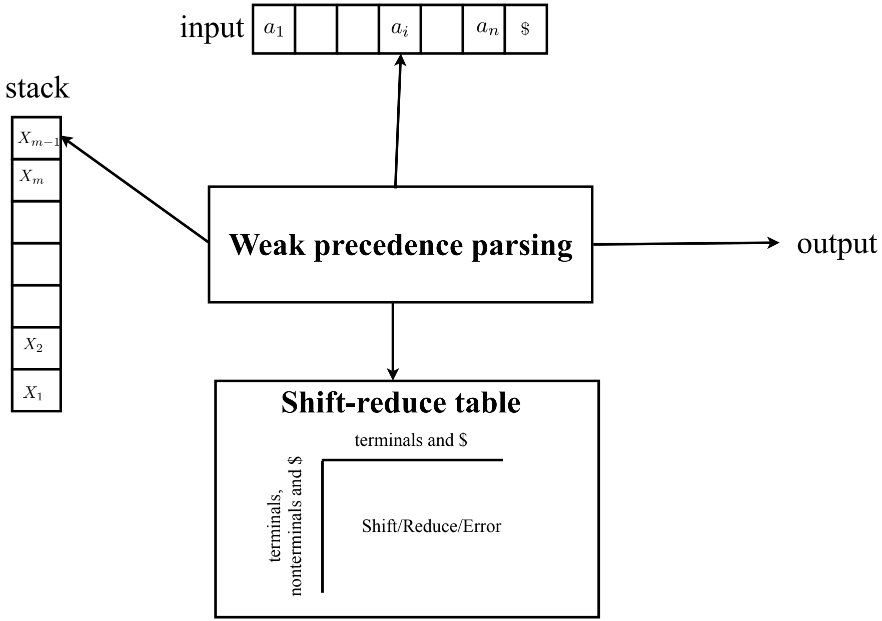
Operator precedence parsing

- 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 with respect 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 $
$a$ = GETNEXTTOKEN()
while (True)
    if (Stack== $\$S$ and $a == \$$)
        break // Parsing is over
    $X_m$ = TOP(Stack)
    if ($SRT[X_m, a]$ = shift)
        Push $a$ onto the stack
        $a$ = GETNEXTTOKEN()
    elseif ($SRT[X_m, a]$ = 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 $Y \rightarrow X_{m-r+1} \dots X_m$
        Pop $r$ elements off the stack
        Push $Y$ onto the stack
        Output $Y \rightarrow X_{m-r+1} \dots X_m$
    else call error-recovery routine
```

Example for the expression grammar

Example:

Shift/reduce table

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

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

  4. $X \gtrdot a$ if $A \rightarrow \alpha B \beta$ 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 α, β, γ, 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

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow$ **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 $X \to L_1 L_2 \ldots L_k$
        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 $X \to L_1 L_2 \ldots L_k$
                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

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow$ **id**

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

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

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow$ **id**

Shift/reduce table

	$*$	$+$	(	)	**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 202)
  - $T \rightarrow T * F$ and $T \rightarrow F$ but we don't have $* \lessdot T$ (see slide 202)

Removing ϵ 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 ϵ.
- Example:

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

is transformed into

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

Summary of weak precedence parsing

Construction of a weak precedence parser

- Eliminate ambiguity *(or not, see later)*
- Eliminate productions with ϵ and ensure that there are no two productions with identical RHS
- Construct the shift/reduce table
- Check that there are no conflict during the construction
- Check condition 4 of slide 205

Using ambiguous grammars with bottom-up parsers

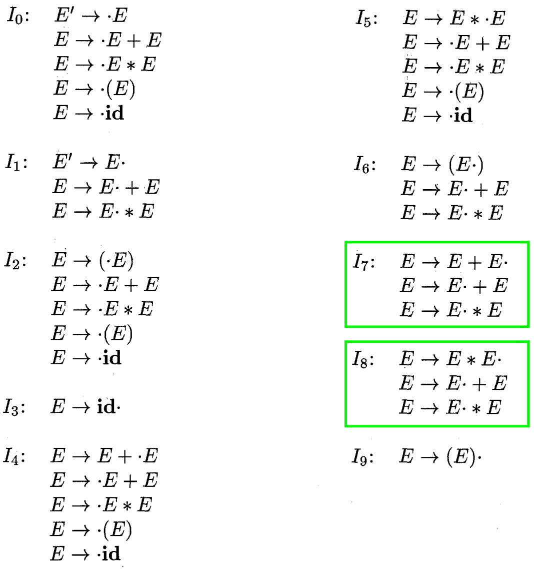
- All grammars used in the construction of Shift/Reduce parsing tables must be un-ambiguous
- We can still create a parsing table for an ambiguous grammar but there will be conflicts
- We can often resolve these conflicts in favor of one of the choices to disambiguate the grammar
- Why use an ambiguous 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

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

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



$$\begin{array}{ll} I_0: & E' \rightarrow \cdot E \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \textbf{id} \end{array}$$

$$\begin{array}{ll} I_1: & E' \rightarrow E \cdot \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}$$

$$\begin{array}{ll} I_2: & E \rightarrow (\cdot E) \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \textbf{id} \end{array}$$

$$\begin{array}{ll} I_3: & E \rightarrow \textbf{id} \cdot \end{array}$$

$$\begin{array}{ll} I_4: & E \rightarrow E + \cdot E \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \textbf{id} \end{array}$$

$$\begin{array}{ll} I_5: & E \rightarrow E * \cdot E \\ & E \rightarrow \cdot E + E \\ & E \rightarrow \cdot E * E \\ & E \rightarrow \cdot (E) \\ & E \rightarrow \cdot \textbf{id} \end{array}$$

$$\begin{array}{ll} I_6: & E \rightarrow (E \cdot) \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}$$

$$\boxed{\begin{array}{ll} I_7: & E \rightarrow E + E \cdot \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}}$$

$$\boxed{\begin{array}{ll} I_8: & E \rightarrow E * E \cdot \\ & E \rightarrow E \cdot + E \\ & E \rightarrow E \cdot * E \end{array}}$$

$$\begin{array}{ll} I_9: & E \rightarrow (E) \cdot \end{array}$$

(Dragonbook)

Disambiguation

Example:

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

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

  We can choose:

  - $ACTION[7, *]$ =shift $\Rightarrow$ precedence to $*$
  - $ACTION[7, *]$ =reduce $E \rightarrow E + E$ $\Rightarrow$ precedence to $+$

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

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

  We can choose:

  - $ACTION[7, +]$ =shift $\Rightarrow$ $+$ is right-associative
  - $ACTION[7, +]$ =reduce $E \rightarrow E + E$ $\Rightarrow$ $+$ is left-associative

(same analysis for I_8)

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 a “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

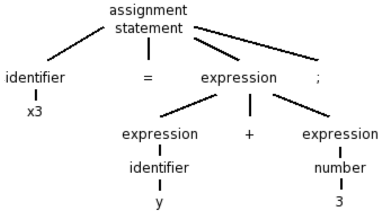
- Parsing algorithms presented so far only check that the program is syntactically correct
- In practice, the parser needs also to build the parse tree (also called concrete syntax tree)
- Its construction is easily embedded into the parsing algorithm

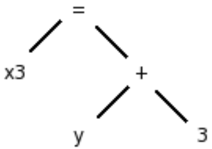
- Top-down parsing:
  - Recursive descent: let each parsing function return the sub-trees for the parts of the input they parse
  - Table-driven: each nonterminal on the stack points to its node in the partially built syntax tree. When the nonterminal is replaced by one of its RHS, nodes for the symbols 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 an simplified abstract syntax tree
  - Depending on the complexity of the compiler, the syntax tree might even not be constructed





Conclusion: 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
- 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 like Yacc or Bison (but JavaCC is top-down)

For your project

- The choice of a parsing technique is left open for the project but we ask you to implement the parser by yourself (Yacc, bison or other parser generators are forbidden)
- Weak precedence parsing was the recommended method in previous implementations of this course
- Motivate your choice in your report and explain any transformation you had to apply to your grammar to make it fit the parser's constraints
- To avoid mistakes, you should build the parsing tables by program