Parsing

Bent Thomsen

Department of Computer Science

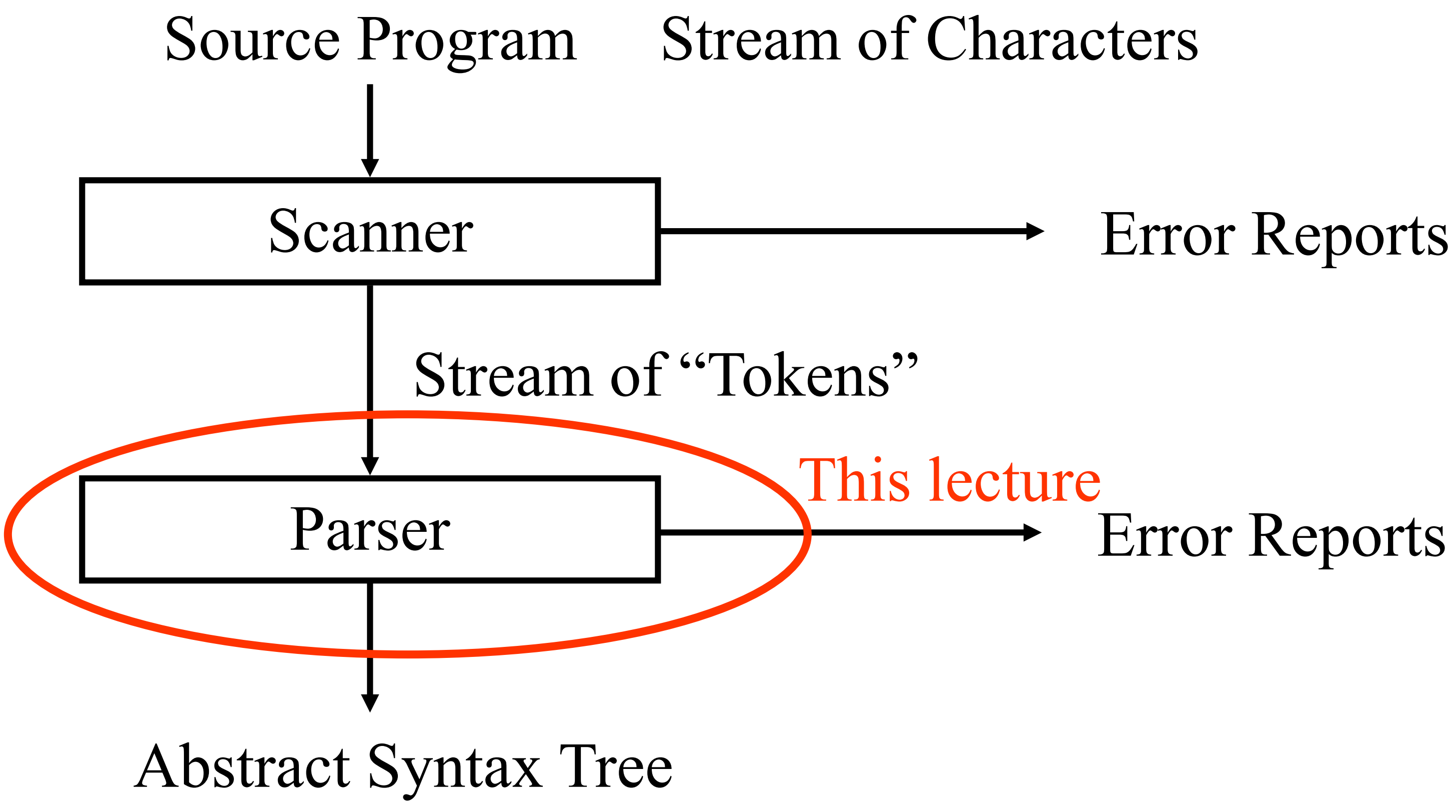
Aalborg University

Learning goals

- Get an overview of bottom up parsing
- Understand what shift/reduce and reduce/reduce conflicts are
- Get an overview of JavaCUP
- Get an overview of SableCC

Syntax Analysis

Dataflow chart



Generation of parsers

- We have seen that recursive decent parsers can be constructed by hand or automatically, e.g. JavaCC
- However, recursive decent parsers only work for LL(k) grammars
  - No Left-recursion
  - No Common prefixes (*)

  - (*) Note that the LL(*) approach used by ANTLR can deal with common prefixes, but not left recursion in general, though ANTLR4 can do some left recursion elimination.

1	Stmt	$\rightarrow$	if Expr then StmtList endif
2		|	if Expr then StmtList else StmtList endif
3	StmtList	$\rightarrow$	StmtList ; Stmt
4		|	Stmt
5	Expr	$\rightarrow$	var + Expr
6		|	var

Figure 5.12: A grammar with common prefixes.

```
procedure FACTOR ( )
    foreach A ∈ N do
        α ← LongestCommonPrefix(ProductionsFor(A))
        while |α| > 0 do
            V ← new NonTerminal ( )
            Productions ← Productions ∪ { A → αV }
            foreach p ∈ ProductionsFor(A) | RHS(p) = αβ_p do          13
                Productions ← Productions − { p }
                Productions ← Productions ∪ { V → β_p }
            α ← LongestCommonPrefix(ProductionsFor(A))
end
```

Figure 5.13: Factoring common prefixes.

1	Stmt	→	if Expr then StmtList V_1
2	V_1	→	endif
3		|	else StmtList endif
4	StmtList	→	StmtList ; Stmt
5		|	Stmt
6	Expr	→	var V_2
7	V_2	→	+ Expr
8		|	λ

Figure 5.14: Factored version of the grammar in Figure 5.12.

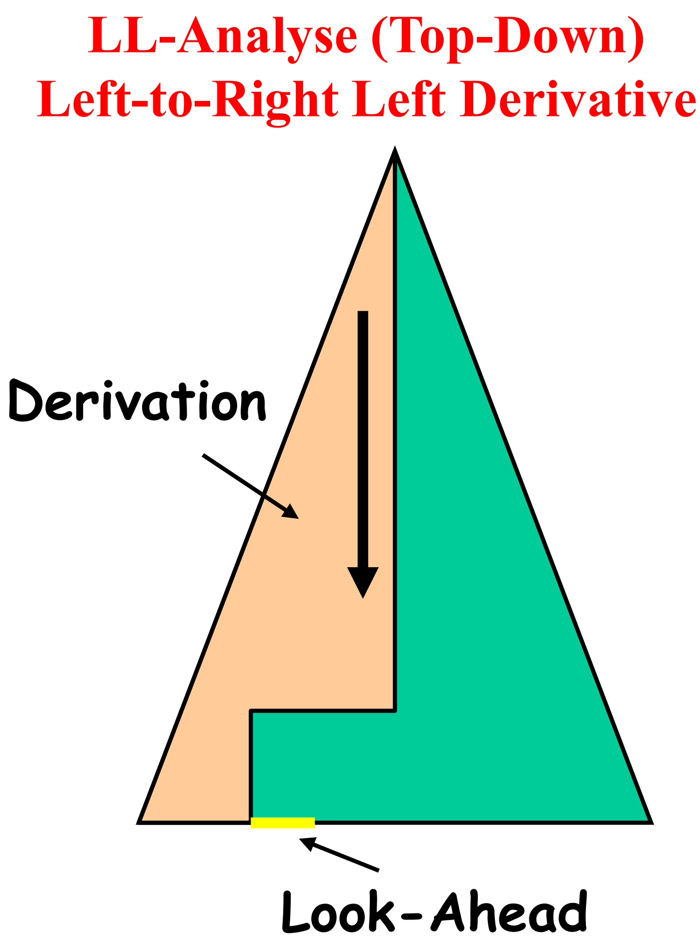
procedure ELIMINATELEFTRECURSION( )
    foreach $\mathsf{A} \in N$ **do**
        if $\exists\, r \in \mathit{ProductionsFor}(\mathsf{A}) \mid \mathit{RHS}(r) = \mathsf{A}\alpha$
        then
            $X \leftarrow$ **new** *NonTerminal*( )
            $Y \leftarrow$ **new** *NonTerminal*( )
            foreach $p \in \mathit{ProductionsFor}(\mathsf{A})$ **do**
                if $p = r$
                then $\mathit{Productions} \leftarrow \mathit{Productions} \cup \{\, \mathsf{A} \rightarrow X\ Y \,\}$
                else $\mathit{Productions} \leftarrow \mathit{Productions} \cup \{\, X \rightarrow \mathit{RHS}(p) \,\}$
            $\mathit{Productions} \leftarrow \mathit{Productions} \cup \{\, Y \rightarrow \alpha Y, Y \rightarrow \lambda \,\}$
end

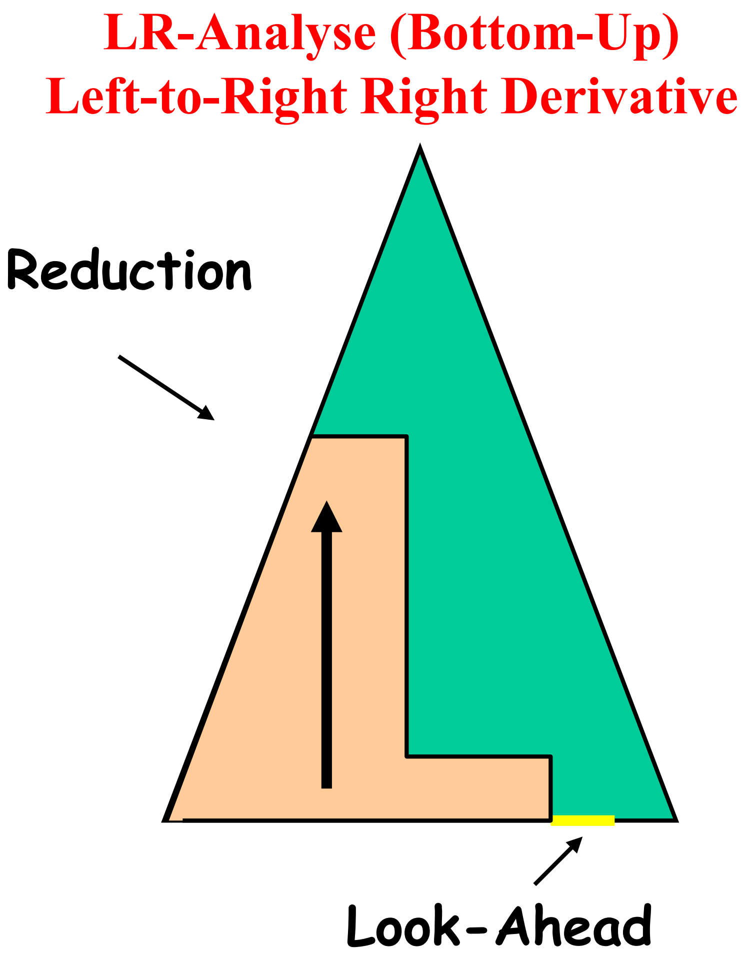
Figure 5.15: Eliminating left recursion.

1	Stmt	$\rightarrow$	if Expr then StmtList V_1
2	V_1	$\rightarrow$	endif
3		|	else StmtList endif
4	StmtList	$\rightarrow$	X Y
5	X	$\rightarrow$	Stmt
6	Y	$\rightarrow$	; Stmt Y
7		|	λ
8	Expr	$\rightarrow$	var V_2
9	V_2	$\rightarrow$	+ Expr
10		|	λ

Figure 5.16: LL(1) version of the grammar in Figure 5.14.

Top-Down vs. Bottom-Up parsing



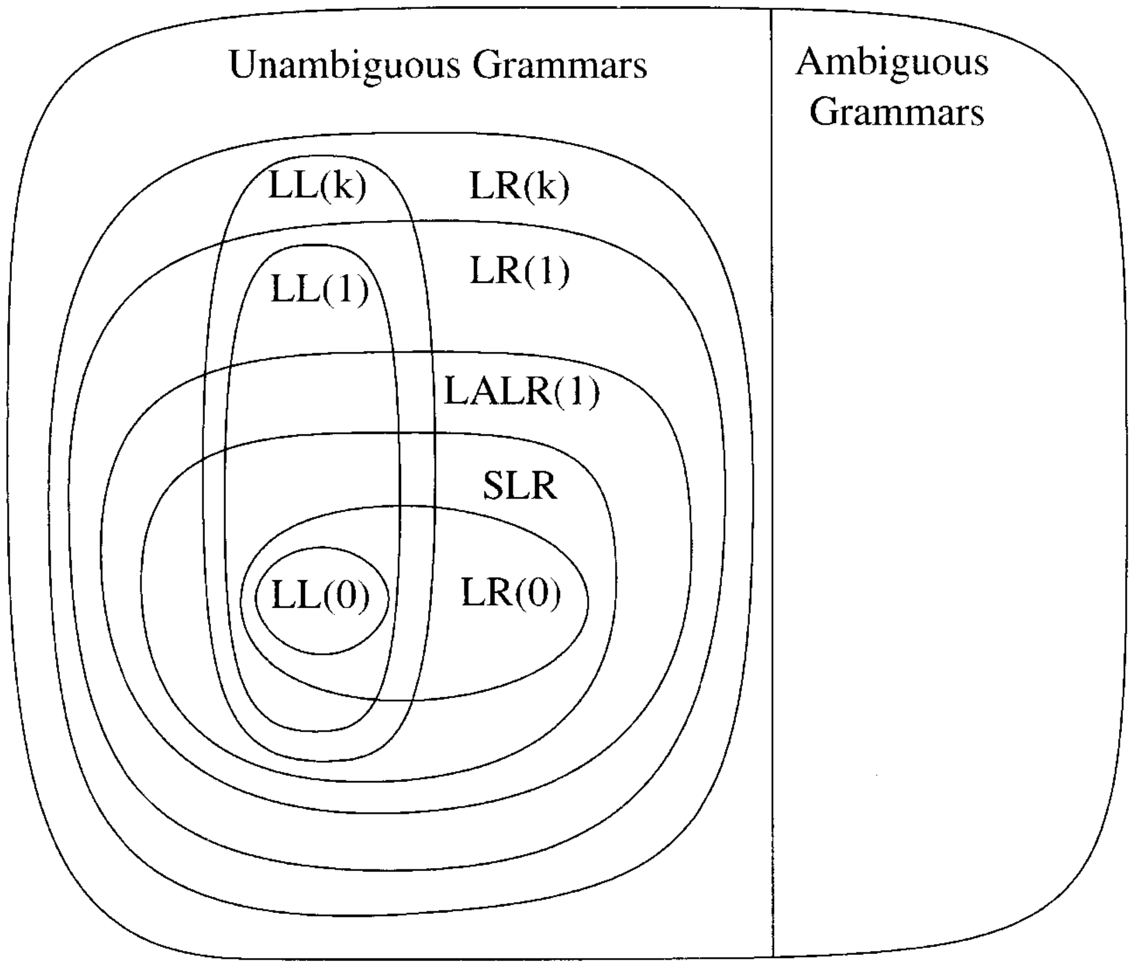


Generation of parsers

- Sometimes we need a more powerful language
- The LR languages are more powerful
  - Can recognize LR(0), SLR(1), LALR(1), LR(k) grammars
    - bigger class of grammars than LL
  - Can handle left recursion!
  - Usually more convenient because less need to rewrite the grammar.
- LR parsing methods are the most commonly used for automatic tools today (LALR in particular)
  - Parsers for LR languages use a bottom-up parsing strategy
  - Harder to implement than LL parsers
    - but tools exist (e.g. JavaCUP, Yacc, C#CUP and SableCC)

- Bottom-up parsers can handle the largest class of grammars that can be parsed deterministically

Hierarchy



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

Bottom Up Parsers: Overview of Algorithms

- LR(0) : The simplest algorithm
  - theoretically important but rather weak (not practical)
- SLR(1) : An improved version of LR(0)
  - more practical but still rather weak.
- LR(1) : LR(0) algorithm with extra lookahead token.
  - very powerful algorithm. Not often used because of large memory requirements (very big parsing tables)
  - Note: LR(0) and LR(1) use 1 lookahead taken when operating
    - 0 resp. 1 refer to token used in table construction.
- LR(k) for k>0, k tokens are use for operation and table
- LALR : "Watered down" version of LR(1)
  - still very powerful, but has much smaller parsing tables
  - most commonly used algorithm today

Fundamental idea

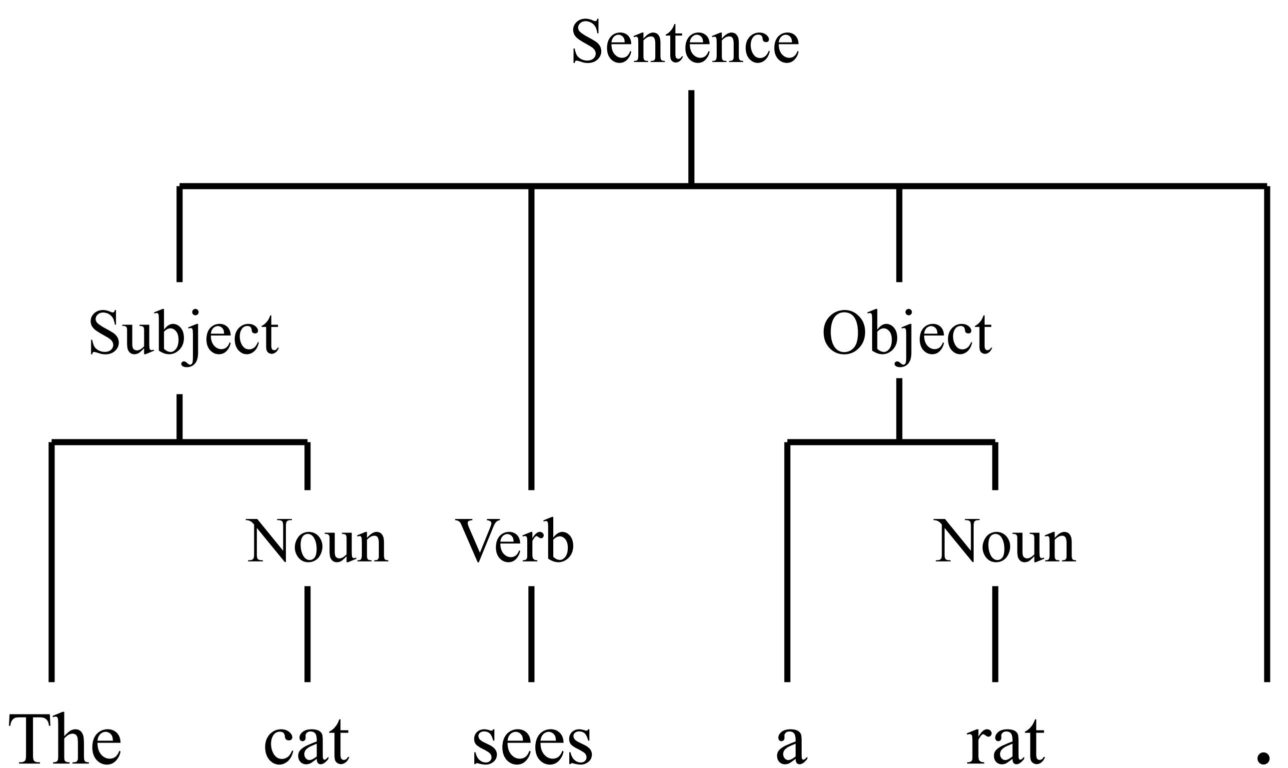
- Read through every construction and recognize the construction at the end

- LR:
  - Left – the string is read from left to right
  - Right – we get a right derivation (in reverse)

- The parse tree is build from bottom up
  - Corresponds to a right derivation in reverse

Bottom up parsing

The parse tree “grows” from the bottom (leafs) up to the top (root).



Right derivations

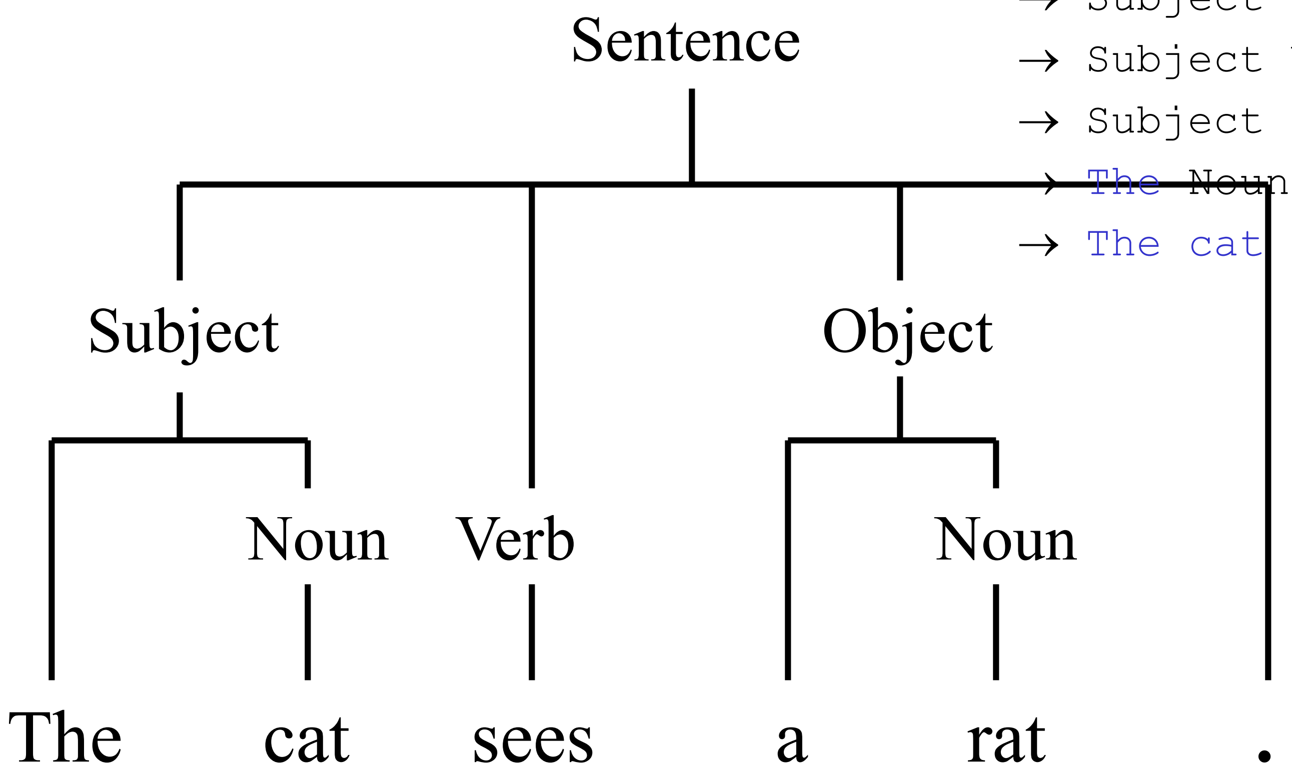
```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

```
Sentence
→ Subject Verb Object .
→ Subject Verb a Noun .
→ Subject Verb a rat .
→ Subject sees a rat .
→ The Noun sees a rat .
→ The cat sees a rat .
```

Bottom up parsing

The parse tree "grows" from the bottom (leafs) up to the top (root).
Just read the right derivations backwards

```
Sentence
→ Subject Verb Object .
→ Subject Verb a Noun .
→ Subject Verb a rat .
→ Subject sees a rat .
→ The Noun sees a rat .
→ The cat sees a rat .
```



Some Terminology

- A Rightmost (canonical) derivation is a derivation where the rightmost nonterminal is replaced at each step. A rightmost derivation from α to β is noted $\alpha \stackrel{*}{\Rightarrow}_{rm} \beta$.
- A reduction transforms uwv to uAv if $A \rightarrow w$ is a production
- α is a right sentential form if $S \stackrel{*}{\Rightarrow}_{rm} \alpha$ with $\alpha = \beta x$ where x is a string of terminals.
- A handle of a right sentential form γ $(= \alpha\beta w)$ is a production $A \rightarrow \beta$ and a position in γ where β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ:
$$S \stackrel{*}{\Rightarrow}_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$$
    - Informally, a handle is a production we can reverse without getting stuck.
    - If the handle is $A \rightarrow \beta$, we will also call β the handle.

handles and reductions

```
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
```

```
The cat sees a rat .
→ the Noun sees a rat .
→ Subject sees a rat .
→ Subject Verb a rat .
→ Subject Verb a Noun .
→ Subject Verb Object .
→ Sentence
```

```
Handles:
Noun ::= cat
Subject ::= the Noun
Verb ::= sees
Noun ::= rat
Object ::= a Noun
Sentence  ::=
 Subject Verb Object.
```

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 → ←
```

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

The knitting games

1 Start → E \$
2 E → plus E E
3 | num

Derivation

$$\begin{aligned}\text{Start} &\Rightarrow_{rm} \text{E \$} \\ &\Rightarrow_{rm} \text{plus E E \$} \\ &\Rightarrow_{rm} \text{plus E num \$} \\ &\Rightarrow_{rm} \text{plus num num \$}\end{aligned}$$

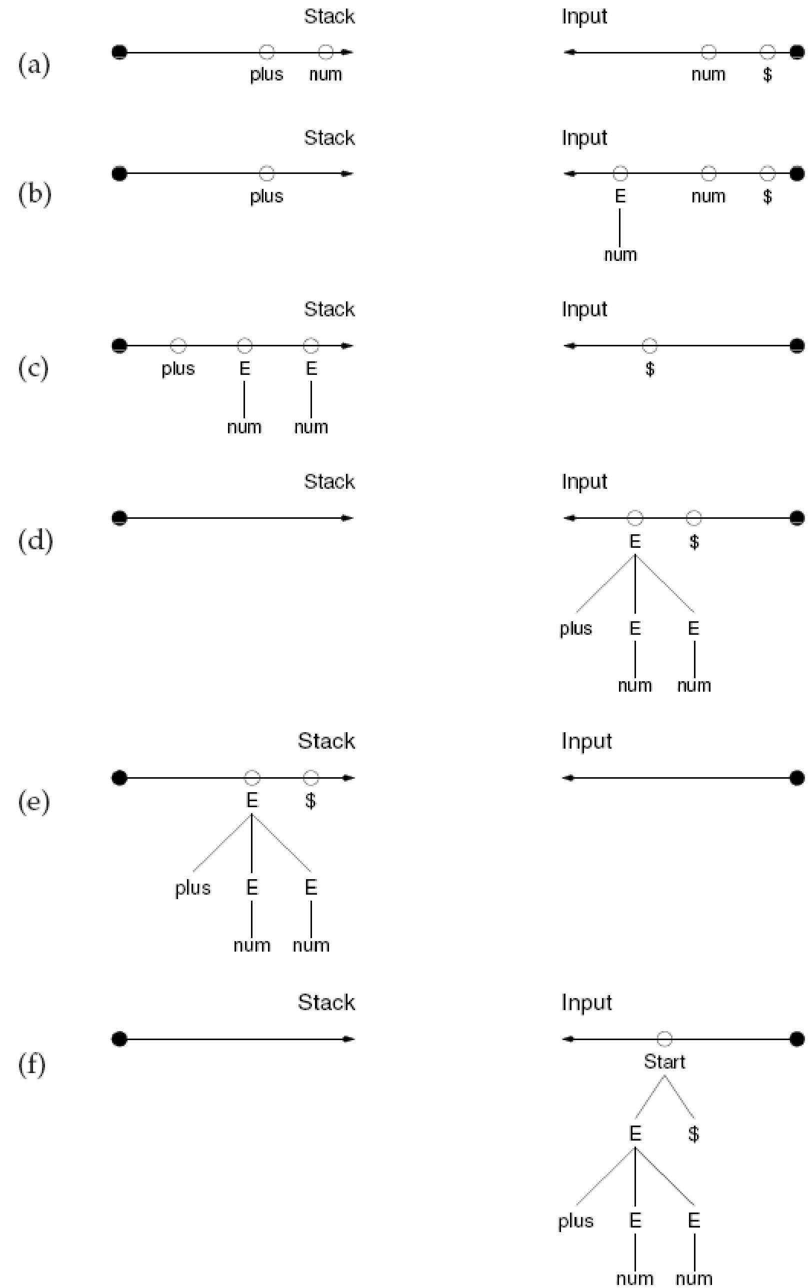


Figure 6.1: Bottom-up parsing resembles knitting.

Bottom Up Parsing

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

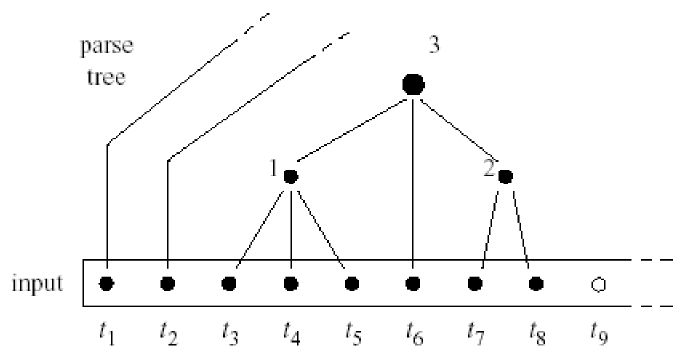


Figure 2.52 A bottom-up parser constructing its first, second, and third nodes.

(1,6,2) is a handle

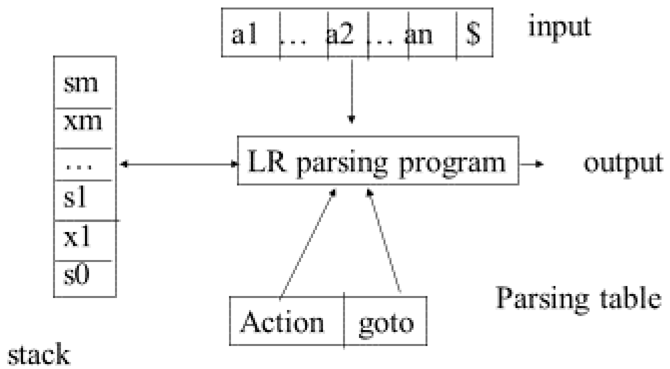
Bottom Up Parsers

- 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**
      - The sequence of children is built by pushing also called **shifting** elements on a 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.

The LR-parse algorithm

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

Bottom-up Parsing

- Shift-Reduce Algorithms
  - **Shift** is the action of moving the next token to the top of the parse stack (and record the state)
  - **Reduce** is the action of replacing the handle on the top of the parse stack with its corresponding LHS

  - Note: In Fischer et. al. the reduce action is a two step process where the LHS is prepended the input stream first and next is shifted to the parse stack (remember the knitting game)

The parse table

- For every state and every terminal
  - either shift x

    Put next input-symbol on the stack and go to state x
  - or reduce production

    On the stack we now have symbols to go backwards in the production – afterwards do a goto
- For every state and every non-terminal
  - Goto x

    Tells us, in which state to be in after a reduce-operation

    (Note as Fischer et. al. prepends non-terminals to input, they have a shift/goto action in their tables)
- Empty cells in the table indicate an error

```
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.

Example Grammar

- (0) S’ $\rightarrow$ S$
  - This production *augments* the grammar
- (1) S $\rightarrow$ (S)S
- (2) S $\rightarrow$ ε

- This grammar generates all expressions of matching parentheses

Example - parse table

	(	)	$	*S'*	*S*
0	s2	r2	r2		g1
1		s3	r0		
2	s2	r2	r2		g3
3		s4			
4	s2	r2	r2		g5
5		r1	r1		

By reduce we indicate the number of the production
r0 = accept
Never a goto by S'

Example – parsing

Stack	Input	Action
$\$_0$	()()$	shift 2
$\$_0(_2$	)()$	reduce $S \rightarrow \varepsilon$
$\$_0(_2S_3$	)()$	shift 4
$\$_0(_2S_3)_4$	()$	shift 2
$\$_0(_2S_3)_4(_2$	)$	reduce $S \rightarrow \varepsilon$
$\$_0(_2S_3)_4(_2S_3$	)$	shift 4
$\$_0(_2S_3)_4(_2S_3)_4$	$	reduce $S \rightarrow \varepsilon$
$\$_0(_2S_3)_4(_2S_3)_4S_5$	$	reduce $S \rightarrow (S)S$
$\$_0(_2S_3)_4S_5$	$	reduce $S \rightarrow (S)S$
$\$_0S_1$	$	reduce $S' \rightarrow S$

- (0) $S' \rightarrow S\$$
  - This production *augments* the grammar
- (1) $S \rightarrow (S)S$
- (2) $S \rightarrow \varepsilon$

	(	)	$	*S'*	*S*
0	s2	r2	r2		g1
1		s3	r0		
2	s2	r2	r2		g3
3		s4			
4	s2	r2	r2		g5
5		r1	r1		

The resultat

- Read the productions backwards and we get a right derivation:
- $S' \Rightarrow S \Rightarrow (S)S \Rightarrow (S)(S)S$
  $\Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ()()$

- (0) $S' \rightarrow S\$$
  - This production *augments* the grammar
- (1) $S \rightarrow (S)S$
- (2) $S \rightarrow \varepsilon$

LR(0)-DFA

- How do we get the parse table?
- We build a DFA and encode it in a table!
  - Every state is a set of items
  - Transitions are labeled by symbols
  - States must be *closed*
  - New states are constructed from states and transitions

LR(0)-items

Item :

A production with a selected position marked by a point

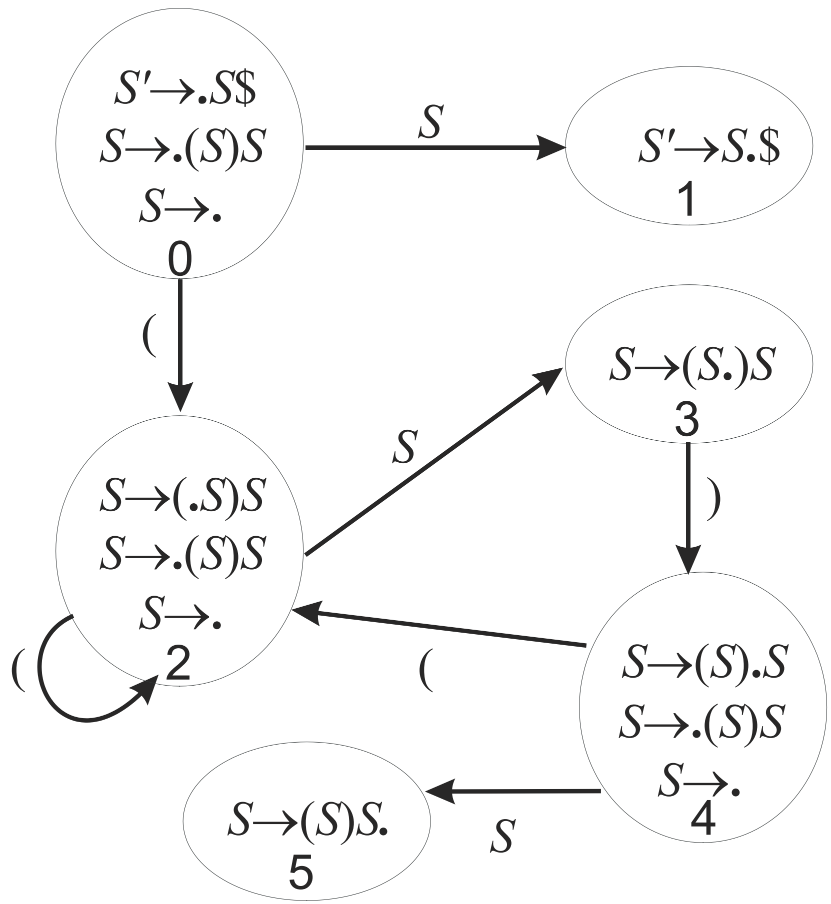
$X \rightarrow \alpha.\beta$ indicates that on the stack we have α and the first of the input can be derived from β

Our example grammar has the following items:

$S' \rightarrow .S\$$	$S' \rightarrow S.\$$	$(S' \rightarrow S\$.)$
$S \rightarrow .(S)S$	$S \rightarrow (.S)S$	$S \rightarrow (S.)S$
$S \rightarrow (S).S$	$S \rightarrow (S)S.$	$S \rightarrow .$

Rules with . at the end are the handles

The DFA for our grammar



```
function ComputeLR0(Grammar) returns (Set, State)
    States ← ∅
    StartItems ← { Start → • RHS(p) | p ∈ ProductionsFor(Start) }   7
    StartState ← AddState(States, StartItems)
    while (s ← WorkList.ExtractElement()) ≠ ⊥ do                    8
        call ComputeGoto(States, s)
    return ((States, StartState))
end
function AddState(States, items) returns State
    if items ∉ States                                               9
    then
        s ← newState(items)                                         10
        States ← States ∪ { s }
        WorkList ← WorkList ∪ { s }                                 11
        Table[s][★] ← error                                         12
    else   s ← FindState(items)
    return (s)
end
function AdvanceDot(state, X) returns Set
    return ({ A → αX • β | A → α • Xβ ∈ state })                    13
end
```

Figure 6.9: LR(0) construction.

```
function Closure(state) returns Set
    ans ← state
    repeat                                                        (14)
        prev ← ans
        foreach A → α • Bγ ∈ ans do                               (15)
            foreach p ∈ ProductionsFor(B) do
                ans ← ans ∪ { B → • RHS(p) }                      (16)
    until ans = prev
    return (ans)
end
procedure ComputeGoto(States, s)
    closed ← Closure(s)                                           (17)
    foreach X ∈ (N ∪ Σ) do                                        (18)
        RelevantItems ← AdvanceDot(closed, X)                     (19)
        if RelevantItems ≠ ∅
        then
            Table[s][X] ← shift AddState(States, RelevantItems)   (20)
end
```

Figure 6.10: LR(0) closure and transitions.

```
procedure CompleteTable( Table, grammar )
    call ComputeLookahead( )
    foreach state ∈ Table do
        foreach rule ∈ Productions(grammar) do
            call TryRuleInState( state, rule )
    call AssertEntry( StartState, GoalSymbol, accept )                21
end
procedure AssertEntry( state, symbol, action )
    if Table[state][symbol] = error                                   22
    then  Table[state][symbol] ← action
    else
        call ReportConflict( Table[state][symbol], action )           23
end
```

Figure 6.13: Completing an LR(0) parse table.

```
procedure ComputeLookahead( )
    /*  Reserved for the LALR(k) computation given in Section 6.5.2  */
end
procedure TryRuleInState( s, r )
    if LHS( r ) → RHS( r ) • ∈ s
    then
        foreach X ∈ (Σ ∪ N) do  call AssertEntry( s, X, reduce r )
end
```

Figure 6.14: LR(0) version of TryRuleInState.

Pause

Shift-reduce-conflicts

- What happens, if there is a shift and a reduce in the same cell
  - so we have a shift-reduce-conflict
  - and the grammar is not LR(0)

- Our example grammar is not LR(0)

  - (0) $S' \rightarrow S\$$
    - This production *augments* the grammar
  - (1) $S \rightarrow (S)S$
  - (2) $S \rightarrow \varepsilon$

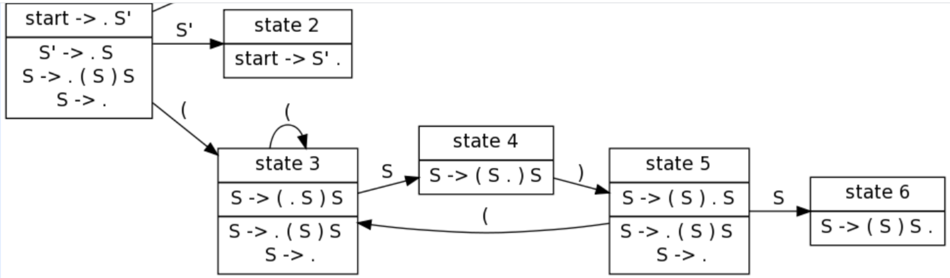
Shift-reduce-conflicts

	(	)	$	*S'*	*S*
0	**s2/r2**	r2	r2		g1
1	r0	**s3/r0**	r0		
2	**s2/r2**	r2	r2		g3
3		s4			
4	**s2/r2**	r2	r2		g5
5	r1	r1	r1		

http://smlweb.cpsc.ucalgary.ca/

http://smlweb.cpsc.ucalgary.ca/

Not secure | smlweb.cpsc.ucalgary.ca/lr0.php?grammar=S%27+->+S+.%0AS+->+%28+S+%29+S%0A+++%7C++.%0A&substs=



Grammar

S' → S .
S → (S) S
| .

LR(0) Table

	$	)	(	S'	S
0	r(S → &epsilon)	r(S → &epsilon)	r(S → &epsilon)/s3	s2	s1
1	r(S' → S)	r(S' → S)	r(S' → S)		
2	acc	acc	acc		
3	r(S → &epsilon)	r(S → &epsilon)	r(S → &epsilon)/s3		s4
4		s5			
5	r(S → &epsilon)	r(S → &epsilon)	r(S → &epsilon)/s3		s6
6	r(S → (S) S)	r(S → (S) S)	r(S → (S) S)		

SLR(1) Table

	$	)	(	S'	S
0	r(S → &epsilon)	r(S → &epsilon)	s3	s2	s1
1	r(S' → S)				
2	acc				
3	r(S → &epsilon)	r(S → &epsilon)	s3		s4
4		s5			
5	r(S → &epsilon)	r(S → &epsilon)	s3		s6
6	r(S → (S) S)	r(S → (S) S)			

The grammar is not LR(0) because:

- shift/reduce conflict in state 0.

LR(0) Conflicts

The LR(0) algorithm doesn’t always work. Sometimes there are “problems” with the grammar causing LR(0) conflicts.

An LR(0) conflict is a situation (DFA state) in which there is more than one possible action for the algorithm.

More precisely there are two kinds of conflicts:

- Shift-reduce
  - When the algorithm cannot decide between a shift action or a reduce action
- Reduce-reduce
  - When the algorithm cannot decide between two (or more) reductions (for different grammar rules).

LR(0) vs. SLR(1)

- LR(0) - when constructing the parse table, we do not look at the next symbol in the input before we decide whether to shift or to reduce
  - Note that we do use the next symbol in the input when looking up in the parse table

- SLR(1) - here we do look at the next symbol
- the parse table is a bit different:
  - shift and goto as with LR(0)
  - reduce $X \rightarrow \alpha$ only in cells (X,w) with $w \in \text{follow}(X)$
  - this means fewer reduce-actions and therefore this rule removes at lot of potential s/r- or r/r-conflicts

procedure TryRuleInState(s, r)

  if $\text{LHS}(r) \rightarrow \text{RHS}(r) \bullet \in s$

  then

    foreach $\mathcal{X} \in \mathsf{Follow}(\text{LHS}(r))$ **do**

      call AssertEntry$(s, \mathcal{X}, \mathsf{reduce}\ \mathrm{r})$

end

Figure 6.23: SLR(1) version of TryRuleInState.

LR(1)

- Items are now pairs ($A \to \alpha.\beta$, t)
  - t is a terminal such that $t \in \text{follow}(A)$
  - means that the top of the stack is α and the input can be derived from βt

  - The initial state is generated from ($S' \to .S\$$, ?)
  - Closure-operation is different
  - Shift and Goto is (more or less) the same
  - state I with item ($A \to \alpha.$, z) gives a reduce $A \to \alpha$ in cell (I,z)

  - LR(1)-parse tables are very big

Marker 7**:** We initialize *StartItems* by including LR(1) items that have \$ as the follow symbol:

$StartItems \leftarrow \{ [\,\mathsf{Start} \rightarrow \bullet \, \mathrm{RHS}(p)\,,\$\,] \mid p \in \textsc{ProductionsFor}(\,\mathsf{Start}\,) \}$

Marker 13**:** We augment the LR(0) item so that AdvanceDot returns the appropriate LR(1) items:

return $(\{ [\,\mathsf{A} \rightarrow \alpha \mathcal{X} \bullet \beta\,,\mathsf{a}\,] \mid [\,\mathsf{A} \rightarrow \alpha \bullet \mathcal{X}\beta\,,\mathsf{a}\,] \in state \})$

Marker 15**:** This entire loop is replaced by the following:

foreach $[\,\mathsf{A} \rightarrow \alpha \bullet \mathsf{B}\gamma\,,\mathsf{a}\,] \in ans$ **do**
    foreach $p \in \textsc{ProductionsFor}(B)$ **do**
        foreach $b \in \mathsf{First}(\gamma \mathsf{a})$ **do** 31
            $ans \leftarrow ans \cup \{ [\,\mathsf{B} \rightarrow \bullet \, \mathrm{RHS}(p)\,,b\,] \}$

Figure 6.38: Modifications to Figures 6.9 and 6.10 to obtain an LR(1) parser

procedure TryRuleInState(s, r)
    if $[\,\mathrm{LHS}(r) \rightarrow \mathrm{RHS}(r) \bullet \;, w\,] \in s$
    then call AssertEntry$(s, w,$ reduce r$)$
end

Figure 6.39: LR(1) version of TryRuleInState.

Example

0: $S' \rightarrow S\$$

1: $S \rightarrow V{=}E$

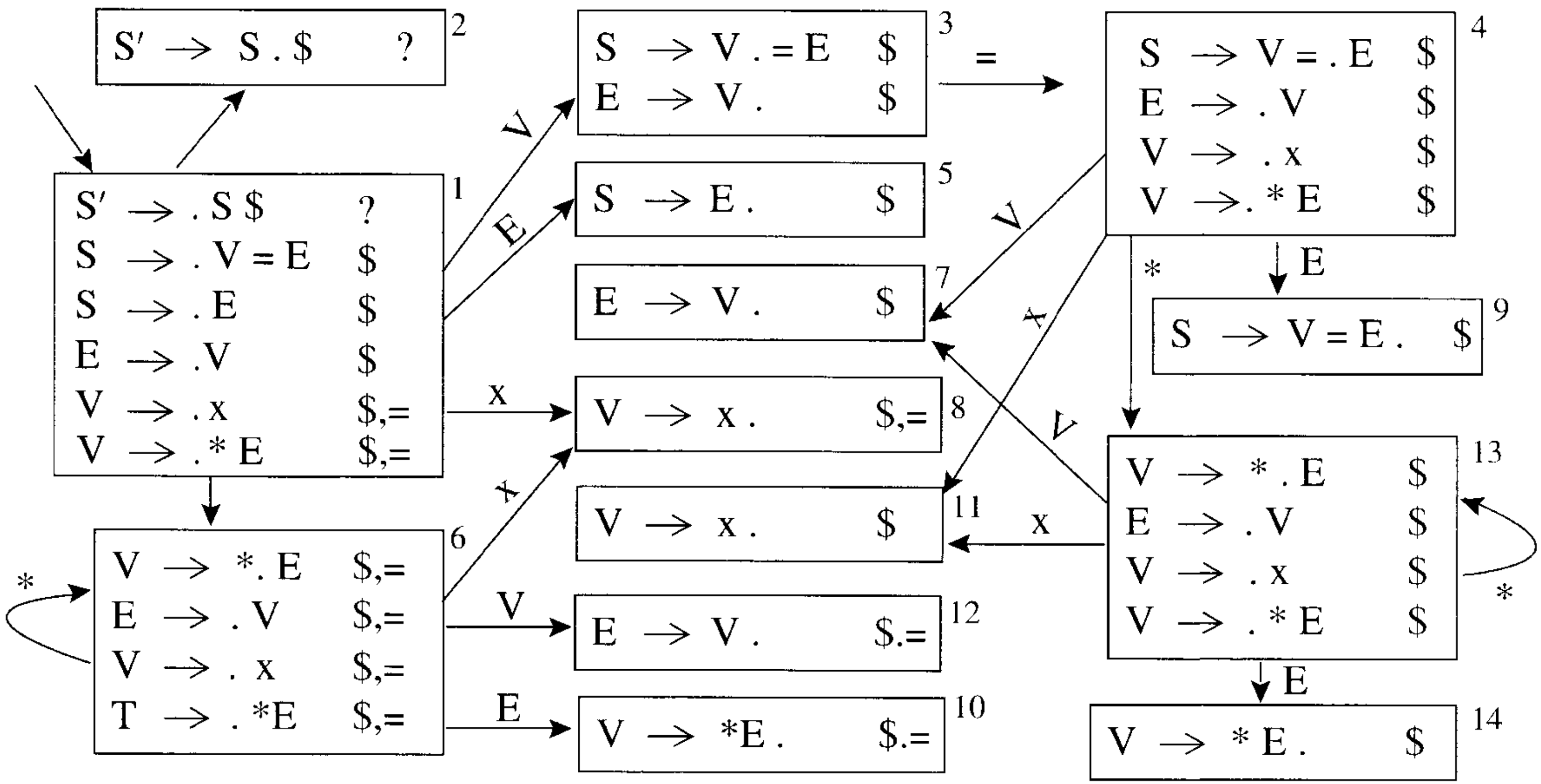
2: $S \rightarrow E$

3: $E \rightarrow V$

4: $V \rightarrow x$

5: $V \rightarrow {*}E$

LR(1)-DFA


S′ → S . $?
2
S → V . = E $
E → V . $
3
=
S → V = . E $
E → . V $
V → . x $
V → . * E $
4
S′ → . S $?
S → . V = E $
S → . E $
E → .V $
V → . x $,=
V → .* E $,=
1
V
E
S → E . $
5
E → V . $
7
V
x
*
E
S → V = E . $
9
x
V → x . $,=
8
V
V → * . E $
E → . V $
V → . x $
V → . * E $
13
x
V → x . $
11
*
V → *. E $,=
E → . V $,=
V → . x $,=
T → . *E $,=
6
*
V
E → V . $.=
12
E
V → *E . $.=
10
E
V → * E . $
14

LR(1)-parse table

	x	*	=	$	S	E	V		x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3	8			r4	r4			
2				acc				9				r1			
3			s4	r3				10			r5	r5			
4	s11	s13				g9	g7	11				r4			
5				r2				12			r3	r3			
6	s8	s6				g10	g12	13	s11	s13				g14	g7
7				r3				14				r5			

LALR(1)

- A variant of LR(1) - gives smaller parse tables

- We allow ourselves in the DFA to combine states, where the items are the same except the *x*.

- In our example we combine the states
  - 6 and 13
  - 7 and 12
  - 8 and 11
  - 10 and 14

```
procedure TryRuleInState(s, r)
    if LHS(r) → RHS(r) •  ∈ s
    then
        foreach 𝒳 ∈ Σ do
            if 𝒳 ∈ ItemFollow((s, LHS(r) → RHS(r) • ))
            then call AssertEntry(s, 𝒳, reduce r)
end
```

Figure 6.27: LALR(1) version of TryRuleInState.

```
procedure ComputeLookahead( )
    call BuildItemPropGraph( )
    call EvalItemPropGraph( )
end
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex((s, item))                                24
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor(Start) do
        ItemFollow((StartState, Start → • RHS(p))) ← {$}                  25
    foreach s ∈ States do
        foreach A → α • Bγ ∈ s do                                         26
            v ← Graph.FindVertex((s, A → α • Bγ))
            call Graph.AddEdge(v, (Table[s][B], A → αB • γ))              27
            foreach (w ← (s, B → • δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First(γ)                  28
                if AllDeriveEmpty(γ)                                      29
                then call Graph.AddEdge(v, w)
end
procedure EvalItemPropGraph( )
    repeat                                                                30
        changed ← false
        foreach (v, w) ∈ Graph.Edges do
            old ← ItemFollow(w)
            ItemFollow(w) ← ItemFollow(w) ∪ ItemFollow(v)
            if ItemFollow(w) ≠ old
            then changed ← true
    until not changed
end
```

Figure 6.28: LALR(1) version of ComputeLookahead.

LALR(1)-parse-table

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				acc			
3			s4	r3			
4	s8	s6				g9	g7
5							
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

4 kinds of parsers

- 4 ways to generate the parse table
- LR(0)
  - Easy, but only a few grammars are LR(0)
- SLR(1)
  - Relativey easy, a few more grammars are SLR
- LR(1)
  - Expensive, but alle common languages are LR(1)
- LALR(1)
  - A bit difficult, but simpler and more efficient than LR(1)
  - In practice allmost all grammars are LALR(1)

Parser Conflict Resolution

Most programming language grammars are LR(1). But, in practice, you still encounter grammars which have parsing conflicts.

=> a common cause is an **ambiguous grammar**

Ambiguous grammars always have parsing conflicts (because they are ambiguous this is just unavoidable).

In practice, parser generators still generate a parser for such grammars, using a "resolution rule" to resolve parsing conflicts deterministically.

=> The resolution rule may or may not do what you want/expect

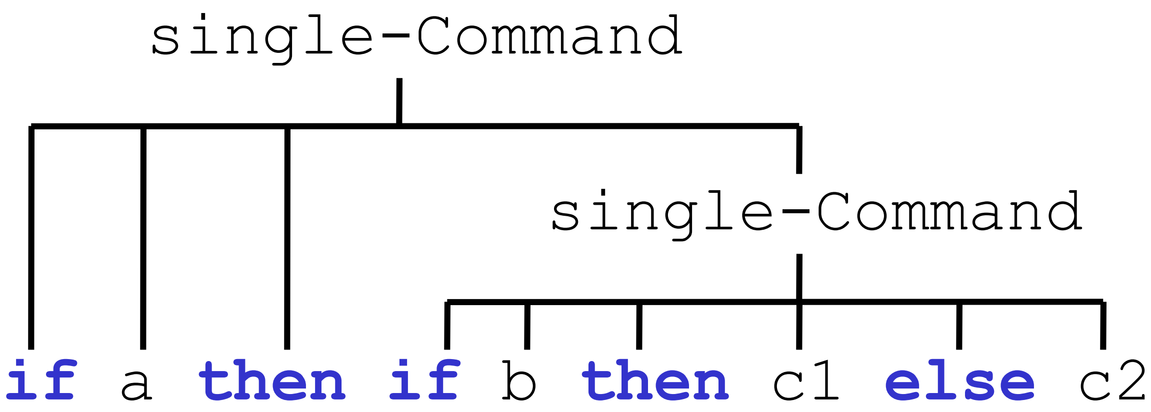
=> You will get a warning message. If you know what you are doing this can be ignored. Otherwise => try to solve the conflict by disambiguating the grammar.

Parser Conflict Resolution

Example: (from Mini Triangle grammar)

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

This parse tree?

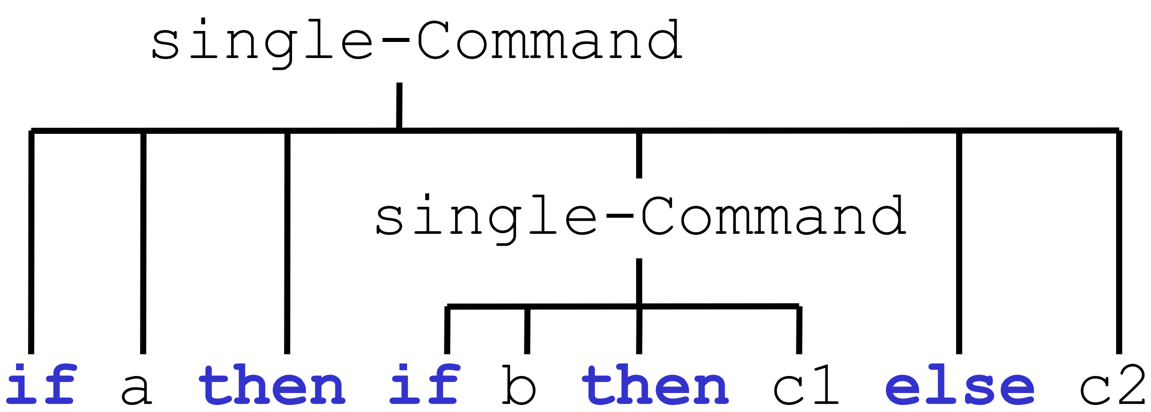


Parser Conflict Resolution

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
```

Rewrite Grammar:

```
sC   ::= CsC
       |  OsC
CsC  ::= if E then CsC else CsC
CsC  ::= ...
OsC  ::= if E then sC
       |  if E then CsC else OsC
```

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?

Parser Conflict Resolution

There is usually also a default resolution rule for shift-reduce conflicts, for example the rule which appears first in the grammar description has priority.

Reduce-reduce conflicts usually mean there is a real problem with your grammar.

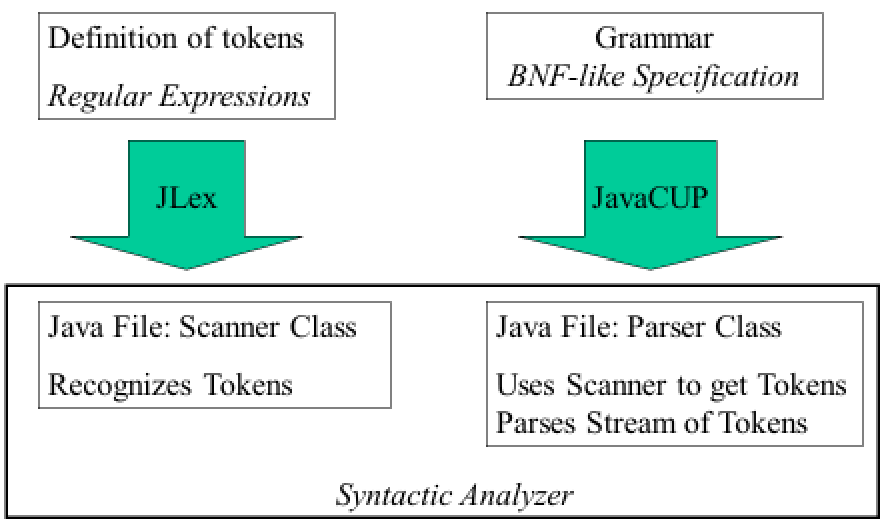
=> You need to fix it! Don’t rely on the resolution rule!

Enough background!

- All of this may sound a bit difficult (and it is)
- But it can all be automated!
- Now lets talk about tools
  - CUP (or Yacc for Java)
  - SableCC

Java Cup

- Accepts specification of a CFG and produces an LALR(1) parser (expressed in Java) with action routines expressed in Java
- Similar to yacc in its specification language, but with a few improvements (better name management)
- Usually used together with JLex (or JFlex)



Java Cup Specification Structure

```
java_cup_spec ::= package_spec
                  import_list
                  code_part
                  init_code
                  scan_code
                  symbol_list
                  precedence_list
                  start_spec
                  production_list
```

- What does it mean?
  - Package and import control Java naming
  - Code and init_code allow insertion of code in generated output
  - Scan code specifies how scanner (lexer) is invoked
  - Symbol list and precedence list specify terminal and non-terminal names and their precedence
  - Start and production specify grammar and its start point

Calculator JavaCup Specification (calc.cup)

```
terminal            PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
terminal Integer  NUMBER;
non terminal Integer expr;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
expr  ::= expr PLUS expr
        | expr MINUS expr
        | expr TIMES expr
        | expr DIVIDE expr
        | LPAREN expr RPAREN
        | NUMBER
       ;
```

- Is the grammar ambiguous?
- How can we get PLUS, NUMBER, ...?
  - They are the terminals returned by the scanner.
- How to connect with the scanner?

Ambiguous Grammar Error

- If we enter the grammar

  Expression ::= Expression PLUS Expression;

- without precedence JavaCUP will tell us:

```
Shift/Reduce conflict found in state #4
between Expression ::= Expression PLUS Expression .
and Expression ::= Expression . PLUS Expression
under symbol PLUS
Resolved in favor of shifting.
```

- The grammar is ambiguous!
- Telling JavaCUP that PLUS is left associative helps.

Evaluate the expression

- The previous specification only indicates the success or failure of a parser. No semantic action is associated with grammar rules.
- To calculate the expression, we must add java code in the grammar to carry out actions at various points.
- Form of the semantic action:

```
expr:e1 PLUS expr:e2
 {: RESULT = new Integer(e1.intValue()+ e2.intValue());    :}
```

  - Actions (java code) are enclosed within a pair {:   :}
  - Labels e2, e2: the objects that represent the corresponding terminal or non-terminal;
  - RESULT: The type of RESULT should be the same as the type of the corresponding non-terminals. e.g., expr is of type Integer, so RESULT is of type integer.

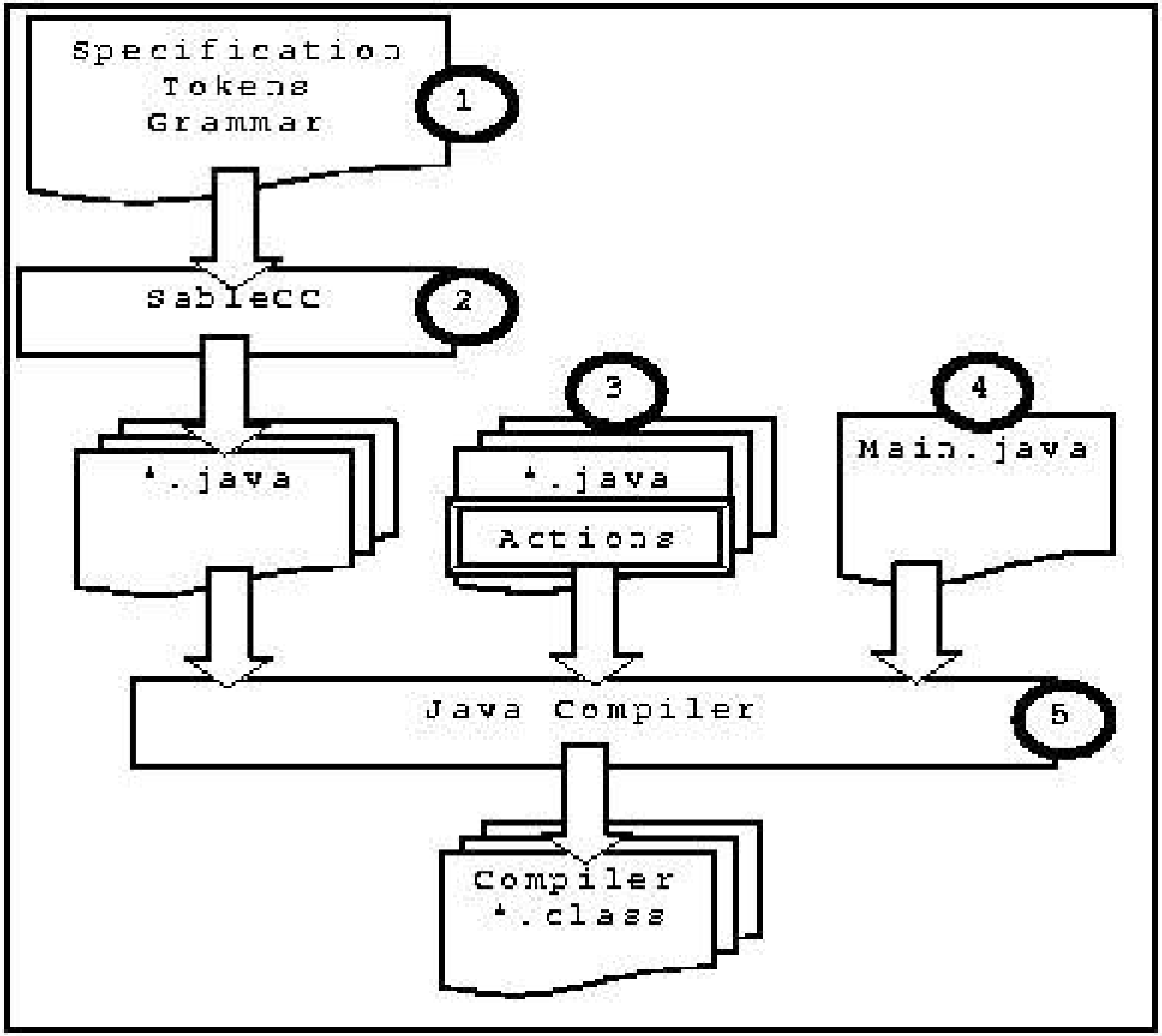
Change the calc.cup

```
terminal          PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
terminal Integer  NUMBER;
non terminal AST expr;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
expr  ::= expr:e1 PLUS expr:e2 {: RESULT = new Computing("+",e1,e2);    :}
        | expr:e1 MINUS expr:e2  {: RESULT = new Computing("-",e1,e2);  :}
        | expr:e1 TIMES expr:e2  {: RESULT = new Computing("*",e1,e2);  :}
        | expr:e1 DIVIDE expr:e2  {: RESULT = new Computing("\",e1,e2);  :}
        | LPAREN expr:e RPAREN {: RESULT = e;    :}
        | NUMBER:e {: RESULT= new IntConsting(e.intValue()); :}
```

SableCC

- Object Oriented compiler framework written in Java
  - There are also versions for C++ and C#
- Front-end compiler compiler like JavaCC and JLex/CUP
- Lexer generator based on DFA
- Parser generator based on LALR(1)
- Object oriented framework generator:
  - Strictly typed Abstract Syntax Tree
  - Tree-walker classes
  - Uses inheritance to implement actions
  - Provides visitors for user manipulation of AST
    - E.g. type checking and code generation

Steps to build a compiler with SableCC



1. Create a SableCC specification file
2. Call SableCC
3. Create one or more working classes, possibly inherited from classes generated by SableCC
4. Create a Main class activating lexer, parser and working classes
5. Compile with Javac

SableCC Example

```
Package Prog
Helpers
  digit = ['0' .. '9'];
  tab = 9;  cr = 13;  lf = 10;
  space = ' ';
  graphic = [[32 .. 127] + tab];

Tokens
  blank = (space | tab | cr | lf)* ;
  comment = '//' graphic* (cr | lf);
  while = 'while';
  begin = 'begin';
  end = 'end';
  do = 'do';
  if = 'if';
  then = 'then';
  else = 'else';
  semi = ';';
  assign = '=';
  int = digit digit*;
  id = ['a'..'z'](['a'..'z']|['0'..'9'])*;

Ignored Tokens
  blank, comment;
```

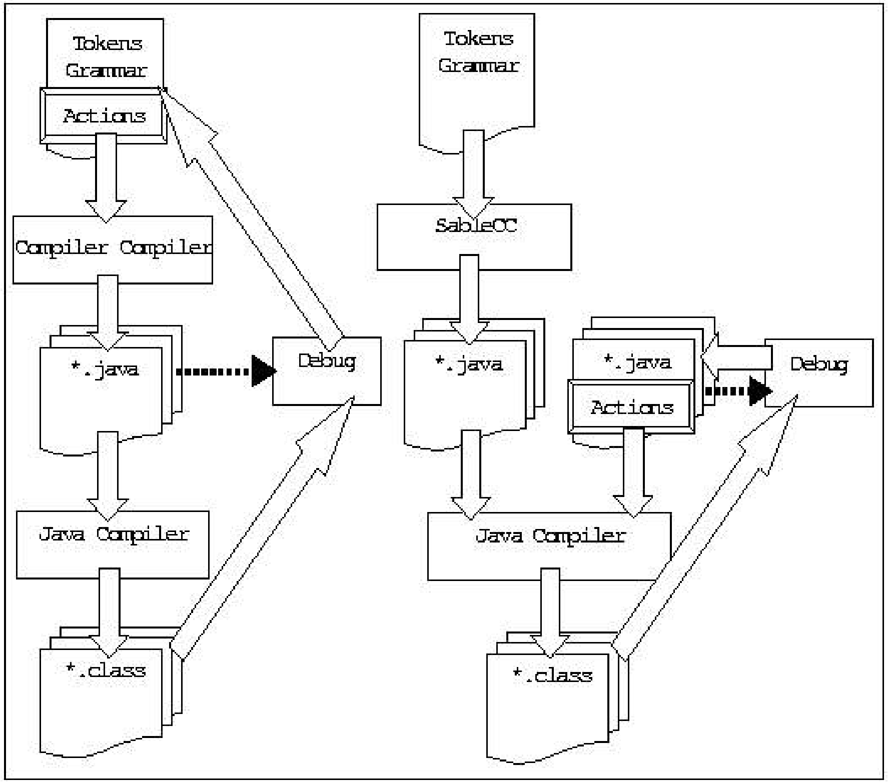
```
Productions
  prog = stmlist;

  stm = {assign} [left:]:id assign [right]:id|
        {while} while id do stm |
        {begin} begin stmlist end |
        {if_then} if id then stm;

stmlist = {stmt} stm |
          {stmtlist} stmlist semi stm;
```

SableCC output

- The *lexer* package containing the Lexer and LexerException classes
- The *parser* package containing the Parser and ParserException classes
- The *node* package contains all the classes defining typed AST
- The *analysis* package containing one interface and three classes mainly used to define AST walkers based on the visitors pattern

JLex/CUP vs. SableCC

- SableCC advantages
  - Automatic AST builder for multi-pass compilers
  - Compiler generator out of development cycle when grammar is stable
  - Easier debugging
  - Access to sub-node by name, not position
  - Clear separation of user and machine generated code
  - Automatic AST pretty-printer
  - Version 3.0 allows declarative grammar transformations



What can you do now in your projects?

- Extract a core of your language
- Define CFG for this core
  - Transform into LL(1)
  - Transform into LALR (probably not necessary)
- Build:
  - Recursive decent parser (and lexer) by hand
  - Try JavaCC and/or ANTLR
  - Try JFlex/CUP
  - Try SableCC
  - (Try other parser tools, e.g. Coco/R, Gold Parser)
- Conclude which one is most appropriate for your project