Transforming a grammar for $LL(1)$ parsing

- Ambiguous grammars are not $LL(1)$ but unambiguous grammars are not necessarily $LL(1)$
- Having a non-$LL(1)$ unambiguous grammar for a language does not mean that this language is not $LL(1)$.
- But there are languages for which there exist unambiguous context-free grammars but no $LL(1)$ grammar.

- We will see two grammar transformations that improve the chance to get a $LL(1)$ grammar:
  - Elimination of left-recursion
  - Left-factorization

Left-recursion

- The following expression grammar is unambiguous but it is not $LL(1)$:

$$
\begin{array}{rcl}
Exp & \rightarrow & Exp + Exp2 \\
Exp & \rightarrow & Exp - Exp2 \\
Exp & \rightarrow & Exp2 \\
Exp2 & \rightarrow & Exp2 * Exp3 \\
Exp2 & \rightarrow & Exp2 / Exp3 \\
Exp2 & \rightarrow & Exp3 \\
Exp3 & \rightarrow & \textbf{num} \\
Exp3 & \rightarrow & (Exp)
\end{array}
$$

- Indeed, $First(\alpha)$ is the same for all RHS α of the productions for *Exp* et *Exp*2
- This is a consequence of *left-recursion*.

Left-recursion

- Recursive productions are productions defined in terms of themselves. Examples: $A \to Ab$ ou $A \to bA$.
- When the recursive nonterminal is at the left (resp. right), the production is said to be left-recursive (resp. right-recursive).
- Left-recursive productions can be rewritten with right-recursive productions
- Example:

$$\begin{array}{rcl} N & \to & N\alpha_1 \\ & \vdots & \\ N & \to & N\alpha_m \\ N & \to & \beta_1 \\ & \vdots & \\ N & \to & \beta_n \end{array} \quad \Leftrightarrow \quad \begin{array}{rcl} N & \to & \beta_1 N' \\ & \vdots & \\ N & \to & \beta_n N' \\ N' & \to & \alpha_1 N' \\ & \vdots & \\ N' & \to & \alpha_m N' \\ N' & \to & \epsilon \end{array}$$

Right-recursive expression grammar

$$\begin{array}{rcl} Exp & \rightarrow & Exp + Exp2 \\ Exp & \rightarrow & Exp - Exp2 \\ Exp & \rightarrow & Exp2 \\ Exp2 & \rightarrow & Exp2 * Exp3 \\ Exp2 & \rightarrow & Exp2 / Exp3 \\ Exp2 & \rightarrow & Exp3 \\ Exp3 & \rightarrow & \textbf{num} \\ Exp3 & \rightarrow & (Exp) \end{array} \quad \Leftrightarrow \quad \begin{array}{rcl} Exp & \rightarrow & Exp2\,Exp' \\ Exp' & \rightarrow & +Exp2\,Exp' \\ Exp' & \rightarrow & -Exp2\,Exp' \\ Exp' & \rightarrow & \epsilon \\ Exp2 & \rightarrow & Exp3\,Exp2' \\ Exp2' & \rightarrow & *Exp3\,Exp2' \\ Exp2' & \rightarrow & /Exp3\,Exp2' \\ Exp2' & \rightarrow & \epsilon \\ Exp3 & \rightarrow & \textbf{num} \\ Exp3 & \rightarrow & (Exp) \end{array}$$

Left-factorisation

- The RHS of these two productions have the same *First* set.

$$\begin{array}{rcl} Stat & \rightarrow & \textbf{if } Exp \textbf{ then } Stat \textbf{ else } Stat \\ Stat & \rightarrow & \textbf{if } Exp \textbf{ then } Stat \end{array}$$

- The problem can be solved by left factorising the grammar:

$$\begin{array}{rcl} Stat & \rightarrow & \textbf{if } Exp \textbf{ then } Stat\ ElseStat \\ ElseStat & \rightarrow & \textbf{else } Stat \\ ElseStat & \rightarrow & \epsilon \end{array}$$

- Note
  - The resulting grammar is ambiguous and the parsing table will contain two rules for $M[ElseStat, \textbf{else}]$ (because $\textbf{else} \in Follow(ElseStat)$ and $\textbf{else} \in First(\textbf{else } Stat)$)
  - Ambiguity can be solved in this case by letting $M[ElseStat, \textbf{else}] = \{ElseStat \rightarrow \textbf{else } Stat\}$.

Hidden left-factors and hidden left recursion

- Sometimes, left-factors or left recursion are hidden
- Examples:
  - The following grammar:

$$\begin{array}{rcl} A & \rightarrow & da|acB \\ B & \rightarrow & abB|daA|Af \end{array}$$

    has two overlapping productions: $B \rightarrow daA$ and $B \stackrel{*}{\Rightarrow} daf$.
  - The following grammar:

$$\begin{array}{rcl} S & \rightarrow & Tu|wx \\ T & \rightarrow & Sq|vvS \end{array}$$

    has left recursion on T ($T \stackrel{*}{\Rightarrow} Tuq$)
- Solution: expand the production rules by substitution to make left-recursion or left factors visible and then eliminate them

Summary

Construction of a $LL(1)$ parser from a CFG grammar

- Eliminate ambiguity
- Eliminate left recursion
- left factorization
- Add an extra start production $S' \rightarrow S\$$ to the grammar
- Calculate *First* for every production and *Follow* for every nonterminal
- Calculate the parsing table
- Check that the grammar is $LL(1)$

Recursive implementation

- From the parsing table, it is easy to implement a predictive parser recursively (with one function per nonterminal)

$$
\begin{array}{lcl}
T' & \rightarrow & T\$ \\
T & \rightarrow & R \\
T & \rightarrow & aTc \\
R & \rightarrow & \epsilon \\
R & \rightarrow & bR
\end{array}
$$

	a	b	c	$\$$
T'	$T' \rightarrow T\$$	$T' \rightarrow T\$$		$T' \rightarrow T\$$
T	$T \rightarrow aTc$	$T \rightarrow R$	$T \rightarrow R$	$T \rightarrow R$
R		$R \rightarrow bR$	$R \rightarrow \epsilon$	$R \rightarrow \epsilon$

```
function parseT'() =
  if next = 'a' or next = 'b' or next = '$' then
    parseT() ; match('$')
  else reportError()

function parseT() =
  if next = 'b' or next = 'c' or next = '$' then
    parseR()
  else if next = 'a' then
    match('a') ; parseT() ; match('c')
  else reportError()

function parseR() =
  if next = 'c' or next = '$' then
    (* do nothing *)
  else if next = 'b' then
    match('b') ; parseR()
  else reportError()
```

(Mogensen)

Outline

Bottom-up parsing

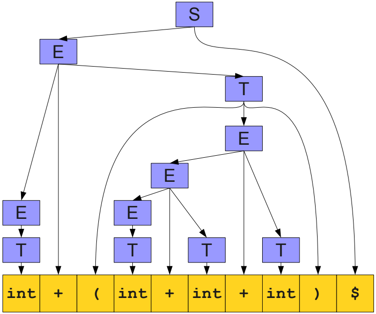
- A bottom-up parser creates the parse tree starting from the leaves towards the root
- It tries to convert the program into the start symbol
- Most common form of bottom-up parsing: shift-reduce parsing

Bottom-up parsing: example

Grammar:

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow T \\
E &\rightarrow E + T \\
T &\rightarrow \textbf{int} \\
T &\rightarrow (E)
\end{aligned}
$$

Bottum-up parsing of
$int + (int + int + int)$



(Keith Schwarz)

Bottom-up parsing: example

Grammar:

$$
\begin{array}{lcl}
S & \rightarrow & E \\
E & \rightarrow & T \\
E & \rightarrow & E + T \\
T & \rightarrow & \textbf{int} \\
T & \rightarrow & (E)
\end{array}
$$

Bottum-up parsing of
$int + (int + int + int)$:

$int + (int + int + int)\$$
$T + (int + int + int)\$$
$E + (int + int + int)\$$
$E + (T + int + int)\$$
$E + (E + int + int)\$$
$E + (E + T + int)\$$
$E + (E + int)\$$
$E + (E + T)\$$
$E + (E)\$$
$E + T\$$
$E\$$
S

Top-down parsing is often done as a rightmost derivation in reverse
(There is only one if the grammar is unambiguous).

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 $\alpha \overset{*}{\Rightarrow}_{rm} \beta$.
- A reduction transforms uwv to uAv if $A \rightarrow w$ is a production
- α is a right sentential form if $S \overset{*}{\Rightarrow}_{rm} \alpha$ avec $\alpha = \beta x$ 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:

$$S \overset{*}{\Rightarrow}_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$$

  - 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.

Handle: example

Grammar:

$$
\begin{array}{lcl}
S & \rightarrow & E \\
E & \rightarrow & T \\
E & \rightarrow & E + T \\
T & \rightarrow & \textbf{int} \\
T & \rightarrow & (E)
\end{array}
$$

Bottum-up parsing of
$int + (int + int + int)$

$\textcolor{red}{int} + (int + int + int)\$$
$\textcolor{red}{T} + (int + int + int)\$$
$E + (\textcolor{red}{int} + int + int)\$$
$E + (\textcolor{red}{T} + int + int)\$$
$E + (E + \textcolor{red}{int} + int)\$$
$E + (\textcolor{red}{E + T} + int)\$$
$E + (E + \textcolor{red}{int})\$$
$E + (\textcolor{red}{E + T})\$$
$E + \textcolor{red}{(E)}\$$
$\textcolor{red}{E + T}\$$
$\textcolor{red}{E}\$$
S

The handle is in red in each right sentential form

Finding the handles

- Bottom-up parsing = finding the handle in the right sentential form obtained at each step
- This handle is unique as soon as the grammar is unambiguous (because in this case, the rightmost derivation is unique)
- Suppose that our current form is uvw and the handle is $A \rightarrow v$ (getting uAw after reduction). w can not contain any nonterminals (otherwise we would have reduced a handle somewhere in w)

Shift/reduce parsing

Proposed model for a bottom-up parser:

- Split the input into two parts:
  - Left substring is our work area
  - Right substring is the input we have not yet processed
- All handles are reduced in the left substring
- Right substring consists only of terminals
- At each point, decide whether to:
  - Move a terminal across the split (shift)
  - Reduce a handle (reduce)

Shift/reduce parsing: example

Grammar:

$$
\begin{aligned}
E &\rightarrow E + T | T \\
T &\rightarrow T * F | F \\
F &\rightarrow (E) | \textbf{id}
\end{aligned}
$$

Bottum-up parsing of
$id + id * id$

Left substring	Right substring	Action
$\$$	$id + id * id\$$	Shift
$\$id$	$+id * id\$$	Reduce by $F \rightarrow \textbf{id}$
$\$F$	$+id * id\$$	Reduce by $T \rightarrow F$
$\$T$	$+id * id\$$	Reduce by $E \rightarrow T$
$\$E$	$+id * id\$$	Shift
$\$E+$	$id * id\$$	Shift
$\$E + id$	$*id\$$	Reduce by $F \rightarrow \textbf{id}$
$\$E + F$	$*id\$$	Reduce by $T \rightarrow F$
$\$E + T$	$*id\$$	Shift
$\$E + T*$	$id\$$	Shift
$\$E + T * id$	$\$$	Reduce by $F \rightarrow \textbf{id}$
$\$E + T * F$	$\$$	Reduce by $T \rightarrow T * F$
$\$E + T$	$\$$	Reduce by $E \rightarrow E + T$
$\$E$	$\$$	Accept

Shift/reduce parsing

- In the previous example, all the handles were to the far right end of the left area (not inside)
- This is convenient because we then never need to shift from the left to the right and thus could process the input from left-to-right in one pass.

- Is it the case for all grammars? Yes !
- Sketch of proof: by induction on the number of reduces
  - After no reduce, the first reduction can be done at the right end of the left area
  - After at least one reduce, the very right of the left area is a nonterminal (by induction hypothesis). This nonterminal must be part of the next reduction, since we are tracing a rightmost derivation backwards.

Shift/reduce parsing

- Consequence: the left area can be represented by a stack (as all activities happen at its far right)
- Four possible actions of a shift-reduce parser:
  1. Shift: push the next terminal onto the stack
  2. Reduce: Replace the handle on the stack by the nonterminal
  3. Accept: parsing is successfully completed
  4. Error: discover a syntax error and call an error recovery routine

Shift/reduce parsing

- There still remain two open questions: At each step:
  - How to choose between shift and reduce?
  - If the decision is to reduce, which rules to choose (i.e., what is the handle)?
- Ideally, we would like this choice to be deterministic given the stack and the next k input symbols (to avoid backtracking), with k typically small (to make parsing efficient)
- Like for top-down parsing, this is not possible for all grammars
- Possible conflicts:
  - shift/reduce conflict: it is not possible to decide between shifting or reducing
  - reduce/reduce conflict: the parser can not decide which of several reductions to make

Shift/reduce parsing

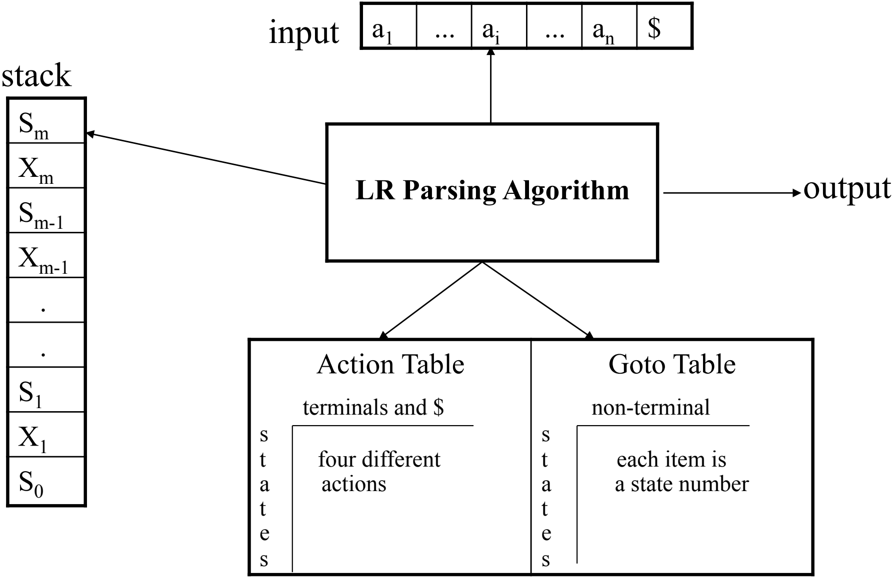
We will see two main categories of shift-reduce parsers:

- LR-parsers
  - They cover a wide range of grammars
  - Different variants from the most specific to the most general: SLR, LALR, LR
- Weak precedence parsers
  - They work only for a small class of grammars
  - They are less efficient than LR-parsers
  - They are simpler to implement

LR-parsers

- LR(k) parsing: Left-to-right, Rightmost derivation, *k* symbols lookahead.
- Advantages:
  - The most general non-backtracking shift-reduce parsing, yet as efficient as other less general techniques
  - Can detect syntactic error as soon as possible (on a left-to-right scan of the input)
  - Can recognize virtually all programming language constructs (that can be represented by context-free grammars)
  - Grammars recognized by LR parsers is a proper subset of grammars recognized by predictive parsers (LL(k) $\subset$ LR(k))
- Drawbacks:
  - More complex to implement than predictive (or operator precedence) parsers
- Like table-driven predictive parsing, LR parsing is based on a parsing table.

Structure of a LR parser



Structure of a LR parser

- A configuration of a LR parser is described by the status of its stack and the part of the input not analysed (shifted) yet:

$$(s_0 X_1 s_1 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$$

  where X_i are (terminal or nonterminal) symbols, a_i are terminal symbols, and s_i are state numbers (of a DFA)
- A configuration corresponds to the right sentential form

$$X_1 \dots X_m a_i \dots a_n$$

- Analysis is based on two tables:
  - an action table that associates an action $ACTION[s, a]$ to each state s and nonterminal a.
  - a goto table that gives the next state $GOTO[s, A]$ from state s after a reduction to a nonterminal A

Actions of a LR-parser

- Let us assume the parser is in configuration

$$(s_0 X_1 s_1 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$$

(initially, the state is $(s_0, a_1 a_2 \ldots a_n \$)$*, where* $a_1 \ldots a_n$ *is the input word)*

- $\mathrm{ACTION}[s_m, a_i]$ can take four values:
  1. Shift s: shifts the next input symbol and then the state s on the stack $(s_0 X_1 s_1 \ldots X_m s_m, a_i a_{i+1} \ldots a_n) \rightarrow (s_0 X_1 s_1 \ldots X_m a_i s, a_{i+1} \ldots a_n)$
  2. Reduce $A \rightarrow \beta$ (denoted by rn where n is a production number)
     - Pop $2|\beta|$ $(= r)$ items from the stack
     - Push A and s where $s = GOTO[s_{m-r}, A]$
       $(s_0 X_1 s_1 \ldots X_m s_m, a_i a_{i+1} \ldots a_n) \rightarrow$
       $(s_0 X_1 s_1 \ldots X_{m-r} s_{m-r} A s, a_i a_{i+1} \ldots a_n)$
     - Output the prediction $A \rightarrow \beta$
  3. Accept: parsing is successfully completed
  4. Error: parser detected an error (typically an empty entry in the action table).

LR-parsing algorithm

```
Create a stack with the start state s_0
a = GETNEXTTOKEN()
while (True)
    s = POP()
    if (ACTION[s, a] = shift t)
        Push a and t onto the stack
        a = GETNEXTTOKEN()
    elseif (ACTION[s, a] = reduce A → β)
        Pop 2|β| elements off the stack
        Let state t now be the state on the top of the stack
        Push GOTO[t, A] onto the stack
        Output A → β
    elseif (ACTION[s, a] = accept)
        break // Parsing is over
    else call error-recovery routine
```

Example: parsing table for the expression grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow$ **id**

	Action Table						Goto Table		
state	**id**	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Example: LR parsing with the expression grammar

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Constructing the parsing tables

- There are several ways of building the parsing tables, among which:
  - LR(0): no lookahead, works for only very few grammars
  - SLR: the simplest one with one symbol lookahead. Works with less grammars than the next ones
  - LR(1): very powerful but generate potentially very large tables
  - LALR(1): tradeoff between the other approaches in terms of power and simplicity
  - LR(k), $k > 1$: exploit more lookahead symbols
- LALR(1) is used in parser generators like Yacc
- We will only see SLR in this course

- Main idea of all methods: build a DFA whose states keep track of where we are in a parse

LR(0) item

- An LR(0) item (or item for short) of a grammar G is a production of G with a dot at some position of the body.
- Example: $A \rightarrow XYZ$ yields four items:

$$\begin{array}{c} A \rightarrow .XYZ \\ A \rightarrow X.YZ \\ A \rightarrow XY.Z \\ A \rightarrow XYZ. \end{array}$$

  ($A \rightarrow \epsilon$ generates one item $A \rightarrow .$)
- An item indicates how much of a production we have seen at a given point in the parsing process.
  - $A \rightarrow X.YZ$ means we have just seen on the input a string derivable from X (and we hope to get next YZ).
- Each state of the SLR parser will correspond to a set of LR(0) items
- A particular collection of sets of LR(0) items (the canonical LR(0) collection) is the basis for constructing SLR parsers

Construction of the canonical LR(0) collection

- The grammar G is first augmented into a grammar G' with a new start symbol S' and a production $S' \rightarrow S$ where S is the start symbol of G
- We need to define two functions:
  - $\text{Closure}(I)$: extends the set of items I when some of them have a dot to the left of a nonterminal
  - $\text{Goto}(I, X)$: moves the dot past the symbol X in all items in I
- These two functions will help define a DFA:
  - whose states are (closed) sets of items
  - whose transitions (on terminal and nonterminal symbols) are defined by the Goto function

CLOSURE

CLOSURE(I)

repeat

  for any item $A \rightarrow \alpha.X\beta$ in I

    for any production $X \rightarrow \gamma$

      $I = I \cup \{X \rightarrow .\gamma\}$

until I does not change

return I

Example:

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

$$
\text{CLOSURE}(\{E' \rightarrow .E\}) = \{E' \rightarrow .E, \quad E \rightarrow .E + T, \quad E \rightarrow .T, \quad T \rightarrow .T * F, \quad T \rightarrow .F, \quad F \rightarrow .(E), \quad F \rightarrow .\ \textbf{id}
$$

Goto

$$\textsc{Goto}(I, X)$$

Set J to the empty set
for any item $A \rightarrow \alpha.X\beta$ in I
$\quad J = J \bigcup \{A \rightarrow \alpha X.\beta\}$
return $\textsc{closure}(J)$

Example:

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

$$\begin{array}{lcl} I_0 & = & \{E' \rightarrow .E, \\ & & E \rightarrow .E + T \\ & & E \rightarrow .T \\ & & T \rightarrow .T * F \\ & & T \rightarrow .F \\ & & F \rightarrow .(E) \\ & & F \rightarrow .\,\textbf{id} \end{array}$$

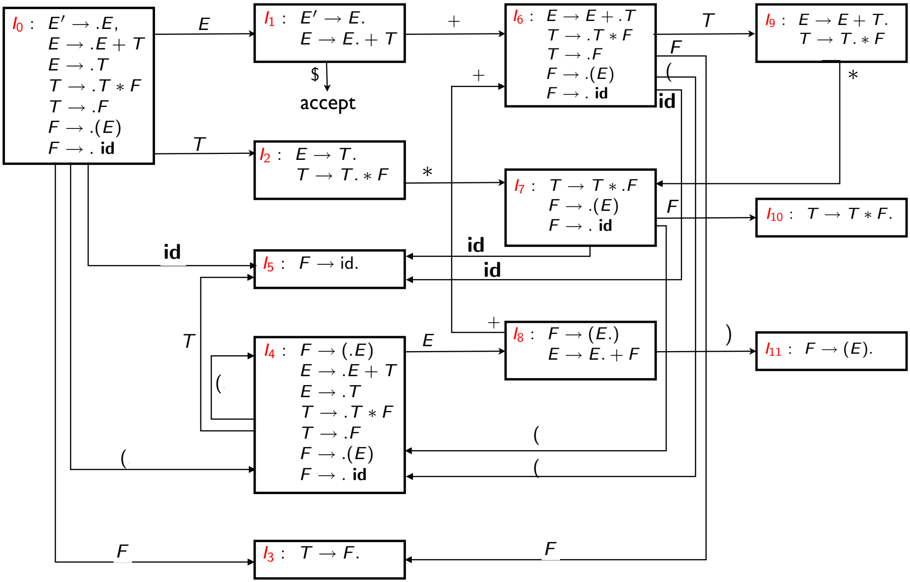
$$\begin{array}{l} \textsc{goto}(I_0, E) = \{E' \rightarrow E., E \rightarrow E. + T\} \\ \textsc{goto}(I_0, T) = \{E \rightarrow T., T \rightarrow T. * F\} \\ \textsc{goto}(I_0, F) = \{T \rightarrow F.\} \\ \textsc{goto}(I_0, '(') = \textsc{Closure}(\{F \rightarrow (.E)\}) \\ \qquad = \{F \rightarrow (.E)\} \cup (I_0 \setminus \{E' \rightarrow E\}) \\ \textsc{goto}(I_0, \text{id}) = \{F \rightarrow \text{id}.\} \end{array}$$

Construction of the canonical collection

$C = \{\text{CLOSURE}(\{S' \to .S\})$
for each item set I in C
  for each item $A \to \alpha.X\beta$ in I
    $C = C \cup \text{GOTO}(I, X)$
return C

- Collect all sets of items reachable from the initial state by one or several applications of GOTO.
- Item sets in C are the state of a DFA, GOTO is its transition function

Example



I0 : E′ → .E,
E → .E + T
E → .T
T → .T ∗ F
T → .F
F → .(E)
F → . id
I1 : E′ → E.
E → E. + T
$
accept
I2 : E → T.
T → T. ∗ F
I3 : T → F.
I4 : F → (.E)
E → .E + T
E → .T
T → .T ∗ F
T → .F
F → .(E)
F → . id
I5 : F → id.
I6 : E → E + .T
T → .T ∗ F
T → .F
F → .(E)
F → . id
I7 : T → T ∗ .F
F → .(E)
F → . id
I8 : F → (E.)
E → E. + F
I9 : E → E + T.
T → T. ∗ F
I10 : T → T ∗ F.
I11 : F → (E).
E
T
id
(
F
+
∗
)

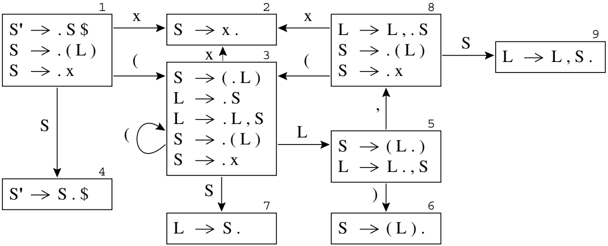
Constructing the LR(0) parsing table

1. Construct $c = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of $LR(0)$ items for G' (the augmented grammar)
2. State i of the parser is derived from I_i. Actions for state i are as follows:
   2.1 If $A \rightarrow \alpha.a\beta$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then $ACTION[i, a] = \text{Shift } j$
   2.2 If $A \rightarrow \alpha.$ is in I_i, then set $ACTION[i, a] = \text{Reduce } A \rightarrow \alpha$ for all terminals a.
   2.3 If $S' \rightarrow S.$ is in I_i, then set $ACTION[i, \$] = \text{Accept}$
3. If $\text{GOTO}(I_i, X) = I_j$, then $GOTO[i, X] = j$.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state s_0 is the set of items containing $S' \rightarrow .S$

$\Rightarrow$ *LR(0) because the chosen action (shift or reduce) only depends on the current state*

Example of a LR(0) grammar

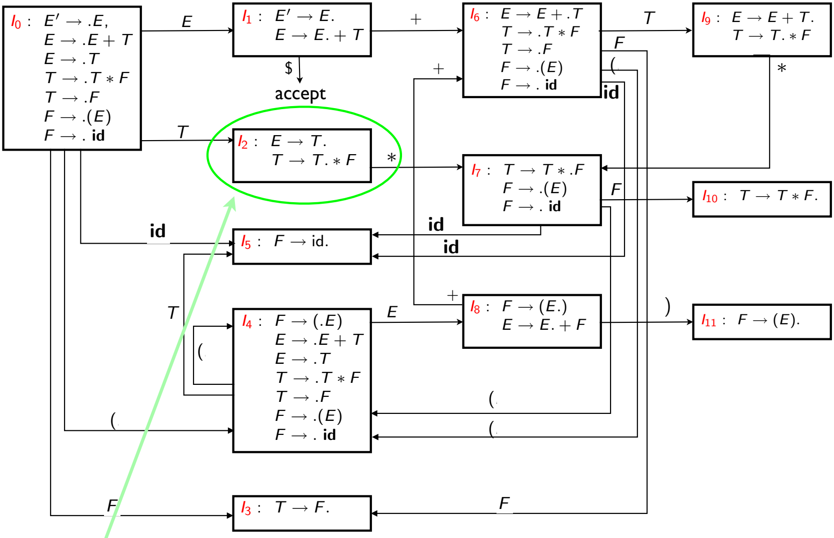
0 $S' \rightarrow S\$$
1 $S \rightarrow (\ L\ )$
2 $S \rightarrow x$
3 $L \rightarrow S$
4 $L \rightarrow L\ ,\ S$



	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

(Appel)

Example of a non LR(0) grammar



Conflict: in state 2, we don't know whether to shift or reduce.

Constructing the SLR parsing tables

1. Construct $c = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of $LR(0)$ items for G' (the augmented grammar)
2. State i of the parser is derived from I_i. Actions for state i are as follows:
   2.1 If $A \rightarrow \alpha.a\beta$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then $ACTION[i, a] =$ Shift j
   2.2 If $A \rightarrow \alpha.$ is in I_i, then $ACTION[i, a] =$ Reduce $A \rightarrow \alpha$ for all terminals a in $Follow(A)$ where $A \neq S'$
   2.3 If $S' \rightarrow S.$ is in I_i, then set $ACTION[i, \$] =$ Accept
3. If $\text{GOTO}(I_i, A) = I_j$ for a nonterminal A, then $GOTO[i, A] = j$
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state s_0 is the set of items containing $S' \rightarrow .S$

$\Rightarrow$ *the simplest form of one symbol lookahead, SLR (Simple LR)*

Example

	Action Table							Goto Table		
state	**id**	**+**	*****	**(**	**)**	**$**		**E**	**T**	**F**
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

	First	*Follow*
E	**id** (	$ +)
T	**id** (	$ + *)
F	**id** (	$ + *)

SLR(1) grammars

- A grammar for which there is no (shift/reduce or reduce/reduce) conflict during the construction of the SLR table is called SLR(1) (or SLR in short).
- All SLR grammars are unambiguous but many unambiguous grammars are not SLR
- There are more SLR grammars than LL(1) grammars but there are LL(1) grammars that are not SLR.

Conflict example for SLR parsing

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

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

I_3: $S \rightarrow R \cdot$

I_4: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id} \cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow *R \cdot$

I_8: $R \rightarrow L \cdot$

I_9: $S \rightarrow L = R \cdot$

(Dragonbook)

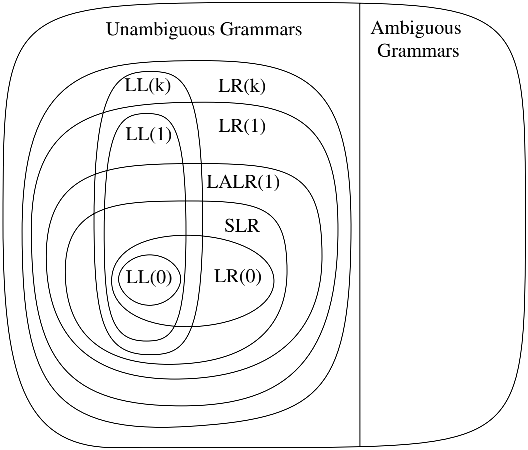
$Follow(R)$ contains '$=$'. In I_2, when seeing '$=$' on the input, we don't know whether to shift or to reduce with $R \rightarrow L$.

Summary of SLR parsing

Construction of a SLR parser from a CFG grammar

- Eliminate ambiguity *(or not, see later)*
- Add the production $S' \rightarrow S$, where S is the start symbol of the grammar
- Compute the LR(0) canonical collection of LR(0) item sets and the GOTO function (transition function)
- Add a shift action in the action table for transitions on terminals and goto actions in the goto table for transitions on nonterminals
- Compute *Follow* for each nonterminals (which implies first adding $S'' \rightarrow S'\$$ to the grammar and computing *First* and *Nullable*)
- Add the reduce actions in the action table according to *Follow*
- Check that the grammar is SLR *(and if not, try to resolve conflicts, see later)*

Hierarchy of grammar classes



(Appel)

Next week

End of syntax analysis

- Operator precedence parsing
- Error detection and recovery
- Building the parse tree