Bottom-Up Syntax Analysis

– Wilhelm/Maurer: Compiler Design, Chapter 8 –

Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de
and
Mooly Sagiv
Tel Aviv University
sagiv@math.tau.ac.il

Subjects

- Functionality and Method
- Example Parsers
- Derivation of a Parser
- Conflicts
- $LR(k)$–Grammars
- $LR(1)$–Parser Generation
- Bison

Bottom-Up Syntax Analysis

Input: A stream of symbols (tokens)

Output: A syntax tree or error

Method: **until** input consumed or error **do**

- shift next symbol or reduce by some production
- decide what to do by looking one symbol ahead

Properties

- Constructs the syntax tree in a bottom-up manner
- Finds the rightmost derivation (in reversed order)
- Reports error as soon as the already read part of the input is not a prefix of a program (valid prefix property)

Parsing $aabb$ in the grammar G_{ab} with $S \rightarrow aSb|\epsilon$

Stack	Input	Action	Dead ends
$\$$	$aabb\#$	shift	reduce $S \rightarrow \epsilon$
$\$a$	$abb\#$	shift	reduce $S \rightarrow \epsilon$
$\$aa$	$bb\#$	reduce $S \rightarrow \epsilon$	shift
$\$aaS$	$bb\#$	shift	reduce $S \rightarrow \epsilon$
$\$aaSb$	$b\#$	reduce $S \rightarrow aSb$	shift, reduce $S \rightarrow \epsilon$
$\$aS$	$b\#$	shift	reduce $S \rightarrow \epsilon$
$\$aSb$	$\#$	reduce $S \rightarrow aSb$	reduce $S \rightarrow \epsilon$
$\$S$	$\#$	accept	reduce $S \rightarrow \epsilon$

Issues:

- Shift vs. Reduce
- Reduce $A \rightarrow \beta$, Reduce $B \rightarrow \alpha\beta$

Parsing aa in the grammar $S \longrightarrow AB, S \longrightarrow A, A \longrightarrow a, B \longrightarrow a$

Stack	Input	Action	Dead ends
$\$$	$aa\#$	shift	
$\$a$	$a\#$	reduce $A \longrightarrow a$	reduce $B \longrightarrow a$, shift
$\$A$	$a\#$	shift	reduce $S \longrightarrow A$
$\$Aa$	$\#$	reduce $B \longrightarrow a$	reduce $A \longrightarrow a$
$\$AB$	$\#$	reduce $S \longrightarrow AB$	
$\$S$	$\#$	accept	

Issues:

- Shift vs. Reduce
- Reduce $A \longrightarrow \beta$, Reduce $B \longrightarrow \alpha\beta$

Shift-Reduce Parsers

- The bottom–up Parser is a shift–reduce parser, each step is
  - a shift: consuming the next input symbol or
  - a reduction: reducing a suffix of the stack contents by some production.
- the problem is to decide when to stop shifting and make a reduction instead.
- a next right side to reduce is called a "handle",

  reducing too early: dead end,

  reducing too late: burying the handle.

LR-Parsers – Deterministic Shift–Reduce Parsers

Parser decides whether to shift or to reduce based on

- the contents of the stack and
- k symbols lookahead into the rest of the input

Property of the LR–Parser: it suffices to consider the topmost state on the stack instead of the whole stack contents.

From P_G to LR–Parsers for G

- P_G has non-deterministic choice of expansions,
- LL–parsers eliminate non–determinism by looking ahead at expansions,
- LR–parsers follow all possibilities in parallel (corresponds to the subset–construction in **NFA** $\rightarrow$ **DFA**).

Derivation

1. Characteristic finite automaton of P_G, a description of P_G
2. Make deterministic
3. Interpret as control of a push down automaton
4. Check for "inedaquate" states

From P_G to LR–Parsers for G

- P_G has non-deterministic choice of expansions,
- LL–parsers eliminate non–determinism by looking ahead at expansions,
- LR–parsers follow all possibilities in parallel (corresponds to the subset–construction in **NFA** $\rightarrow$ **DFA**).

Derivation

1. Characteristic finite automaton of P_G, a description of P_G
2. Make deterministic
3. Interpret as control of a push down automaton
4. Check for "inedaquate" states

Characteristic Finite Automaton of P_G

NFA $char(P_G) = (Q_c, V_c, \Delta_c, q_c, F_c)$ — the **characteristic finite automaton** of P_G :

- $Q_c = It_G$ — states: the items of G
- $V_c = V_T \cup V_N$ — input alphabet: the sets of term. and non-term. symbols
- $q_c = [S' \to .S]$ — start state
- $F_c = \{[X \to \alpha.] \mid X \to \alpha \in P\}$ — final states: the complete items
- $\Delta_c =$
  $\{([X \to \alpha.Y\beta], Y, [X \to \alpha Y.\beta]) | X \to \alpha Y \beta \in P$ and
  $Y \in V_N \cup V_T\} \cup$
  $\{([X \to \alpha.Y\beta], \varepsilon, [Y \to .\gamma]) \mid X \to \alpha Y \beta \in P$ and $Y \to \gamma \in P\}$

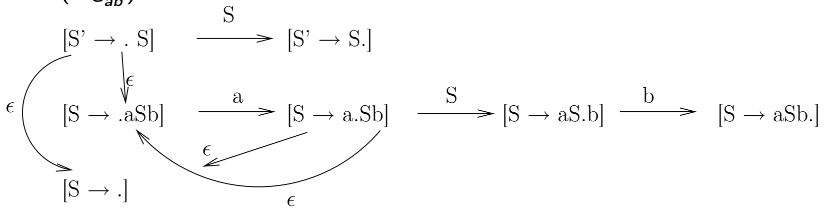
Item PDA for G_{ab}: $S \rightarrow aSb|\epsilon$

$P_{G_{ab}}$

Stack	Input	New Stack
$[S' \rightarrow .S]$	ϵ	$[S' \rightarrow .S][S \rightarrow .aSb]$
$[S' \rightarrow .S]$	ϵ	$[S' \rightarrow .S][S \rightarrow .]$
$[S \rightarrow .aSb]$	a	$[S \rightarrow a.Sb]$
$[S \rightarrow a.Sb]$	ϵ	$[S \rightarrow a.Sb][S \rightarrow .aSb]$
$[S \rightarrow a.Sb]$	ϵ	$[S \rightarrow a.Sb][S \rightarrow .]$
$[S \rightarrow aS.b]$	b	$[S \rightarrow aSb.]$
$[S \rightarrow a.Sb][S \rightarrow .]$	ϵ	$[S \rightarrow aS.b]$
$[S \rightarrow a.Sb][S \rightarrow aSb.]$	ϵ	$[S \rightarrow aS.b]$
$[S' \rightarrow .S][S \rightarrow aSb.]$	ϵ	$[S' \rightarrow S.]$
$[S' \rightarrow .S][S \rightarrow .]$	ϵ	$[S' \rightarrow S.]$

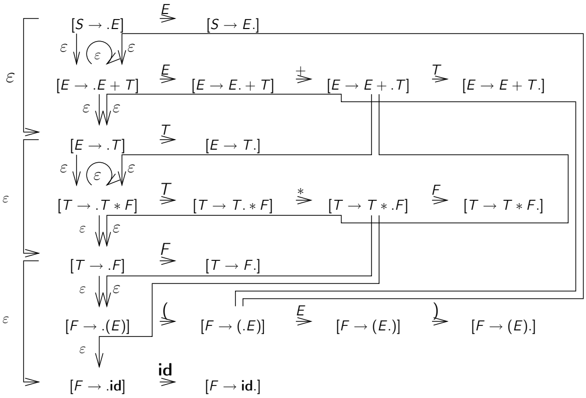
The Characteristic NFA

$char(P_{G_{ab}})$



Characteristic NFA for G_0

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



Interpreting $char(P_G)$

State of $char(P_G)$ is the *current* state of P_G, i.e. the state on top of P_G's stack. Adding actions to the transitions and states of $char(P_G)$ to describe P_G:

ε–transitions: push new state of $char(P_G)$ onto stack of P_G: new current state.

reading transitions: reading transitions of P_G: replace current state of P_G by the shifted one.

final state: Actions in P_G:

- pop final state $[X \to \alpha.]$ from the stack,
- do a transition from the new topmost state under X,
- push the new state onto the stack.

The Handle Revisited

- The bottom up–Parser is a shift–reduce–parser, each step is

a shift: consuming the next input symbol,
making a transition under it from the current state,
pushing the new state onto the stack.

a reduction: reducing a suffix of the stack contents by some production,
making a transition under the left side non–terminal from the new current state,
pushing the new state.

- the problem is the localization of the "handle", the next right side to reduce.

  reducing too early: dead end,
  reducing too late: burying the handle.

Handles and Viable Prefixes

Some Abbreviations:
RMD – rightmost derivation
RSF – right sentential form
$S' \underset{rm}{\overset{*}{\Longrightarrow}} \beta X u \underset{rm}{\Longrightarrow} \beta\alpha u$ – a RMD of cfg G.

- α is a **handle** of $\beta\alpha u$.
  The part of a RSF next to be reduced.
- Each prefix of $\beta\alpha$ is a **viable prefix**.
  A prefix of a RSF stretching at most up to the end of the handle,
  i.e. reductions if possible then only at the end.

Examples in G_0

RSF	handle	viable prefix	Reason
$E + F$	F	$E,\ E+,\ E+F$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E + T \underset{rm}{\Longrightarrow} E + F$
$T * \mathbf{id}$	$\mathbf{id}$	$T,\ T*,\ T * \mathbf{id}$	$S \overset{3}{\underset{rm}{\Longrightarrow}} T * F \underset{rm}{\Longrightarrow} T * \mathbf{id}$
$F * \mathbf{id}$	F	F	$S \overset{4}{\underset{rm}{\Longrightarrow}} T * \mathbf{id} \underset{rm}{\Longrightarrow} F * \mathbf{id}$

Valid Items

$[X \rightarrow \alpha.\beta]$ is **valid** for the viable prefix $\gamma\alpha$, if there exists a RMD $S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma X w \underset{rm}{\Longrightarrow} \gamma\alpha\beta w$.

An item valid for a viable prefix gives one interpretation of the parsing situation.

Some viable prefixes of G_0

Viable Prefix	Valid Items	Reason	γ	w	X	α	β
$E+$	$[E \rightarrow E + .T]$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E + T$	ε	ε	E	$E+$	T
	$[T \rightarrow .F]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} E + T \underset{rm}{\Longrightarrow} E + F$	$E+$	ε	T	ε	F
	$[F \rightarrow .\mathbf{id}]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} E + F \underset{rm}{\Longrightarrow} E + \mathbf{id}$	$E+$	ε	F	ε	$\mathbf{id}$
$(E+($	$[F \rightarrow (.E)]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E + F)$ $\underset{rm}{\Longrightarrow} (E + (E))$	$(E+$	$)$	F	$($	$E)$

Valid Items and Parsing Situations

Given some input string $xuvw$.
The RMD
$S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma X w \underset{rm}{\Longrightarrow} \gamma\alpha\beta w \overset{*}{\underset{rm}{\Longrightarrow}} \gamma\alpha v w \overset{*}{\underset{rm}{\Longrightarrow}} \gamma u v w \overset{*}{\underset{rm}{\Longrightarrow}} x u v w$
describes the following sequence of partial derivations:
$\gamma \overset{*}{\underset{rm}{\Longrightarrow}} x \qquad \alpha \overset{*}{\underset{rm}{\Longrightarrow}} u \qquad \beta \overset{*}{\underset{rm}{\Longrightarrow}} v \qquad X \underset{rm}{\Longrightarrow} \alpha\beta$
$S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma X w$
executed by the bottom-up parser in this order.
The valid item $[X \to \alpha\,.\,\beta]$ for the viable prefix $\gamma\alpha$ describes the situation after partial derivation 2.

Theorems

$char(P_G) = (Q_c, V_c, \Delta_c, q_c, F_c)$

Theorem

For each viable prefix there is at least one valid item.

Every parsing situation is described by at least one valid item.

Theorem

Let $\gamma \in (V_T \cup V_N)^*$ *and* $q \in Q_c$.
$(q_c, \gamma) \vdash^*_{char(P_G)} (q, \varepsilon)$ *iff* γ *is a viable prefix and* q *is a valid item for* γ.

A viable prefix brings $char(P_G)$ from its initial state to all its valid items.

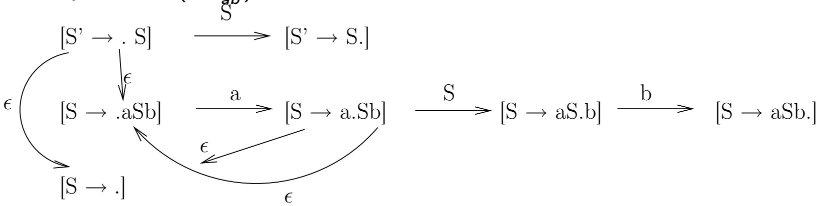
Theorem

The language of viable prefixes of a cfg is regular.

Making $char(P_G)$ deterministic

Apply **NFA** $\rightarrow$ **DFA** to $char(P_G)$: Result LR-DFA(G).

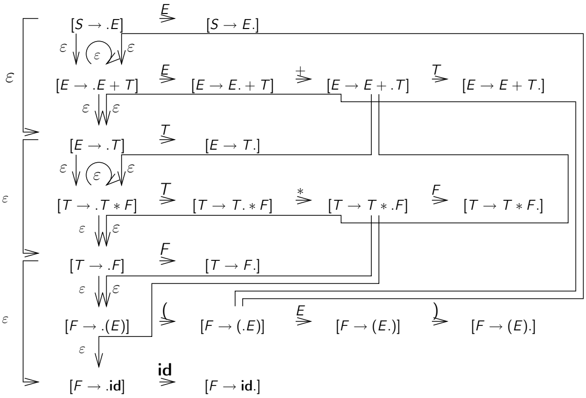
Example: $char(P_{G_{ab}})$



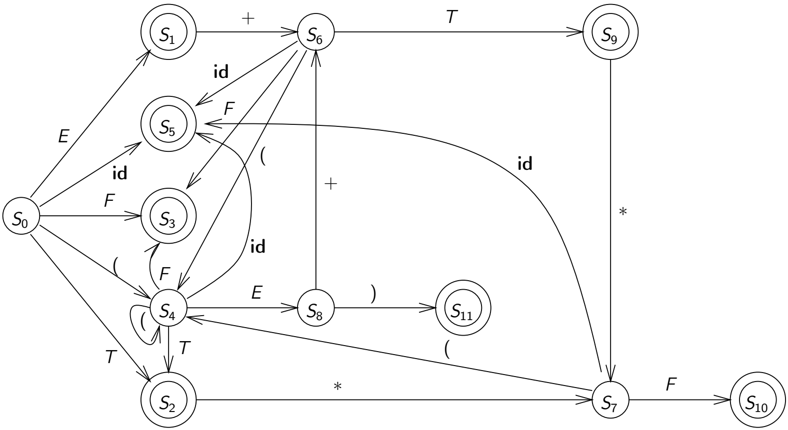
LR-DFA(G_{ab}):

Characteristic NFA for G_0

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



LR-DFA(G_0)



The States of LR-DFA(G_0) as Sets of Items

$$S_0 = \{ [S \rightarrow .E], [E \rightarrow .E + T], [E \rightarrow .T], [T \rightarrow .T * F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .\mathbf{id}] \}$$

$$S_1 = \{ [S \rightarrow E.], [E \rightarrow E. + T] \}$$

$$S_2 = \{ [E \rightarrow T.], [T \rightarrow T. * F] \}$$

$$S_3 = \{ [T \rightarrow F.] \}$$

$$S_4 = \{ [F \rightarrow (.E)], [E \rightarrow .E + T], [E \rightarrow .T], [T \rightarrow .T * F] [T \rightarrow .F] [F \rightarrow .(E)] [F \rightarrow .\mathbf{id}] \}$$

$$S_5 = \{ [F \rightarrow \mathbf{id}.] \}$$

$$S_6 = \{ [E \rightarrow E + .T], [T \rightarrow .T * F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .\mathbf{id}] \}$$

$$S_7 = \{ [T \rightarrow T * .F], [F \rightarrow .(E)], [F \rightarrow .\mathbf{id}] \}$$

$$S_8 = \{ [F \rightarrow (E.)], [E \rightarrow E. + T] \}$$

$$S_9 = \{ [E \rightarrow E + T.], [T \rightarrow T. * F] \}$$

$$S_{10} = \{ [T \rightarrow T * F.] \}$$

$$S_{11} = \{ [F \rightarrow (E).] \}$$

Theorems

$char(P_G) = (Q_c, V_c, \Delta_c, q_c, F_c)$ and
$LR - DFA(G) = (Q_d, V_N \cup V_T, \Delta, q_d, F_d)$

Theorem

Let γ be a viable prefix and $p(\gamma) \in Q_d$ be the uniquely determined state, into which LR-DFA(G) transfers out of the initial state by reading γ, i.e., $(q_d, \gamma) \vdash^*_{LR-DFA(G)} (p(\gamma), \varepsilon)$. Then

(a) $p(\varepsilon) = q_d$

(b) $p(\gamma) = \{q \in Q_c \mid (q_c, \gamma) \vdash^*_{char(P_G)} (q, \varepsilon)\}$

(c) $p(\gamma) = \{i \in It_G \mid i \text{ valid for } \gamma\}$

(d) *Let Γ the (in general infinite) set of all viable prefixes of G. The mapping $p : \Gamma \to Q_d$ defines a finite partition on Γ.*

(e) *$L(LR\text{-}DFA(G))$ is the set of viable prefixes of G, which end in a handle.*

G_0

$\gamma = E + F$ is a viable prefix of G_0.
With the state $p(\gamma) = S_3$ are also associated:
$F,\ (F,\ ((F,\ (((F,\ldots$
$T * (F,\ T * ((F,\ T * (((F,\ldots$
$E + F,\ E + (F,\ E + ((F,\ldots$
Regard S_6 in LR-DFA(G_0).
It consists of all valid items for the viable prefix $E+$,
i.e., the items
$[E \to E + .T], [T \to .T * F], [T \to .F], [F \to .\mathbf{id}], [F \to .(E)]$.
Reason:
$E+$ is prefix of the RSF $E + T$;

$$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E + T \underset{rm}{\Longrightarrow} E + F \underset{rm}{\Longrightarrow} E + \mathbf{id}$$

$$\uparrow \qquad \uparrow \qquad \uparrow$$

Therefore $[E \to E + .T]$ $[T \to .F]$ $[F \to .\mathbf{id}]$ are valid.

What the LR-DFA(G) describes

LR-DFA(G) interpreted as a PDA $P_0(G) = (\Gamma, V_T, \Delta, q_0, \{q_f\})$

Γ, (stack alphabet): the set Q_d of states of LR-DFA(G).

$q_0 = q_d$ (initial state): in the stack of $P_0(G)$ initially.

$q_f = \{[S' \to S.]\}$ the final state of LR-DFA(G),

$\Delta \subseteq \Gamma^* \times (V_T \cup \{\varepsilon\}) \times \Gamma^*$ (transition relation):

Defined as follows:

LR-DFA(G)'s Transition Relation

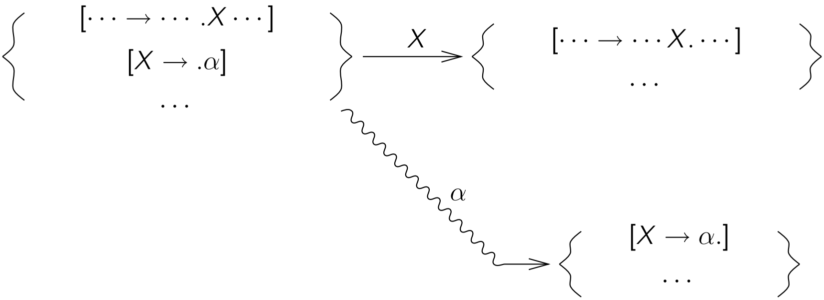
shift: $(q, a, q\,\delta_d(q, a)) \in \Delta$, if $\delta_d(q, a)$ defined.
Read next input symbol a and push successor state of q under a (item $[X \to \cdots .a \cdots] \in q$).

reduce: $(q\,q_1 \ldots q_n, \varepsilon, q\,\delta_d(q, X)) \in \Delta$,
if $[X \to \alpha.] \in q_n,\ |\alpha| = n$.
Remove $|\alpha|$ entries from the stack.
Push the successor of the new topmost state under X onto the stack.

Note the difference in the stacking behavior:

- the Item PDA P_G keeps on the stack only one item for each production under analysis,
- the PDA described by the LR-DFA(G) keeps $|\alpha|$ states on the stack for a production $X \to \alpha\beta$ represented with item $[X \to \alpha.\beta]$

Reduction in PDA $P_0(G)$



Some observations and recollections

- also works for reductions of ϵ,
- each state has a unique entry symbol,
- the stack contents uniquely determine a viable prefix,
- current state (topmost) is the state associated with this viable prefix,
- current state consists of all items valid for this viable prefix.

Non-determinism in $P_0(G)$

$P_0(G)$ is non-deterministic if either

Shift–reduce conflict: There are shift as well as reduce transitions out of one state, or

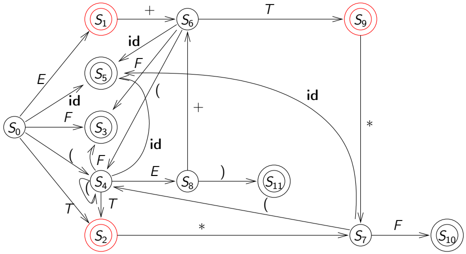
Reduce–reduce conflict: There are more than one reduce transitions from one state.

States with a shift–reduce conflict have at least one read item $[X \rightarrow \alpha\, .a\, \beta]$ and at least one complete item $[Y \rightarrow \gamma.]$.

States with a reduce–reduce conflict have at least two complete items $[Y \rightarrow \alpha.]$, $[Z \rightarrow \beta.]$.

A state with a conflict is **inadequate**.

Some Inadequate States



LR-DFA(G_0) has three inadequate states, S_1, S_2 and S_9.

- S_1: Can reduce E to S (complete item $[S \rightarrow E.]$) or read "+" (shift–item $[E \rightarrow E. + T]$);
- S_2: Can reduce T to E (complete item $[E \rightarrow T.]$) or read "$*$" (shift-item $[T \rightarrow T. * F]$);
- S_9: Can reduce $E + T$ to E (complete item $[E \rightarrow E + T.]$) or read "$*$" (shift–item $[T \rightarrow T. * F]$).

Direct Construction of the LR-DFA(G)

Algorithm LR-DFA:
Input: cfg $G = (V'_N, V_T, P', S')$
Output: LR-DFA$(G) = (Q_d, V_N \cup V_T, q_d, \delta_d, F_d)$
Method: The states and the transitions of the LR-DFA(G) are constructed using the following three functions *Start*, *Closure* and *Succ*
F_d – set of states with at least one complete item

var q, q': **set of item**;
Q_q: **set of set of item**;
δ_d: **set of item** $\times (V_N \cup V_T) \rightarrow$ **set of item**;

```
function  Start: set of item; return({[S' → .S]});
function  Closure(s : set of item) : set of item;
      (* ε-Succ states of algorithm NFA → DFA *)
begin  q := s;
   while  exists [X → α.Yβ] in  q and  Y → γ in  P
         and  [Y → .γ] not in q do
      add [Y → .γ] to q
   od;
   return(q)
end ;
function  Succ(s : set of item, Y : V_N ∪ V_T) : set of item;
   return({[X → αY.β] | [X → α.Yβ] ∈ s});
```

```
begin
    Q_d := {Closure(Start)}; (* start state *)
    δ_d := ∅;
    foreach q in Q_d and X in V_N ∪ V_T do
        let q' = Closure(Succ(q, X)) in
            if q' ≠ ∅ (* X–successor exists *)
            then
                if q' not in Q_d (* new state created *)
                then Q_d := Q_d ∪ {q'}
                fi;
                δ_d := δ_d ∪ {q --X--> q'} (* new transition *)
            fi
        tel
    od
end
```

Where: $Q_d := \{Closure(Start)\}$; $\delta_d := \emptyset$; $V_N \cup V_T$; $q' = Closure(Succ(q, X))$; $q' \neq \emptyset$; $Q_d := Q_d \cup \{q'\}$; $\delta_d := \delta_d \cup \{q \xrightarrow{X} q'\}$

LR(k)–Grammars

G – LR(k)–Grammar iff in each RMD
$S' = \alpha_0 \underset{rm}{\Longrightarrow} \alpha_1 \underset{rm}{\Longrightarrow} \alpha_2 \cdots \underset{rm}{\Longrightarrow} \alpha_m = v$
and in each RSF $\alpha_i = \gamma\beta w$

- the handle can be localized, and
- the production to be applied can be determined

by regarding the prefix $\gamma\beta$ of α_i and at most k symbols after the handle, β.
I.e., the splitting of α_i into $\gamma\beta w$ and the production $X \rightarrow \beta$, such that $\alpha_{i-1} = \gamma X w$, is uniquely determined by $\gamma\beta$ and $k : w$.

LR(k)–Grammars

Definition: A cfg G is an **LR(k)-Grammar**, iff
$S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha X w \underset{rm}{\Longrightarrow} \alpha\beta w$ and
$S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma Y x \underset{rm}{\Longrightarrow} \alpha\beta y$ and
$k : w = k : y$ implies
that $\alpha = \gamma$ and $X = Y$ and $x = y$.

Example 1

Cfg G_{nLL} with the productions

$$\begin{array}{lcl} S & \rightarrow & A \mid B \\ A & \rightarrow & aAb \mid 0 \\ B & \rightarrow & aBbb \mid 1 \end{array}$$

- $L(G) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$.
- G_{nLL} is not LL(k) for arbitrary k, but G_{nLL} is LR(0)-grammar.
- The RSFs of G_{nLL} (handle)
  - $S, \underline{A}, \underline{B}$,
  - $a^n \underline{aBbb} b^{2n}, a^n \underline{aAb} b^n$,
  - $a^n a \underline{0} b b^n, a^n a \underline{1} bbb^{2n}$.

Example 1 (cont'd)

- Only $a^n aAbb^n$ and $a^n aBbbb^{2n}$ allow 2 different reductions.
  - reduce $\overbrace{a^n}^{\gamma}\overbrace{aAb}^{\beta}b^n$ to $a^n Ab^n$: part of a RMD $S \underset{rm}{\overset{*}{\Longrightarrow}} a^n Ab^n \underset{rm}{\Longrightarrow} a^n aAbb^n$,
  - reduce $a^n aAbb^n$ to $a^n aSbb^n$: not part of any RMD.
- The prefix a^n of $a^n Ab^n$ uniquely determines, whether
  - A is the handle ($n = 0$), or
  - whether aAb is the handle ($n > 0$).
- The RSFs $a^n Bb^{2n}$ are treated analogously.

Example 2

Cfg G_1 with
$S \rightarrow aAc$
$A \rightarrow Abb \mid b$

- $L(G_1) = \{ab^{2n+1}c \mid n \geq 0\}$
- G_1 is LR(0)–grammar.

RSF $\overbrace{a}^{\gamma}\overbrace{Abb}^{\beta}b^{2n}c$: only legal reduction is to $aAb^{2n}c$,
uniquely determined by the prefix $aAbb$.

RSF $\overbrace{a}^{\gamma}\overbrace{b}^{\beta}b^{2n}c$: b is the handle,
uniquely determined by the prefix ab.

Example 3

Cfg G_2 with
$S \rightarrow aAc$
$A \rightarrow bbA \mid b$.

- $L(G_2) = L(G_1)$
- G_2 is LR(1)–grammar.
- Critical RSF $ab^n w$.
  - $1 : w = b$ implies, handle in w;
  - $1 : w = c$ implies, last b in b^n is handle.

Example 4

Cfg G_3 with $S \rightarrow aAc \qquad A \rightarrow bAb \mid b$.

- $L(G_3) = L(G_1)$,
- G_3 is not LR(k)–grammar for arbitrary k.

Choose an arbitrary k.

Regard two RMDs

$$S \overset{*}{\underset{rm}{\Longrightarrow}} ab^nAb^nc \underset{rm}{\Longrightarrow} ab^nbb^nc$$

$$S \overset{*}{\underset{rm}{\Longrightarrow}} ab^{n+1}Ab^{n+1}c \underset{rm}{\Longrightarrow} ab^{n+1}bb^{n+1}c \quad \text{where } n \geq k$$

Choose $\alpha = ab^n, \beta = b, \gamma = ab^{n+1}, w = b^nc, y = b^{n+2}c$.

It holds $k : w = k : y = b^k$.

$\alpha \neq \gamma$ implies that G_3 is not an LR(k)–grammar.

Adding Lookahead

Lookahead will be used to resolve conflicts.

- $[X \rightarrow \alpha_1.\alpha_2, L]$ – **LR(k)–item**, if $X \rightarrow \alpha_1\alpha_2 \in P$ and $L \subseteq V_{T\#}^{\leq k}$.
- $[X \rightarrow \alpha_1.\alpha_2]$ – **core** of $[X \rightarrow \alpha_1.\alpha_2, L]$,
- L – the **lookahead set** of $[X \rightarrow \alpha_1.\alpha_2, L]$.
- $[X \rightarrow \alpha_1.\alpha_2, L]$ is **valid** for a viable prefix $\alpha\alpha_1$, if for all $u \in L$ there is a RMD $S'\# \underset{rm}{\overset{*}{\Longrightarrow}} \alpha X w \underset{rm}{\Longrightarrow} \alpha\alpha_1\alpha_2 w$ with $u = k : w$.

The context–free items can be regarded as LR(0)-items if $[X \rightarrow \alpha_1.\alpha_2, \{\varepsilon\}]$ is identified with $[X \rightarrow \alpha_1.\alpha_2]$.

Example from G_0

(1) $[E \rightarrow E + .T, \{), +, \#\}]$ is a valid LR(1)–item for $(E+$
(2) $[E \rightarrow T., \{*\}]$ is not a valid LR(1)-item for any viable prefix

Reason:
(1) $S' \overset{*}{\underset{rm}{\Longrightarrow}} (E) \underset{rm}{\Longrightarrow} (E + T) \overset{*}{\underset{rm}{\Longrightarrow}} (E + T + \mathbf{id})$ where

$$\alpha = (, \;\alpha_1 = E+, \;\alpha_2 = T, \;u = +, \;w = +\mathbf{id})$$

(2) The string $E*$ can occur in no RMD.

LR–Parser

Take their decisions (to shift or to reduce) by consulting

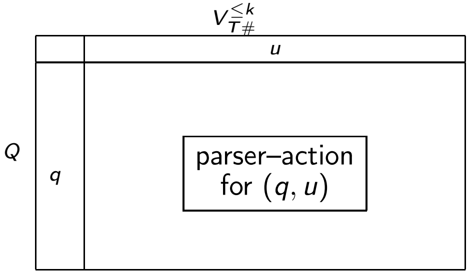
- the viable prefix γ in the stack, actually the by γ uniquely determined state (on top of the stack),
- the next k symbols of the remaining input.
- Recorded in an **action**–table.
- The entries in this table are:

shift:	read next input symbol;
reduce ($X \rightarrow \alpha$):	reduce by production $X \rightarrow \alpha$;
error:	report error
accept:	report successful termination.

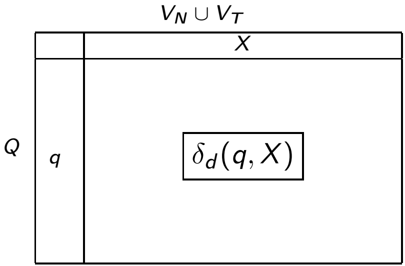
A **goto**–table records the transition function of the LR–DFA(G).

The action– and the goto–table

action-table



goto-table



Parser Table for $S \rightarrow aSb|\epsilon$

Action–table

state	sets of items	symbols		
		a	b	$\#$
0	$\left\{ \begin{array}{l} [S' \rightarrow .S], \\ [S \rightarrow .aSb], \\ [S \rightarrow .] \end{array} \right\}$	s		$r(S \rightarrow \epsilon)$
1	$\left\{ \begin{array}{l} [S \rightarrow a.Sb], \\ [S \rightarrow .aSb], \\ [S \rightarrow .] \end{array} \right\}$	s	$r(S \rightarrow \epsilon)$	
2	$\{[S \rightarrow aS.b]\}$		s	
3	$\{[S \rightarrow aSb.]\}$		$r(S \rightarrow aSb)$	$r(S \rightarrow aSb)$
4	$\{[S' \rightarrow S.]\}$			$accept$

Goto–table

state	symbol			
	a	b	$\#$	S
0	1			4
1	1			2
2		3		
3				
4				

Parsing *aabb*

Stack	Input	Action
\$ 0	$aabb\#$	shift 1
\$ 0 1	$abb\#$	shift 1
\$ 0 1 1	$bb\#$	reduce $S \rightarrow \epsilon$
\$ 0 1 1 2	$bb\#$	shift 3
\$ 0 1 1 2 3	$b\#$	reduce $S \rightarrow aSb$
\$ 0 1 2	$b\#$	shift 3
\$ 0 1 2 3	$\#$	reduce $S \rightarrow aSb$
\$ 0 4	$\#$	accept

Compressed Representation

- Integrate the terminal columns of the goto–table into the action–table.
- Combine **shift** entry for q and a with $\delta_d(q, a)$.
- Interpret action$[q, a] =$ **shift** p as read a and push p.

Compressed Parser table for $S \rightarrow aSb|\epsilon$

st.	sets of items	symbols			goto
		a	b	$\#$	S
0	$\left\{\begin{array}{l}[S' \rightarrow .S],\\ [S \rightarrow .aSb],\\ [S \rightarrow .]\}\end{array}\right\}$	$s1$		$rS \rightarrow \epsilon$	4
1	$\left\{\begin{array}{l}[S \rightarrow a.Sb],\\ [S \rightarrow .aSb],\\ [S \rightarrow .]\}\end{array}\right\}$	$s1$	$rS \rightarrow \epsilon$		2
2	$\{[S \rightarrow aS.b]\}$		$s3$		
3	$\{[S \rightarrow aSb.]\}$		$rS \rightarrow aSb$	$rS \rightarrow aSb$	
4	$\{[S' \rightarrow S.]\}$			$accept$	

Compressed Parser table for $S \rightarrow AB, S \rightarrow A, A \rightarrow a, B \rightarrow a$

s	sets of items	symbols		goto		
		a	$\#$	A	B	S
0	$\left\{ \begin{array}{l} [S' \rightarrow .S], \\ [S \rightarrow .AB], \\ [S \rightarrow .A], \\ [A \rightarrow .a] \end{array} \right\}$	$s1$		2		5
1	$\{[A \rightarrow a.]\}$	$rA \rightarrow a$	$rA \rightarrow a$			
2	$\left\{ \begin{array}{l} [S \rightarrow A.B], \\ [S \rightarrow A.], \\ [B \rightarrow .a] \end{array} \right\}$	$s3$	$rS \rightarrow A$		4	
3	$\{[B \rightarrow a.]\}$		$rB \rightarrow a$			
4	$\{[S \rightarrow AB.]\}$		$rS \rightarrow AB$			
5	$\{[S' \rightarrow S.]\}$		a			

Parsing *aa*

Stack	Input	Action
$ 0	$aa\#$	**shift** 1
$ 0 1	$a\#$	**reduce** $A \to a$
$ 0 2	$a\#$	**shift** 3
$ 0 2 3	$\#$	**reduce** $B \to a$
$ 0 2 4	$\#$	**reduce** $S \to AB$
$ 0 5	$\#$	**accept**

Algorithm LR(1)–PARSER

type *state* = **set of item;**
var *lookahead:* **symbol;**
(∗ the next not yet consumed input symbol ∗)
S : **stack of state;**
proc *scan*;
(∗ reads the next symbol into *lookahead* ∗)
proc *acc*;
(∗ report successful parse; halt ∗)
proc *err(message*: **string**);
(∗ report error; halt ∗)

```
scan; push(S, q_d);
forever do
    case  action[top(S), lookahead] of
      shift:  begin  push(S, goto[top(S), lookahead]);
                     scan
              end ;
      reduce (X→α) :    begin
                             pop^|α|(S); push(S, goto[top(S), X]);
                             output("X → α")
                        end ;
      accept:   acc;
      error:    err("...");
    end case
od
```

Construction of LR(1)–Parsers

Classes of LR–Parsers:

canonical LR(1): analyze languages of LR(1)–grammars,

SLR(1): use $FOLLOW_1$ to resolve conflicts,
size is size of LR(0)–parser,

LALR(1): refine lookahead sets compared to $FOLLOW_1$,
size is size of LR(0)–parser.
BISON is an LALR(1)–parser generator.

LR(1)–Conflicts

Set of LR(1)-items I has a

shift-reduce-conflict:

if exists at least one item $[X \longrightarrow \alpha.a\beta, L_1] \in I$
and at least one item $[Y \longrightarrow \gamma., L_2] \in I$,
and if $a \in L_2$.

reduce-reduce-conflict:

if it contains at least two items $[X \longrightarrow \alpha., L_1]$
and $[Y \longrightarrow \beta., L_2]$ where $L_1 \cap L_2 \neq \emptyset$.

A state with a conflict is called **inadequate**.

Construction of an LR(1)–Action Table

Input: set of LR(1)–states Q without inadequate states
Output: action-table
Method:

foreach $q \in Q$ **do**
  foreach $LR(1)$–*item* $[K, L] \in q$ **do**
    if $K = [S' \rightarrow S.]$ **and** $L = \{\#\}$
    then $action[q, \#] := accept$
    elseif $K = [X \rightarrow \alpha.]$
    then **foreach** $a \in L$ **do**
      $action[q, a] := reduce(X \rightarrow \alpha)$
      od
    elseif $K = [X \rightarrow \alpha.a\beta]$
    then $action[q, a] := shift$
    fi
  od
od;

foreach $q \in Q$ **and** $a \in V_T$ *such that* action$[q, a]$ *is undef.* **do**
  action$[q, a] := error$
od;

Computing Canonical LR(1)–States

Input: cfg G
Output: char. NFA of a canonical LR(1)–Parser for G.
Method: The states and transitions are constructed
using the functions *Start, Closure* and *Succ*.

var q, q' : **set of item**;
var Q : **set of set of item**;
var δ : **set of item** $\times (V_N \cup V_T) \rightarrow$ **set of item**;
function *Start:* **set of item**;
return($\{[S' \rightarrow .S, \{\#\}]\}$);

Computing Canonical LR(1)–States

function *Closure*(q : **set of item**) : **set of item**;
begin
  foreach $[X \rightarrow \alpha.Y\beta, L]$ **in** q **and** $Y \rightarrow \gamma$ **in** P **do**
    if *exist.* $[Y \rightarrow .\gamma, L']$ **in** q
    then *replace* $[Y \rightarrow .\gamma, L']$ *by* $[Y \rightarrow .\gamma, L' \cup \varepsilon\text{-}ffi(\beta L)]$
    else $q := q \cup \{[Y \rightarrow .\gamma, \varepsilon\text{-}ffi(\beta L)]\}$
    fi
  od;
  return(q)
end ;
function *Succ*(q : **set of item**, $Y : V_N \cup V_T$) : **set of item**;
  return($\{[X \rightarrow \alpha Y.\beta, L] \mid [X \rightarrow \alpha.Y\beta, L] \in q\}$);

Computing Canonical LR(1)–States

```
begin
    Q := {Closure(Start)};   δ := ∅;
    foreach q in Q and  X in V_N ∪ V_T do
        let  q' = Closure(Succ(q, X)) in
            if  q' ≠ ∅ (* X–successor exists *)
            then
               if  q' not in Q (* new state *)
               then  Q := Q ∪ {q'}
               fi;
               δ := δ ∪ {q --X--> q'} (* new transition *)
            fi
        tel
    od
end
```

Computing Canonical LR(1)–States

- The test "q' not in Q" uses an equality test on LR(1)–items. $[K_1, L_1] = [K_2, L_2]$ iff $K_1 = K_2$ and $L_1 = L_2$.
- The canonical LR(1)–parser generator splits LR(0)–states.
- LALR(1)–parsers could be generated by
  - using the equality' test $[K_1, L_1] = [K_2, L_2]$ iff $K_1 = K_2$.
  - and replacing an existing state q'' by a state, in which equal' items $[K_1, L_1] \in q'$ and $[K_2, L_2] \in q''$ are merged to new items $[K_1, L_1 \cup L_2]$.

Example from G_0

$$S'_0 = \; Closure(\text{Start}) = \{[S \rightarrow .E, \{\#\}], [E \rightarrow .E + T, \{\#, +\}], [E \rightarrow .T, \{\#, +\}], [T \rightarrow .T * F, \{\#, +, *\}], [T \rightarrow .F, \{\#, +, *\}], [F \rightarrow .(E), \{\#, +, *\}], [F \rightarrow .\textbf{id}, \{\#, +, *\}] \; \}$$

$$S'_1 = \; Closure(Succ(S'_0, E)) = \{[S \rightarrow E., \{\#\}], [E \rightarrow E. + T, \{\#, +\}] \; \}$$

$$S'_2 = \; Closure(Succ(S'_0, T)) = \{[E \rightarrow T., \{\#, +\}], [T \rightarrow T. * F, \{\#, +, *\}] \; \}$$

$$S'_6 = \; Closure(Succ(S'_1, +)) = \{[E \rightarrow E + .T, \{\#, +\}], [T \rightarrow .T * F, \{\#, +, *\}], [T \rightarrow .F, \{\#, +, *\}], [F \rightarrow .(E), \{\#, +, *\}], [F \rightarrow .\textbf{id}, \{\#, +, *\}] \; \}$$

$$S'_9 = \; Closure(Succ(S'_6, T)) = \{[E \rightarrow E + T., \{\#, +\}], [T \rightarrow T. * F, \{\#, +, *\}] \; \}$$

Inadequate LR(0)–states S_1, S_2 und S_9 are adequate after adding lookahead sets.

S'_1 shifts under "$+$", reduces under "$\#$".

S'_2 shifts under "$*$", reduces under "$\#$" and "$+$",

S'_9 shifts under "$*$", reduces under "$\#$" and "$+$".

Non–canonical LR–Parsers

SLR(1)– and LALR(1)–Parsers are constructed by

1. building an LR(0)–parser,
2. testing for inadequate LR(0)–states,
3. extending complete items by lookahead sets,
4. testing for inadequate LR(1)–states.

The lookahead set for item $[X \rightarrow \alpha.\beta]$ in q is denoted $LA(q, [X \rightarrow \alpha.\beta])$
The function $LA : Q_d \times It_G \rightarrow 2^{V_T \cup \{\#\}}$ is differently defined for SLR(1) (LA_S) und LALR(1) (LA_L).
SLR(1)– and LALR(1)–Parsers have the size of the LR(0)–parser, i.e., no states are split.

Constructing SLR(1)–Parsers

- Add $LA_S(q, [X \to \alpha.]) = FOLLOW_1(X)$ to all complete items;
- Check for inadequate SLR(1)–states.
- Cfg G is **SLR(1)** if it has no inadequate SLR(1)–states.

Example from G_0:
Extend the complete items in the inadequate states S_1, S_2 and S_9 by $FOLLOW_1$ as their lookahead sets.

$$S_1'' = \{ \quad [S \to E., \{\#\}], \quad \text{conflict removed,}$$
$$[E \to E. + T]\} \quad \text{"}+\text{" is not in } \{\#\}$$

$$S_2'' = \{ \quad [E \to T., \{\#, +,)\}], \quad \text{conflict removed,}$$
$$[T \to T. * F]\ \} \quad \text{"}*\text{" is not in } \{\#, +,)\}$$

$$S_9'' = \{ \quad [E \to E + T., \{\#, +,)\}], \quad \text{conflict removed,}$$
$$[T \to T. * F]\ \} \quad \text{"}*\text{" is not in } \{\#, +,)\}$$

G_0 is an SLR(1)–grammar.

A Non–SLR(1)–Grammar

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

Slightly abstracted form of the C–assignment.

States of the LR–DFA as sets of items

$$
\begin{array}{lll}
S_0 & = \{ & [S' \rightarrow .S], \\
 & & [S \rightarrow .L = R], \\
 & & [S \rightarrow .R], \\
 & & [L \rightarrow . * R], \\
 & & [L \rightarrow .\mathbf{id}], \\
 & & [R \rightarrow .L] \quad \} \\
S_1 & = \{ & [S' \rightarrow S.] \quad \} \\
S_2 & = \{ & [S \rightarrow L. = R], \\
 & & [R \rightarrow L.] \quad \} \\
S_3 & = \{ & [S \rightarrow R.] \quad \} \\
S_4 & = \{ & [L \rightarrow * .R], \\
 & & [R \rightarrow .L], \\
 & & [L \rightarrow . * R], \\
 & & [L \rightarrow .\mathbf{id}] \quad \}
\end{array}
$$

$$
\begin{array}{lll}
S_5 & = \{ & [L \rightarrow \mathbf{id}.] \quad \} \\
S_6 & = \{ & [S \rightarrow L = .R], \\
 & & [R \rightarrow .L], \\
 & & [L \rightarrow . * R], \\
 & & [L \rightarrow .\mathbf{id}] \quad \} \\
S_7 & = \{ & [L \rightarrow * R.] \quad \} \\
S_8 & = \{ & [R \rightarrow L.] \quad \} \\
S_9 & = \{ & [S \rightarrow L = R.] \quad \}
\end{array}
$$

S_2 is the only inadequate LR(0)–state.

Extend $[R \rightarrow L.] \in S_2$ by $FOLLOW_1(R) = \{\#, =\}$ does not remove the shift-reduce conflict, since the symbol to shift "=" is in the lookahead set

LALR(1)–Parsers

SLR(1): $LA_S(q, [X \to \alpha.]) =$
$\{a \in V_T \cup \{\#\} \mid S'\# \stackrel{*}{\Longrightarrow} \beta X a \gamma\} = FOLLOW_1(X)$

LALR(1): $LA_L(q, [X \to \alpha.]) =$
$\{a \in V_T \cup \{\#\} \mid S'\# \underset{rm}{\stackrel{*}{\Longrightarrow}} \beta X a w \text{ and } \delta_d^*(q_d, \beta\alpha) = q\}$
Lookahead set $LA_L(q, [X \to \alpha.])$ depends on the state q.

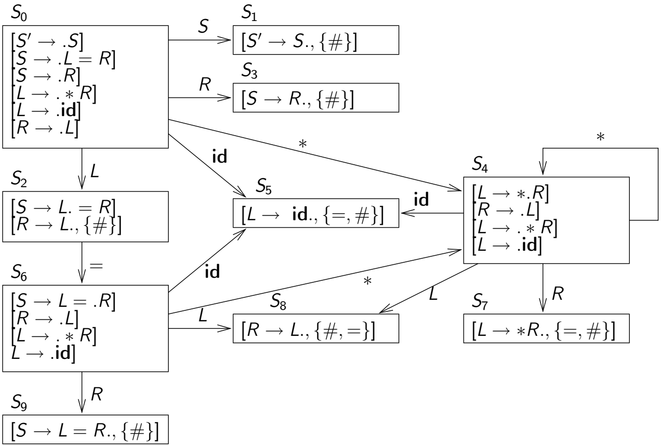
- Add $LA_L(q, [X \to \alpha.])$ to all complete items;
- Check for inadequate LALR(1)–states.
- Cfg G is **LALR(1)** if it has no inadequate LALR(1)–states.
- Definition is not constructive.
- Construction by modifying the LR(1)–Parser Generator, merging items with identical cores.

The Size of LR(1) Parsers

The number of states of canonical and non-canonical LR(1) parsers for Java and C:

	C	Java
LALR(1)	400	600
LR(1)	10000	12000

Non–SLR–Example



Grammar is LALR(1)–grammar.

Interesting Non $LR(1)$ Grammars

- Common "derived" prefix

$$
\begin{array}{rcl}
A & \rightarrow & B_1 ab \\
A & \rightarrow & B_2 ac \\
B_1 & \rightarrow & \epsilon \\
B_2 & \rightarrow & \epsilon
\end{array}
$$

- Optional non-terminals

$$
\begin{array}{rcl}
St & \rightarrow & OptLab\ St' \\
OptLab & \rightarrow & id\ : \\
OPtlab & \rightarrow & \epsilon \\
St' & \rightarrow & id := Exp
\end{array}
$$

- Ambiguous:
  - Ambiguous arithmetic expressions
  - Dangling-else

Bison Specification

Definitions: start-non-terminal+tokens+associativity
%%
Productions
%%
C-Routines

Bison Example

```
%{
int line_number = 1 ;  int error_occ = 0 ;
void yyerror(char *);
#include <stdio.h>
%}
%start exp
%left '+'
%left '*'
%right UMINUS
%token INTCONST
%%
exp:    exp '+' exp { $$ = $1 + $3 ;}
   |    exp '*' exp { $$ = $1 * $3 ;}
   |    '-' exp  %prec UMINUS { $$ = - $2 ; }
   |   '(' exp ')' { $$ = $2 ; }
   | INTCONST
   ;
%%
void yyerror(char *message)
{ fprintf(stderr, "%s near line %ld. \n", message, line_number);
  error_occ=1; }
```

Flex for the Example

```
%{
#include <math.h>
#include "calc.tab.h"
extern int line_number;
%}
Digit [0-9]
%%
{Digit}+                    {yylval = atoi(yytext) ;
                             return(INTCONST); }
\n     {line_number++ ; }
[\t ]+                      ;
.                           {return(*yytext); }
%%
```