Operator Precedence parser

Operator Precedence Parser

- **Operator grammar**
  - small, but an important class of grammars
  - we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
- In an *operator grammar*, no production rule can have:
  - ε at the right side
  - two adjacent non-terminals at the right side.

- Ex:

$E \rightarrow AB$ | $A \rightarrow a$ | $B \rightarrow b$

not operator grammar

$E \rightarrow EOE$ | $E \rightarrow id$ | $O \rightarrow + | * | /$

not operator grammar

$E \rightarrow E+E \mid E*E \mid E/E \mid id$

operator grammar

Operator Precedence Grammar

Let G be an ∈-free operator grammar(No ∈-Production).For each terminal symbols *a* and *b*, the following conditions need to be satisfied.

1. $a \doteq b$, if ∃ a production in RHS of the form $\alpha a \beta b \gamma$, where β is either ∈ or a single non Terminal. <u>*Ex*</u> *S→iCtSeS implies* $i \doteq t$ *and* $t \doteq e$.
2. $a \lessdot b$ if for some non-terminal A ∃ a production in RHS of the form $A \rightarrow \alpha a A \beta$, and $A \Rightarrow^{+} \gamma b \delta$ where γ is either ∈ or a single non-terminal. <u>*Ex*</u> *S→iCtS and* $C \Rightarrow^{+} b$ *implies* $i \lessdot b$.
3. $a \gtrdot b$ if for some non-terminal A ∃ a production in RHS of the form $A \rightarrow \alpha A b \beta$, and $A \Rightarrow^{+} \gamma a \delta$ where δ is either ∈ or a single non-terminal.

   <u>*Ex*</u> *S→iCtS and* $C \Rightarrow^{+} b$ *implies* $b \gtrdot t$.

Example

$E \Rightarrow E+E \Rightarrow E+E+E$

E→E+E | E*E | (E) | id is not a Operator precedence Grammar

- By Rule no. 3 we have + $\lessdot$ + & + $\gtrdot$ +. Where as we can modify the Grammar is as follow
- E→E+T | T, T→T*F | F, F→(E) | id

Precedence relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

$a \lessdot b$	b has higher precedence than a
$a \doteq b$	b has same precedence as a
$a \gtrdot b$	b has lower precedence than a

Precedence Relations

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators.
- Unary minus causes a problem

Operator Precedence

- The intention of the precedence relations is to find the handle of a right-sentential form,

  $<\cdot$ with marking the left end,

  $\doteq$ appearing in the interior of the handle, and

  $\cdot>$ marking the right hand.

Parsing

- In our input string $\$a_1a_2...a_n\$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Example

$E \rightarrow E+E \;|\; E-E \;|\; E*E \;|\; E/E \;|\; E\text{^}E \;|\; (E) \;|\; -E \;|\; id$

$ id +id *id $

- Then the input string id+id*id with the precedence relations inserted will be:

$$\$ \lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$$

Operator Precedence relation table

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Parsing

1. Scan the string from left end until the first $\gtrdot$ is encountered.
2. Then scan backwards (to the left) over any $\doteq$ until a $\lessdot$ is encountered.
3. The handle contains everything to left of the first $\gtrdot$ and to the right of the $\lessdot$ is encountered.

$\lessdot \alpha \gtrdot$ $\qquad$ $\lessdot \alpha \doteq \beta \gtrdot$

Parsing

Stack	Rule	Input
$ <· id ·> + <· id ·> * <· id ·> $	E → id	$ id + id * id $
$ <· + <· id ·> * <· id ·> $	E → id	$ E + id * id $
$ <· + <· * <· id ·> $	E → id	$ E + E * id $
$ <· + <· * ·> $	E → E*E	$ E + E * ·E $
$ <· + ·> $	E → E+E	$ E + E $
$$		

Operator Precedence Parsing

- Ensure the Grammar satisfies the pre-requisite
- Compute Leading and Trailing
- Construct the Operator precedence parsing table
- Parse the string based on the algorithm

- LEADING: for each NT, those terminals that can be the first terminal in a string derived from that NT

- TRAILING: for each NT, those terminals that can be the last terminal in a string derived from that NT

Leading and Trailing

To produce the Operator precedence Table we have to follow the procedure as:

LEADING(A) = { a| A $\Rightarrow^{+} \gamma a \delta$, where γ is ϵ or a single non-terminal.}

TRAILING(A) = { a| A $\Rightarrow^{+} \gamma a \delta$, where δ is ϵ or a single non-terminal.}

Leading

- Based on two rules
- a is in Leading(A) if A → $\gamma a \delta$ where γ is ε or any Non-Terminal
- If a is in Leading(B) and A → Bα, then

  a in Leading(A)

Trailing

- Based on two rules
- a is in Trailing(A) if A → $\gamma a \delta$ where δ is ε or any Non-Terminal
- If a is in Trailing(B) and A → αB, then

  a in Trailing(A)

+ <· Leading(T)

(< Leading(E)

Example

- E → E + T
- E → T
- T → T * F
- T → F
- F → (E)
- F → id

Leading(E) = {+, *, (, id}

Leading(T) = {*, (, id}

Leading(F) = {(, id}

Trailing(E) = {+, *,), id}

Trail(T) = {*,), id}

Trail(F) = {), id}

	+	*	(	)	id	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	·>
)	·>	·>		·>		·>
id	·>	·>		·>		·>
$	<·	<·	<·	<·	<·	O

error

$ $

E

NT T

Trail(E) ·> +

Trail(T) ·> *

Trail(F) ·>)

Leading

- Leading (E) = { + , * , (, id}
- Leading (T) = { *, (, id }
- Leading (F) = {(, id}

Trailing

- Trailing(E) = { +, *,), id}
- Trailing (T) = { *,), id}
- Trailing (F) = {), id}

Operator Precedence Relations

$x_i \; x_{i+1} \; x_{i+2}$

$i \, C \, t \, S$

For each production $A \rightarrow X_1X_2X_3 \ldots X_n$

for i = 1 to n-1

ϵ or NT

$\alpha \, a \, b \, b \, \gamma$

if X_i and X_{i+1} are terminals

set $X_i \doteq X_{i+1}$

if $i \leq n\text{-}2$ and X_i and X_{i+2} are terminals and X_{i+1} is a non-terminal

set $X_i \doteq X_{i+2}$

if X_i is a terminal and X_{i+1} is a non-terminal then for all 'a' in

Leading(X_{i+1}) set X_i <· **a**

$\alpha c B$ $\quad$ $c <\cdot a$

if X_i is a non-terminal and X_{i+1} is a terminal then for all 'a' in

Trailing(X_i) set a ·> X_{i+1}

Bc $\quad$ $a \cdot> c$

	+	-	*	/	^	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
^	·>	·>	·>	·>	<·	<·	<·	·>	·>

	+	-	*	/	^	id	(	)	$
id	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$			$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\doteq$	
)	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$			$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$		

Parsing algorithm

```
set p to point to the first symbol of w$ ;
repeat forever
  if ( $ is on top of the stack and p points to $ ) then return
  else {
    let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;
    if ( a <· b or a ≐ b ) then {               /* SHIFT */
      push b onto the stack;
      advance p to the next input symbol;
    }
    else if ( a ·> b ) then                     /* REDUCE */
      repeat pop stack
      until ( the top of stack terminal is related by <· to the terminal most recently popped );
    else error();
  }
```

Parsing Algorithm

- The input string is w$, the initial stack is $ and a table holds precedence relations between the necessary terminals

Parsing

Stack	Input	Action
$	id+id*id$	$ <· id shift
$id	+id*id$	id ·> + reduce E → id
$	+id*id$	Shift
$+	id*id$	Shift
$ + id	*id$	id ·> * reduce E → id
$ +	* id $	Shift
$ + *	id $	Shift
$ + * id	$	id ·> $ reduce E → id

Parsing

Stack	Input	Action
$+*	$	$* \gtrdot \$$ reduce $E \rightarrow E*E$
$ +	$	$+ \gtrdot \$$ reduce $E \rightarrow E+E$
$	$	Accept

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

Unary minus

- Operator-Precedence parsing cannot handle the unary minus if the grammar has binary subtraction operator.
- The best approach to solve this problem is to tackle at the lexical phase
  - The lexical analyzer can be made to return two different tokens for the unary minus and the binary minus.
  - The lexical analyzer will need a lookhead to distinguish the binary minus from the unary minus.

Unary minus – Precedence set

- Then, we make

  $\theta \lessdot$ unary-minus for any operator

  unary-minus $\gtrdot \theta$ if unary-minus has higher precedence than θ

  unary-minus $\lessdot \theta$ if unary-minus has lower (or equal) precedence than θ

Operator Precedence Parser

- The precedence table is typically stored as a precedence function.
- The precedence table is coded as two functions f() and g()

Precedence function computation

- For symbols a and b.

$f(a) < g(b)$	if $a \lessdot b$
$f(a) = g(b)$	if $a \doteq b$
$f(a) > g(b)$	if $a \gtrdot b$

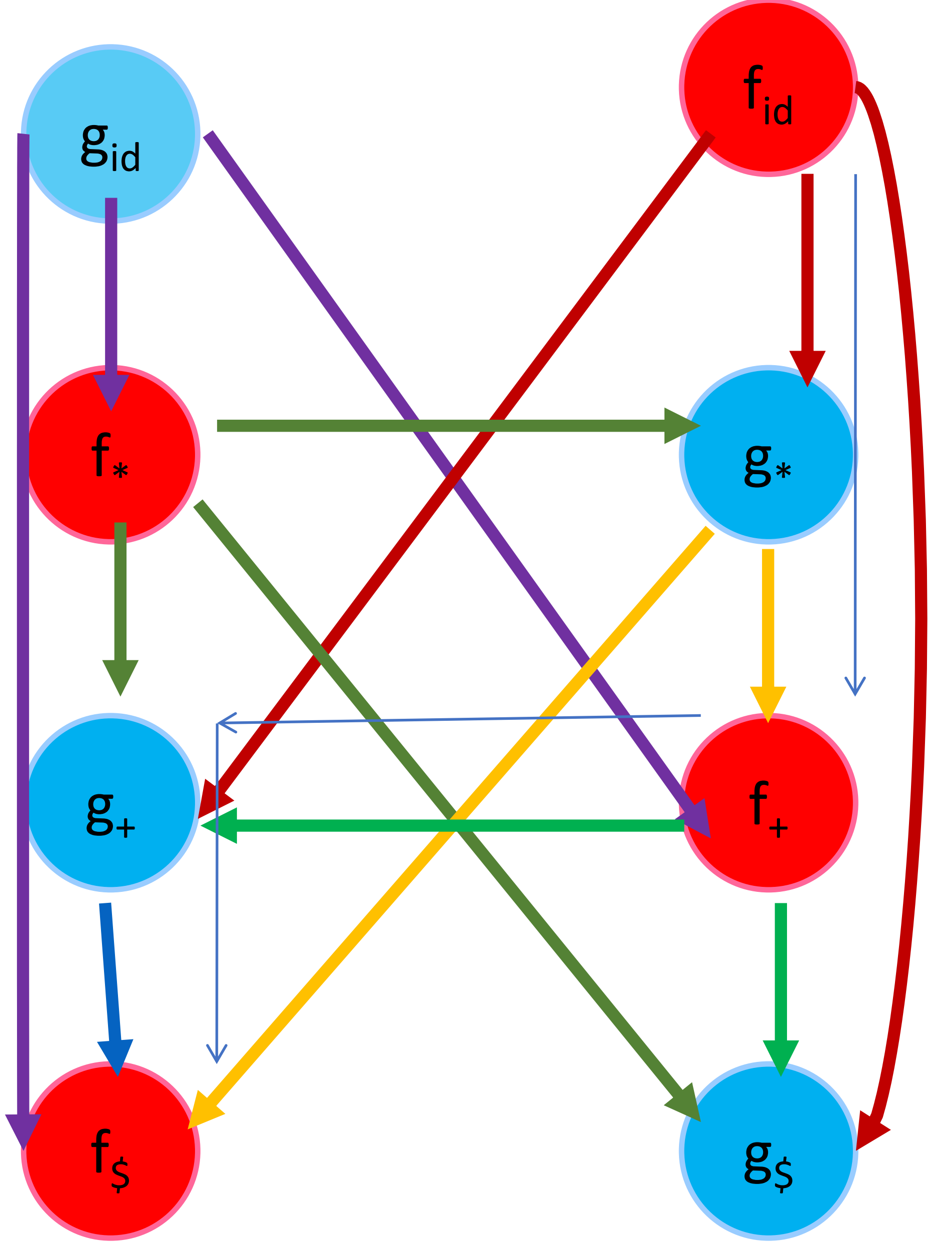
Precedence function algorithm

- Create symbols $\mathbf{f_a}$ and $\mathbf{g_b}$ for each **a** that is a terminal or \$.
- Partition the created symbols into as many groups as possible, in such a way that if $a \doteq b$, then $\mathbf{f_a}$ and $\mathbf{g_b}$ are in the same group.
- Create a directed graph whose nodes are the groups found in the previous step. For any 'a' and 'b', if $a <\cdot\, b$, place an edge from the group of $\mathbf{g_b}$ to the group of $\mathbf{f_a}$. If $a \cdot\!> b$, place an edge from the group of $\mathbf{f_a}$ to that of $\mathbf{g_b}$.

- If the graph constructed has a cycle, then no precedence functions exist. If there are no cycle, let f(a) be the length of the longest path beginning at the group of $\mathbf{f_a}$; let g(a) be the length of the longest path beginning at the group of $\mathbf{g_a}$.

Example

- Consider, the expression grammar
- f+, f*, fid, f$ are the four functions of ‘f’
- g+, g*, gid, g$ are the four function of ‘g’



	+	*	id	$
f	2	4	4	0
g	1	3	5	0

Precedence function

- The length of the longest path is calculated from every node to other node

Drawbacks

- It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
- Small class of grammars.
- Difficult to decide the language of the grammar.

Error situations

- No relation between the terminal on the top of stack and the next input symbol.
- A handle is found (reduction step), but there is no production with this handle as RHS

Error Recovery

1. As in the LL(1) parser, each empty entry is filled with a pointer to an error routine.
2. Matches what the popped handle resembles which right hand side of the production and tries to recover from that situation.

Shift/Reduce Errors

- To recover, we must modify (insert/change)
  - Stack or
  - Input or
  - Both.

Precedence Table

	id	**(**	**)**	**$**
id	**e3**	**e3**	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$\doteq$	**e4**
)	**e3**	**e3**	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	e2	e1

Error Recovery

e1: Scenario: Entire expression is missing

- insert **id** to the input
- issue message: ‘missing operand’ or ‘no input’

Error recovery

- e2: Scenario: Expression begins with a right parenthesis
  - delete) from the input
  - issue message: ‘unbalanced right parenthesis’

Error recovery

- e3: Scenario: id or) is followed by id or (
  - insert **+** to the input
  - issue message: 'missing operator'

id + id

id (id+id)

E) (E)

Error recovery

- e4: Scenario: expression ends with a left parenthesis
  - pop (from the stack
  - issue message: ‘missing right parenthesis’

Example

S → i C t S e S | a

C → b

Leading (S) = { a, i }

Leading (C) = {b}

Trailing (S) = { e, a}

Trailing (C) = {b}

Parsing Table

S → i C t S e S | a
C → b

	i	t	e	a	b	$
i		$\doteq$			$\lessdot$	$\gtrdot$
t	$\lessdot$		$\doteq\lessdot$	$\lessdot$		$\gtrdot$
e	$\lessdot$		$\lessdot$ $\gtrdot$	$\lessdot$		$\gtrdot$
a			$\gtrdot$			$\gtrdot$
b		$\gtrdot$				$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	

$i \doteq t \quad t \doteq e$

iC   tS   eS

$i \lessdot \text{Lead}(C) \quad t \lessdot \text{Lead}(S) \quad e \lessdot \text{Lead}(S)$

$i \lessdot b \quad t \lessdot \{a, i\} \quad e \lessdot \{a, i\}$

Ct   Se

$\text{Trail}(C) \gtrdot t \quad \text{Trail}(S) \gtrdot e$

$\cancel{b \lessdot t} \quad \{e, a\} \gtrdot e$

$b \gtrdot t$

Parsing for if – then grammar

Stack	Input	Action
$	ibtaea$	$ <· i , shift
$ i	b t a e a $	i <· b, shift
$ i b	t a e a $	b ·> t, reduce ic C→b
$ i	t a e a $	i ≐ t, shift
$ i t	a e a $	t <· a, shift
$ i t a	e a $	a ·> e, reduce ictS S→a

If – then grammar parsing

Stack	If	Action
$ i t a	e a $	a ·> e, reduce iCtS
$ i t	e a $	t ≐ e, shift
$ i t e	a $	e <· a, shift
$ i t e a	$	a ·> $, reduce S → a iCtSeS
$ i t e	$	all ·> $, reduce S
$	$	accept

Summary

- Parsing action of Operator Precedence Parser
- Precedence functions
- Error recovery strategies