# 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 $LL(1)$:

$$
\begin{aligned}
Exp &\rightarrow Exp + Exp2 \\
Exp &\rightarrow Exp - Exp2 \\
Exp &\rightarrow Exp2 \\
Exp2 &\rightarrow Exp2 * Exp3 \\
Exp2 &\rightarrow Exp2 / Exp3 \\
Exp2 &\rightarrow Exp3 \\
Exp3 &\rightarrow \mathbf{num} \\
Exp3 &\rightarrow (Exp)
\end{aligned}
$$

- Indeed, $First(\alpha)$ is the same for all RHS $\alpha$ of the productions for $Exp$ et $Exp2$
- This is a consequence of *left-recursion*.

# Left-recursion

- Recursive productions are productions defined in terms of themselves. Examples: $A \rightarrow Ab$ ou $A \rightarrow bA$.
- 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:

$$
\begin{aligned}
N &\rightarrow N\alpha_1 \\
&\vdots \\
N &\rightarrow N\alpha_m \\
N &\rightarrow \beta_1 \\
&\vdots \\
N &\rightarrow \beta_n
\end{aligned}
\qquad \Leftrightarrow \qquad
\begin{aligned}
N &\rightarrow \beta_1 N' \\
&\vdots \\
N &\rightarrow \beta_n N' \\
N' &\rightarrow \alpha_1 N' \\
&\vdots \\
N' &\rightarrow \alpha_m N' \\
N' &\rightarrow \epsilon
\end{aligned}
$$

# Right-recursive expression grammar

$$
\begin{aligned}
Exp &\rightarrow Exp + Exp2 \\
Exp &\rightarrow Exp - Exp2 \\
Exp &\rightarrow Exp2 \\
Exp2 &\rightarrow Exp2 * Exp3 \\
Exp2 &\rightarrow Exp2 / Exp3 \\
Exp2 &\rightarrow Exp3 \\
Exp3 &\rightarrow \mathbf{num} \\
Exp3 &\rightarrow (Exp)
\end{aligned}
\qquad \Leftrightarrow \qquad
\begin{aligned}
Exp &\rightarrow Exp2\,Exp' \\
Exp' &\rightarrow + Exp2\,Exp' \\
Exp' &\rightarrow - Exp2\,Exp' \\
Exp' &\rightarrow \epsilon \\
Exp2 &\rightarrow Exp3\,Exp2' \\
Exp2' &\rightarrow * Exp3\,Exp2' \\
Exp2' &\rightarrow / Exp3\,Exp2' \\
Exp2' &\rightarrow \epsilon \\
Exp3 &\rightarrow \mathbf{num} \\
Exp3 &\rightarrow (Exp)
\end{aligned}
$$

# Left-factorisation

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

$$Stat \rightarrow \textbf{if } Exp \textbf{ then } Stat \textbf{ else } Stat$$
$$Stat \rightarrow \textbf{if } Exp \textbf{ then } Stat$$

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

$$Stat \rightarrow \textbf{if } Exp \textbf{ then } Stat \text{ } ElseStat$$
$$ElseStat \rightarrow \textbf{else } Stat$$
$$ElseStat \rightarrow \epsilon$$

- 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:

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

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

$$
\begin{aligned}
S & \rightarrow & Tu\,|\,wx \\
T & \rightarrow & Sq\,|\,vvS
\end{aligned}
$$

    has left recursion on $T$ ($T \overset{*}{\Rightarrow} Tuq$)

- 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)

$$T' \rightarrow T\$$$
$$T \rightarrow R$$
$$T \rightarrow aTc$$
$$R \rightarrow \epsilon$$
$$R \rightarrow bR$$

|     | a              | b              | c              | $              |
|-----|----------------|----------------|----------------|----------------|
| $T'$ | $T' \to T\$$  | $T' \to T\$$  |                | $T' \to T\$$  |
| $T$  | $T \to aTc$   | $T \to R$     | $T \to R$     | $T \to R$     |
| $R$  |                | $R \to bR$    | $R \to \epsilon$ | $R \to \epsilon$ |

```
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()
```
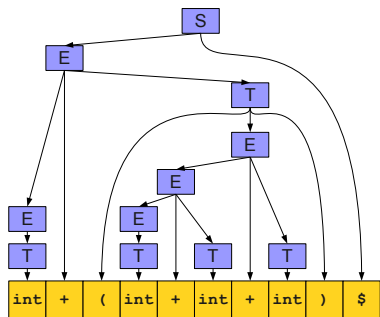
(Mogensen)

# 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

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

Grammar:

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

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

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 $\alpha$ to $\beta$ is noted $\alpha \overset{*}{\Rightarrow}_{rm} \beta$.
- A reduction transforms *uwv* to *uAv* if $A \rightarrow w$ is a production
- $\alpha$ is a right sentential form if $S \overset{*}{\Rightarrow}_{rm} \alpha$.
- A handle of a right sentential form $\gamma$ ($= \alpha\beta w$) is a production $A \rightarrow \beta$ and a position in $\gamma$ where $\beta$ may be found and replaced by $A$ to produce the previous right-sentential form in a rightmost derivation of $\gamma$:

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

  ▸ Informally, a handle is a production we can reverse without getting stuck.
  ▸ If the handle is $A \rightarrow \beta$, we will also call $\beta$ the handle.

# Handle: example

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

Grammar:

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

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

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 $A \to v$ (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:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid \textbf{id}$$

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

| Left substring | Right substring | Action |
|---|---|---|
| \$ | $id + id * id$\$ | Shift |
| \$$id$ | $+id * id$\$ | Reduce by $F \rightarrow$ **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 $F \rightarrow$ **id** |
| \$$E + F$ | $*id$\$ | Reduce by $T \rightarrow F$ |
| \$$E + T$ | $*id$\$ | Shift |
| \$$E + T*$ | $id$\$ | Shift |
| \$$E + T * id$ | \$ | Reduce by $F \rightarrow$ **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
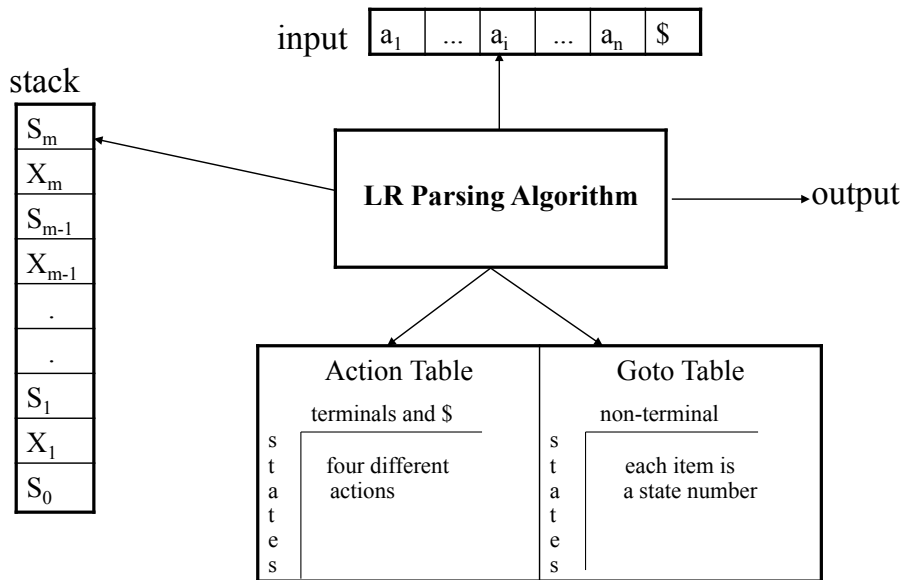  - They are less efficient than LR-parsers
  - They are simpler to implement

# Outline

# 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 superset 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:

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

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

$$X_1 \ldots X_m a_i \ldots a_n$$

- 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 $(s_0 X_1 s_1 \ldots X_m s_m, a_i a_{i+1} \ldots a_n) \rightarrow (s_0 X_1 s_1 \ldots X_m s_m a_i s, a_{i+1} \ldots a_n)$
  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 $s_0$
$a = \text{GETNEXTTOKEN}()$
**while** (True)
    $s = \text{POP}()$
    **if** ($ACTION[s, a] = \text{shift } t$)
        Push $a$ and $t$ onto the stack
        $a = \text{GETNEXTTOKEN}()$
    **elseif** ($ACTION[s, a] = \text{reduce } A \rightarrow \beta$)
        Pop $2|\beta|$ elements off the stack
        Let state $t$ now be the state on the top of the stack
        Push $A$ onto the stack
        Push $GOTO[t, A]$ onto the stack
        Output $A \rightarrow \beta$
    **elseif** ($ACTION[s, a] = \text{accept}$)
        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. $F \rightarrow \textbf{id}$

|       | 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

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

# Parser generators

- LALR(1) is used in most parser generators like Yacc/Bison

- We will nevertheless only see SLR in details:
  - It's simpler.
  - LALR(1) is only minorly more expressive.
  - When a grammar is SLR, then the tables produced by SLR are identical to the ones produced by LALR(1).
  - Understanding of SLR principles is sufficient to understand how to handle a grammar rejected by LALR(1) parser generators (see later).

# 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:

$$A \rightarrow .XYZ$$
$$A \rightarrow X.YZ$$
$$A \rightarrow XY.Z$$
$$A \rightarrow XYZ.$$

($A \rightarrow \epsilon$ 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:
  - CLOSURE($I$): extends the set of items $I$ when some of them have a dot to the left of a nonterminal
  - 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:

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

$$
\begin{aligned}
\text{CLOSURE}(\{E' \rightarrow .E\}) = \{ &E' \rightarrow .E, \\
&E \rightarrow .E + T \\
&E \rightarrow .T \\
&T \rightarrow .T * F \\
&T \rightarrow .F \\
&F \rightarrow .(E) \\
&F \rightarrow .\,\mathbf{id} \}
\end{aligned}
$$

# Goto

$$\text{GOTO}(I, X)$$
    Set $J$ to the empty set
    **for** any item $A \to \alpha.X\beta$ in $I$
        $J = J \bigcup \{A \to \alpha X.\beta\}$
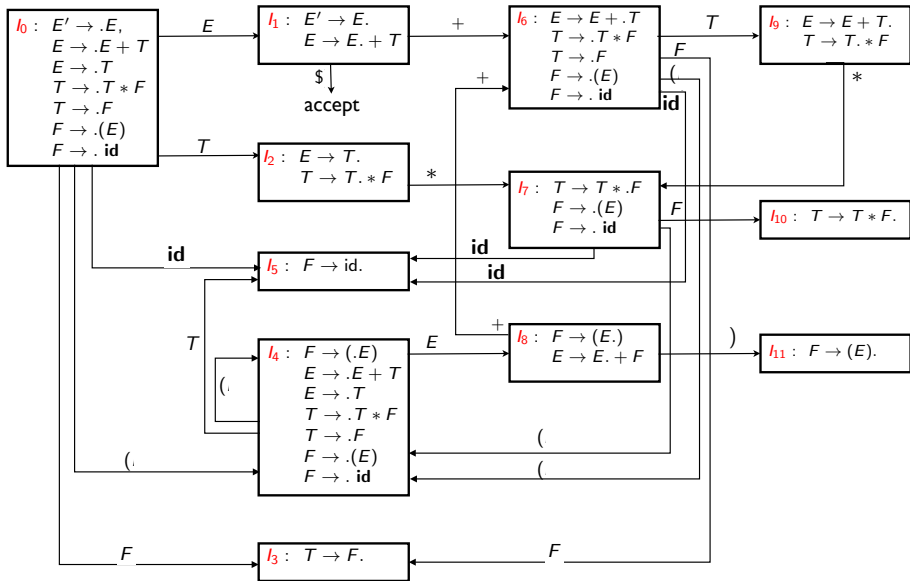    **return** CLOSURE($J$)

Example:

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

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

$\text{GOTO}(I_0, E) = \{E' \to E., E \to E. + T\}$
$\text{GOTO}(I_0, T) = \{E \to T., T \to T. * F\}$
$\text{GOTO}(I_0, F) = \{T \to F.\}$
$\text{GOTO}(I_0,' (') = \text{CLOSURE}(\{F \to (.E)\})$
$\qquad = \{F \to (.E)\} \cup (I_0 \setminus \{E' \to E\})$
$\text{GOTO}(I_0, \text{id}) = \{F \to \text{id}.\}$

# Construction of the canonical collection

$$C = \{\text{CLOSURE}(\{S' \to .S\})\}$$
**repeat**
    **for** each item set $I$ in $C$
        **for** each item $A \to \alpha.X\beta$ in $I$
            $C = C \cup \{\text{GOTO}(I, X)\}$
**until** $C$ did not change in this iteration
**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 states of a DFA, GOTO is its transition function
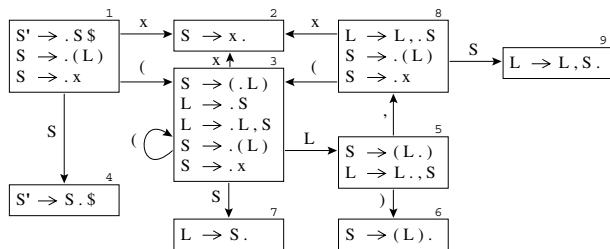
# Example

# 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 $A \to \alpha.a\beta$ is in $I_i$ and $\text{GOTO}(I_i, a) = I_j$, then $ACTION[i, a] = \text{Shift } j$

   2.2 If $A \to \alpha.$ is in $I_i$, then set $ACTION[i, a] = \text{Reduce } A \to \alpha$ for all terminals $a$.

   2.3 If $S' \to S.$ 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 $S' \to .S$

$\Rightarrow$ *LR(0) because the chosen action (shift or reduce) only depends on the current state (but the choice of the next state still depends on the token)*
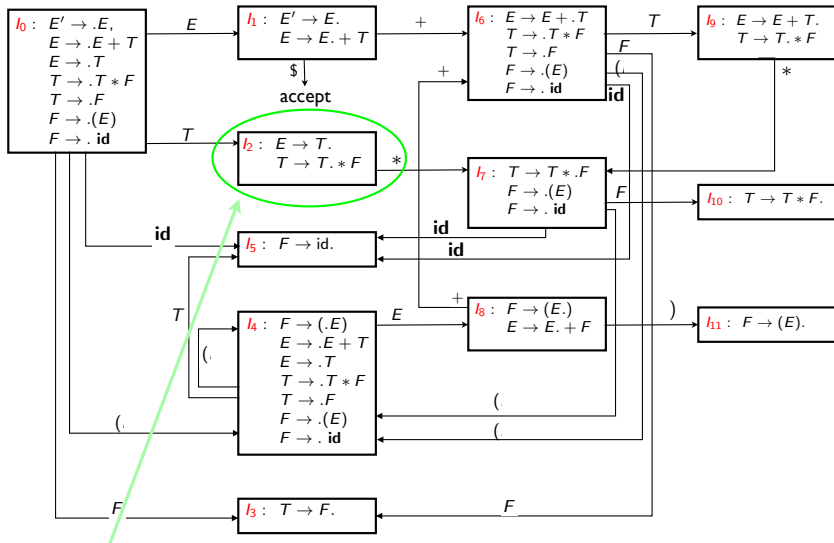
# Example of a LR(0) grammar

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



$I_0$: $E' \to .E$,
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .\mathbf{id}$

$I_1$: $E' \to E.$
$E \to E. + T$

accept

$I_6$: $E \to E + .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .\mathbf{id}$

$I_9$: $E \to E + T.$
$T \to T. * F$

$I_2$: $E \to T.$
$T \to T. * F$

$I_7$: $T \to T * .F$
$F \to .(E)$
$F \to .\mathbf{id}$

$I_{10}$: $T \to T * F.$

$I_5$: $F \to \mathbf{id}.$

$I_4$: $F \to (.E)$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .\mathbf{id}$

$I_8$: $F \to (E.)$
$E \to E. + F$

$I_{11}$: $F \to (E).$

$I_3$: $T \to F.$

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

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

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

$I_1:$   $S' \to S\cdot$

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

$I_3:$   $S \to R\cdot$

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

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

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

$I_7:$   $L \to *R\cdot$

$I_8:$   $R \to L\cdot$

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

(Dragonbook)

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

# Summary of SLR parsing

Construction of a SLR parser from a CFG grammar

- Eliminate ambiguity *(or not, see later)*
- Add the production $S' \to S$, 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 $S'' \to S'\$$ to the grammar and computing *First* and *Nullable*)
- Add the reduce actions in the action table according to *Follow*
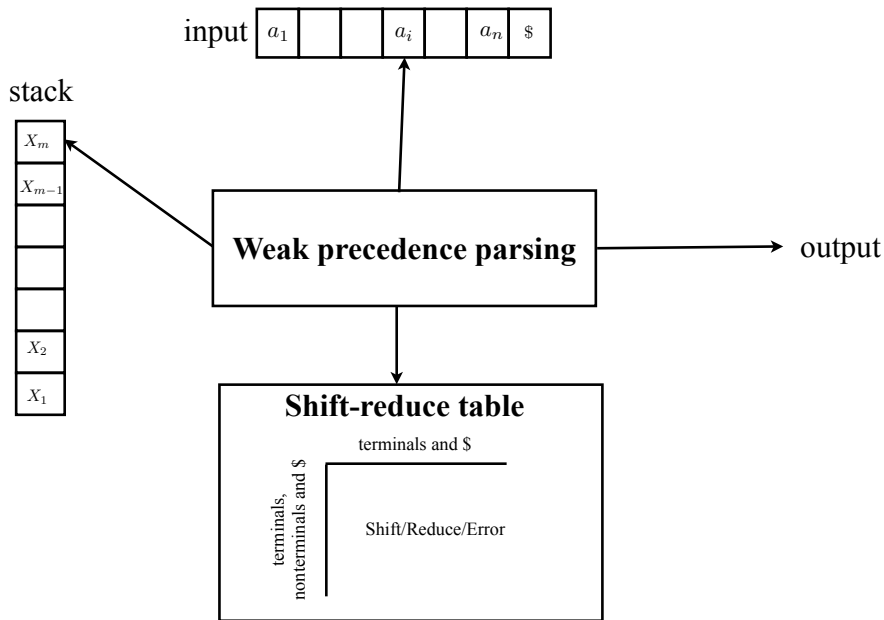- Check that the grammar is SLR *(and if not, try to resolve conflicts, see later)*

# Outline

# Operator precedence parsing

- Bottom-up parsing methods that follow the idea of shift-reduce parsers
- Several flavors: operator, simple, and weak precedence.
- In this course, only weak precedence

- Main differences compared to LR parsers:
  - There is no explicit state associated to the parser (and thus no state pushed on the stack)
  - The decision of whether to shift or reduce is taken based solely on the symbol on the top of the stack and the next input symbol (and stored in a shift-reduce table)
  - In case of reduction, the handle is the longest sequence at the top of stack matching the RHS of a rule

# Structure of the weak precedence parser

# Weak precedence parsing algorithm

Create a stack with the special symbol \$
$a = \text{GETNEXTTOKEN}()$
**while** (True)
    **if** (Stack== \$S and $a ==$ \$)
        break // Parsing is over
    $X_m = \text{TOP}(\text{Stack})$
    **if** ($SRT[X_m, a] = \text{shift}$)
        Push $a$ onto the stack
        $a = \text{GETNEXTTOKEN}()$
    **elseif** ($SRT[X_m, a] = \text{reduce}$)
        Search for the longest RHS that matches the top of the stack
        **if** no match found
            call error-recovery routine
        Let denote this rule by $Y \to X_{m-r+1} \ldots X_m$
        Pop $r$ elements off the stack
        Push $Y$ onto the stack
        Output $Y \to X_{m-r+1} \ldots X_m$
    **else** call error-recovery routine

# Example for the expression grammar

Example:

Shift/reduce table

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \textbf{id}$

|     | $*$ | $+$ | ( | ) | **id** | $ |
|-----|-----|-----|---|---|--------|---|
| E   |     | S   |   | S |        | R |
| T   | S   | R   |   | R |        | R |
| F   | R   | R   |   | R |        | R |
| $*$ |     |     | S |   | S      |   |
| $+$ |     |     | S |   | S      |   |
| (   |     |     | S |   | S      |   |
| )   | R   | R   |   | R |        | R |
| **id** | R | R |   | R |        | R |
| $   |     |     | S |   | S      |   |

# Example of parsing

| Stack | Input | Action |
|---|---|---|
| $\$$ | $id + id * id\$$ | Shift |
| $\$id$ | $+id * id\$$ | Reduce by $F \rightarrow$ **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 $F \rightarrow$ **id** |
| $\$E + F$ | $*id\$$ | Reduce by $T \rightarrow F$ |
| $\$E + T$ | $*id\$$ | Shift |
| $\$E + T*$ | $id\$$ | Shift |
| $\$E + T * id$ | $\$$ | Reduce by $F \rightarrow$ **id** |
| $\$E + T * F$ | $\$$ | Reduce by $T \rightarrow T * F$ |
| $\$E + T$ | $\$$ | Reduce by $E \rightarrow E + T$ |
| $\$E$ | $\$$ | Accept |

# Precedence relation: principle

- We define the (weak precedence) relations $\lessdot$ and $\gtrdot$ between symbols of the grammar (terminals or nonterminals)
  - $X \lessdot Y$ if $XY$ appears in the RHS of a rule or if $X$ precedes a reducible word whose leftmost symbol is $Y$
  - $X \gtrdot Y$ if $X$ is the rightmost symbol of a reducible word and $Y$ the symbol immediately following that word
- Shift when $X_m \lessdot a$, reduce when $X_m \gtrdot a$
- Reducing changes the precedence relation only at the top of the stack (there is thus no need to shift backward)

# Precedence relation: formal definition

- Let $G = (V, \Sigma, R, S)$ be a context-free grammar and $ a new symbol acting as left and right end-marker for the input word. Define $V' = V \cup \{\$\}$

- The weak precedence relations $\lessdot$ and $\gtrdot$ are defined respectively on $V' \times V$ and $V \times V'$ as follows:

  1. $X \lessdot Y$ if $A \to \alpha X B \beta$ is in $R$, and $B \overset{+}{\Rightarrow} Y\gamma$,
  2. $X \lessdot Y$ if $A \to \alpha X Y \beta$ is in $R$
  3. $\$ \lessdot X$ if $S \overset{+}{\Rightarrow} X\alpha$

  4. $X \gtrdot a$ if $A \to \alpha B \beta$ is in $R$, and $B \overset{+}{\Rightarrow} \gamma X$ and $\beta \overset{*}{\Rightarrow} a\gamma$
  5. $X \gtrdot \$$ if $S \overset{+}{\Rightarrow} \alpha X$

  for some $\alpha$, $\beta$, $\gamma$, and $B$

## Construction of the SR table: shift

Shift relation, $\lessdot$:

> Initialize $\mathcal{S}$ to the empty set.
> 1  add $\$ \lessdot S$ to $\mathcal{S}$
> 2  **for** each production $X \rightarrow L_1 L_2 \ldots L_k$
>         **for** $i = 1$ **to** $k - 1$
>             add $L_i \lessdot L_{i+1}$ to $\mathcal{S}$
> 3  **repeat**
>         **for** each* pair $X \lessdot Y$ in $\mathcal{S}$
>             **for** each production $Y \rightarrow L_1 L_2 \ldots L_k$
>                 Add $X \lessdot L_1$ to $\mathcal{S}$
>     **until** $\mathcal{S}$ did not change in this iteration.

* We only need to consider the pairs $X \lessdot Y$ with $Y$ a nonterminal that were added in $\mathcal{S}$ at the previous iteration

# Example of the expression grammar: shift

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \mathbf{id}$

| | |
|---|---|
| Step 1 | $S \lessdot \$$ |
| Step 2 | $E \lessdot +$ |
| | $+ \lessdot T$ |
| | $T \lessdot *$ |
| | $* \lessdot F$ |
| | $( \lessdot E$ |
| | $E \lessdot )$ |
| Step 3.1 | $+ \lessdot F$ |
| | $* \lessdot \mathbf{id}$ |
| | $* \lessdot ($ |
| | $( \lessdot T$ |
| Step 3.2 | $+ \lessdot \mathbf{id}$ |
| | $+ \lessdot ($ |
| | $( \lessdot F$ |
| Step 3.3 | $( \lessdot ($ |
| | $( \lessdot \mathbf{id}$ |

# Construction of the SR table: reduce

Reduce relation, $\gtrdot$:

>Initialize $\mathcal{R}$ to the empty set.
>1  add $S \gtrdot \$$ to $\mathcal{R}$
>2  **for** each production $X \rightarrow L_1 L_2 \ldots L_k$
>      **for** each pair $X \lessdot Y$ in $\mathcal{S}$
>         add $L_k \gtrdot Y$ in $\mathcal{R}$
>3  **repeat**
>      **for** each* pair $X \gtrdot Y$ in $\mathcal{R}$
>         **for** each production $X \rightarrow L_1 L_2 \ldots L_k$
>            Add $L_k \gtrdot Y$ to $\mathcal{R}$
>    **until** $\mathcal{R}$ did not change in this iteration.

\* We only need to consider the pairs $X \gtrdot Y$ with $X$ a nonterminal that were added in $\mathcal{R}$ at the previous iteration.

# Example of the expression grammar: reduce

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \mathbf{id}$

| Step 1 | $E \gtrdot \$$ |
| --- | --- |
| Step 2 | $T \gtrdot +$ |
| | $F \gtrdot *$ |
| | $T \gtrdot )$ |
| Step 3.1 | $T \gtrdot \$$ |
| | $F \gtrdot +$ |
| | $) \gtrdot *$ |
| | $\mathbf{id} \gtrdot *$ |
| | $F \gtrdot )$ |
| Step 3.2 | $F \gtrdot \$$ |
| | $) \gtrdot +$ |
| | $\mathbf{id} \gtrdot +$ |
| | $) \gtrdot )$ |
| | $\mathbf{id} \gtrdot )$ |
| Step 3.3 | $\mathbf{id} \gtrdot \$$ |
| | $) \gtrdot \$$ |

# Weak precedence grammars

- Weak precedence grammars are those that can be analysed by a weak precedence parser.
- A grammar $G = (V, \Sigma, R, S)$ is called a weak precedence grammar if it satisfies the following conditions:
    1. There exist no pair of productions with the same right hand side
    2. There are no empty right hand sides ($A \rightarrow \epsilon$)
    3. There is at most one weak precedence relation between any two symbols
    4. Whenever there are two syntactic rules of the form $A \rightarrow \alpha X \beta$ and $B \rightarrow \beta$, we don't have $X \lessdot B$
- Conditions 1 and 2 are easy to check
- Conditions 3 and 4 can be checked by constructing the SR table.

# Example of the expression grammar

### Shift/reduce table

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \textbf{id}$

|       | $*$ | $+$ | ( | ) | **id** | \$ |
|-------|-----|-----|---|---|--------|----|
| E     |     | S   |   | S |        | R  |
| T     | S   | R   |   | R |        | R  |
| F     | R   | R   |   | R |        | R  |
| $*$   |     |     | S |   | S      |    |
| $+$   |     |     | S |   | S      |    |
| (     |     |     | S |   | S      |    |
| )     | R   | R   |   | R |        | R  |
| **id**| R   | R   |   | R |        | R  |
| \$    |     |     | S |   | S      |    |

- Conditions 1-3 are satisfied (there is no conflict in the SR table)
- Condition 4:
  - $E \rightarrow E + T$ and $E \rightarrow T$ but we don't have $+ \lessdot E$ (see slide 204)
  - $T \rightarrow T * F$ and $T \rightarrow F$ but we don't have $* \lessdot T$ (see slide 204)

# Removing $\epsilon$ rules

- Removing rules of the form $A \rightarrow \epsilon$ is not difficult
- For each rule with $A$ in the RHS, add a set of new rules consisting of the different combinations of $A$ replaced or not with $\epsilon$.
- Example:

$$
\begin{aligned}
S &\rightarrow AbA | B \\
B &\rightarrow b | c \\
A &\rightarrow \epsilon
\end{aligned}
$$

is transformed into

$$
\begin{aligned}
S &\rightarrow AbA | Ab | bA | b | B \\
B &\rightarrow b | c
\end{aligned}
$$

# Summary of weak precedence parsing

Construction of a weak precedence parser

- Eliminate ambiguity *(or not, see later)*
- Eliminate productions with $\epsilon$ and ensure that there are no two productions with identical RHS
- Construct the shift/reduce table
- Check that there is no conflict during the construction
- Check condition 4 of slide 207

# Outline

# Using ambiguous grammars with bottom-up parsers

- All grammars used in the construction of Shift/Reduce parsing tables must be un-ambiguous
- We can still create a parsing table for an ambiguous grammar but there will be conflicts
- We can often resolve these conflicts in favor of one of the choices to disambiguate the grammar
- Why use an ambiguous grammar?
  - Because the ambiguous grammar is much more natural and the corresponding unambiguous one can be very complex
  - Using an ambiguous grammar may eliminate unnecessary reductions
- Example:

$$E \rightarrow E + E | E * E | (E) | \textbf{id} \quad \Rightarrow \quad \begin{array}{l} E \rightarrow E + T | T \\ T \rightarrow T * F | F \\ F \rightarrow (E) | \textbf{id} \end{array}$$

# Set of LR(0) items of the ambiguous expression grammar

$E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$

$Follow(E) = \{\$, +, *, )\}$
$\Rightarrow$ states 7 and 8 have shift/reduce conflicts for $+$ and $*$.

$I_0:$   $E' \to \cdot E$
      $E \to \cdot E + E$
      $E \to \cdot E * E$
      $E \to \cdot(E)$
      $E \to \cdot\mathbf{id}$

$I_1:$   $E' \to E\cdot$
      $E \to E\cdot + E$
      $E \to E\cdot * E$

$I_2:$   $E \to (\cdot E)$
      $E \to \cdot E + E$
      $E \to \cdot E * E$
      $E \to \cdot(E)$
      $E \to \cdot\mathbf{id}$

$I_3:$   $E \to \mathbf{id}\cdot$

$I_4:$   $E \to E + \cdot E$
      $E \to \cdot E + E$
      $E \to \cdot E * E$
      $E \to \cdot(E)$
      $E \to \cdot\mathbf{id}$

$I_5:$   $E \to E * \cdot E$
      $E \to \cdot E + E$
      $E \to \cdot E * E$
      $E \to \cdot(E)$
      $E \to \cdot\mathbf{id}$

$I_6:$   $E \to (E\cdot)$
      $E \to E\cdot + E$
      $E \to E\cdot * E$

$I_7:$   $E \to E + E\cdot$
      $E \to E\cdot + E$
      $E \to E\cdot * E$

$I_8:$   $E \to E * E\cdot$
      $E \to E\cdot + E$
      $E \to E\cdot * E$

$I_9:$   $E \to (E)\cdot$

(Dragonbook)

# Disambiguation

Example:

- Parsing of **id** $+$ **id** $*$ **id** will give the configuration

$$(0E1 + 4E7, *\textbf{id\$})$$

We can choose:
- $ACTION[7, *] =$ shift $5 \Rightarrow$ precedence to $*$
- $ACTION[7, *] =$ reduce $E \to E + E \Rightarrow$ precedence to $+$

- Parsing of **id** $+$ **id** $+$ **id** will give the configuration

$$(0E1 + 4E7, +\textbf{id\$})$$

We can choose:
- $ACTION[7, +] =$ shift $4 \Rightarrow +$ is right-associative
- $ACTION[7, +] =$ reduce $E \to E + E \Rightarrow +$ is left-associative

(same analysis for $I_8$)

# outline

# Top-down versus bottom-up parsing

- Top-down
  - Easier to implement (recursively), enough for most standard programming languages
  - Need to modify the grammar sometimes strongly, less general than bottom-up parsers
  - Used in most hand-written compilers and some parser generators (JavaCC, ANTLR)
- Bottom-up:
  - More general, less strict rules on the grammar, SLR(1) powerful enough for most standard programming languages
  - More difficult to implement, less easy to maintain (add new rules, etc.)
  - Used in most parser generators (Yacc, Bison)

# Hierarchy of grammar classes
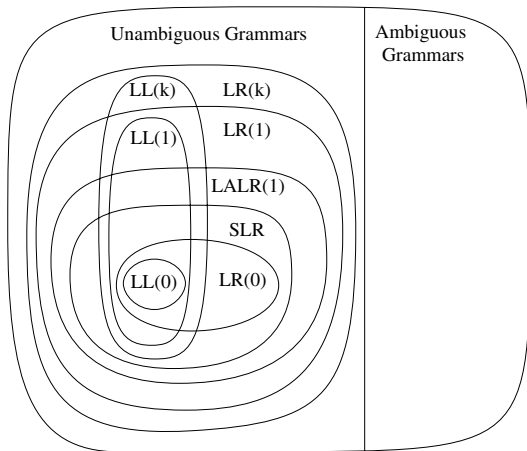


(Appel)

# Error detection and recovery

- In table-driven parsers, there is an error as soon as the table contains no entry (or an error entry) for the current stack (state) and input symbols

- The least one can do: report a syntax error and give information about the position in the input file and the tokens that were expected at that position

- In practice, it is however desirable to continue parsing to report more errors

- There are several ways to recover from an error:
  - ▶ Panic mode
  - ▶ Phrase-level recovery
  - ▶ Introduce specific productions for errors
  - ▶ Global error repair

# Panic-mode recovery

- In case of syntax error within a "phrase", skip until the next synchronizing token is found (e.g., semicolon, right parenthesis) and then resume parsing
- In LR parsing:
  - Scan down the stack until a state $s$ with a goto on a particular nonterminal $A$ is found
  - Discard zero or more input symbols until a symbol $a$ is found that can follow $A$
  - Stack the state $GOTO(s, A)$ and resume normal parsing

# Phrase-level recovery

- Examine each error entry in the parsing table and decide on an appropriate recovery procedure based on the most likely programmer error.
- Examples in LR parsing: $E \rightarrow E + E | E * E | (E) | id$
  - $id + *id$:
    $*$ is unexpected after a $+$: report a "missing operand" error, push an arbitrary number on the stack and go to the appropriate next state
  - $id + id) + id$:
    Report an "unbalanced right parenthesis" error and remove the right parenthesis from the input

# Other error recovery approaches

Introduce specific productions for detecting errors:

- Add rules in the grammar to detect common errors
- Examples for a *C* compiler:
  $I \rightarrow$ **if** $E$ $I$ (parenthesis are missing around the expression)
  $I \rightarrow$ **if** $(E)$ **then** $I$ (**then** is not needed in C)

Global error repair:

- Try to find *globally* the smallest set of insertions and deletions that would turn the program into a syntactically correct string
- Very costly and not always effective

# Building the syntax tree

- Parsing algorithms presented so far only check that the program is syntactically correct

- In practice, the parser also needs to build the parse tree (also called concrete syntax tree)

- Its construction is easily embedded into the parsing algorithm

- Top-down parsing:
  - Recursive descent: let each parsing function return the sub-trees for the parts of the input they parse
  - Table-driven: each nonterminal on the stack points to its node in the partially built syntax tree. When the nonterminal is replaced by one of its RHS, nodes for the symbols on the RHS are added as children to the nonterminal node

# Building the syntax tree

- Bottom-up parsing:
  - Each stack element points to a subtree of the syntax tree
  - When performing a reduce, a new syntax tree is built with the nonterminal at the root and the popped-off stack elements as children

- Note:
  - In practice, the concrete syntax tree is not built but rather a simplified (abstract) syntax tree
  - Depending on the complexity of the compiler, the syntax tree might even not be constructed
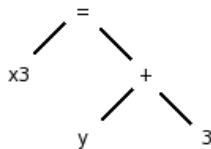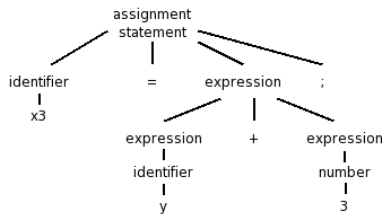
# For your project

- The choice of a parsing technique is left open for the project
- You can either use a parser generator or implement the parser by yourself
- Motivate your choice in your report and explain any transformation you had to apply to your grammar to make it fit the constraints of the parser

- Parser generators:
  - Yacc: Unix parser generator, LALR(1) (companion of Lex)
  - Bison: free implementation of Yacc, LALR(1) (companion of Flex)
  - ANTLR: LL(*), implemented in Java but output code in several languages
  - . . .
- `http://en.wikipedia.org/wiki/Comparison_of_parser_generators`

# An example with Flex/Bison

Example: Parsing of the following expression grammar:

$$
\begin{aligned}
\text{Input} &\rightarrow \text{Input Line} \\
\text{Input} &\rightarrow \epsilon \\
\text{Line} &\rightarrow \text{Exp EOL} \\
\text{Line} &\rightarrow \text{EOL} \\
\text{Exp} &\rightarrow \mathbf{num} \\
\text{Exp} &\rightarrow \text{Exp} + \text{Exp} \\
\text{Exp} &\rightarrow \text{Exp} - \text{Exp} \\
\text{Exp} &\rightarrow \text{Exp} * \text{Exp} \\
\text{Exp} &\rightarrow \text{Exp} / \text{Exp} \\
\text{Exp} &\rightarrow (\text{Exp})
\end{aligned}
$$

https://github.com/prashants/calc

## Flex file: calc.lex

```
%{
#define YYSTYPE double /* Define the main semantic type */
#include "calc.tab.h" /* Define the token constants    */
#include <stdlib.h>
%}
%option yylineno        /* Ask flex to put line number in yylineno */
white [ \t]+
digit [0-9]
integer {digit}+
exponent [eE][+-]?{integer}
real {integer}("."{integer})?{exponent}?
%%
{white} {}
{real} { yylval=atof(yytext); return NUMBER; }
"+" { return PLUS; }
"-" { return MINUS; }
"*" { return TIMES; }
"/" { return DIVIDE; }
"(" { return LEFT; }
")" { return RIGHT; }
"\n" { return END; }
. { yyerror("Invalid token"); }
```

# Bison file: calc.y

- Declaration:
  ```
  %{
  #include <math.h>
  #include <stdio.h>
  #include <stdlib.h>
  #define YYSTYPE double  /* Define the main semantic type */
  extern char *yytext;    /* Global variables of Flex */
  extern int yylineno;
  extern FILE *yyin;
  %}
  ```

- Definition of the tokens and start symbol
  ```
  %token NUMBER
  %token PLUS MINUS TIMES DIVIDE
  %token LEFT RIGHT
  %token END

  %start Input
  ```

# Bison file: calc.y

- Operator associativity and precedence:

```
%left PLUS MINUS
%left TIMES DIVIDE
%left NEG
```

- Production rules and associated actions:

```
%%

Input:                    /* epsilon */
        | Input Line
;

Line:
        END
        | Expression END { printf("Result: %f\n", $1); }
;
```

# Bison file: calc.y

- Production rules and actions (continued):

```
Expression:
        NUMBER { $$ = $1; }
        | Expression PLUS Expression { $$ = $1 + $3; }
        | Expression MINUS Expression { $$ = $1 - $3; }
        | Expression TIMES Expression { $$ = $1 * $3; }
        | Expression DIVIDE Expression { $$ = $1 / $3; }
        | MINUS Expression %prec NEG { $$ = -$2; }
        | LEFT Expression RIGHT { $$ = $2; }
    ;
```

- Error handling:

```
%%

int yyerror(char *s)
{
        printf("%s on line %d - %s\n", s, yylineno, yytext);
}
```

# Bison file: calc.y

- Main functions:

```c
int main(int argc, char **argv)
{
    /* if any input file has been specified read from that */
    if (argc >= 2) {
        yyin = fopen(argv[1], "r");
        if (!yyin) {
            fprintf(stderr, "Failed to open input file\n");
        }
        return EXIT_FAILURE;
    }

    if (yyparse()) {
        fprintf(stdout, "Successful parsing\n");
    }

    fclose(yyin);
    fprintf(stdout, "End of processing\n");
    return EXIT_SUCCESS;
}
```

# Bison file: makefile

- How to compile:
  ```
  bison -v -d calc.y
  flex -o calc.lex.c calc.lex
  gcc -o calc calc.lex.c calc.tab.c -lfl -lm
  ```

- Example:
  ```
  >./calc
  1+2*3-4
  Result: 3.000000
  1+3*-4
  Result: -11.000000
  *2
  syntax error on line 3 - *
  Successful parsing
  End of processing
  ```

## The state machine

Excerpt of calc.output (with *Expression* abbreviated in *Exp*):

```
state 9
    6 Exp: Exp . PLUS Exp
    7    | Exp . MINUS Exp
    8    | Exp . TIMES Exp
    9    | Exp . DIVIDE Exp
   10    | MINUS Exp .

    $default  reduce using rule 10 (Exp)

state 10
    6 Exp: Exp . PLUS Exp
    7    | Exp . MINUS Exp
    8    | Exp . TIMES Exp
    9    | Exp . DIVIDE Exp
   11    | LEFT Exp . RIGHT

    PLUS    shift, and go to state 11
    MINUS   shift, and go to state 12
    TIMES   shift, and go to state 13
    DIVIDE  shift, and go to state 14
    RIGHT   shift, and go to state 16
```

```
state 11
    6 Exp: Exp PLUS . Exp

    NUMBER  shift, and go to state 3
    MINUS   shift, and go to state 4
    LEFT    shift, and go to state 5

    Exp  go to state 17
```