Bottom-up Parsing

Govind kumar Jha
Assistant Professor,
Department of CSE
BCE, Bhgalpur

Parsing Techniques

Top-down parsers *(LL(1), recursive descent)*

- Start at the root of the parse tree from the start symbol and grow toward leaves (similar to a derivation)
- Pick a production and try to match the input
- Bad "pick" $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers *(LR(1), operator precedence)*

- Start at the leaves and grow toward root
- We can think of the process as reducing the input string to the start symbol
- At each reduction step a particular substring matching the right-side of a production is replaced by the symbol on the left-side of the production
- Bottom-up parsers handle a large class of grammars

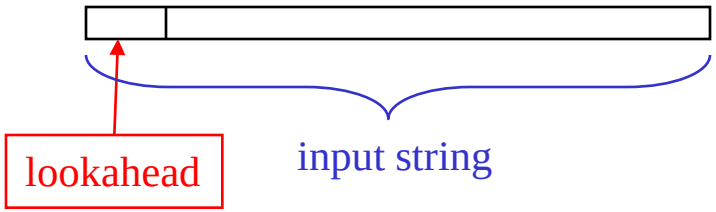
Top-down Parsing

start symbol

S

left-to-right scan

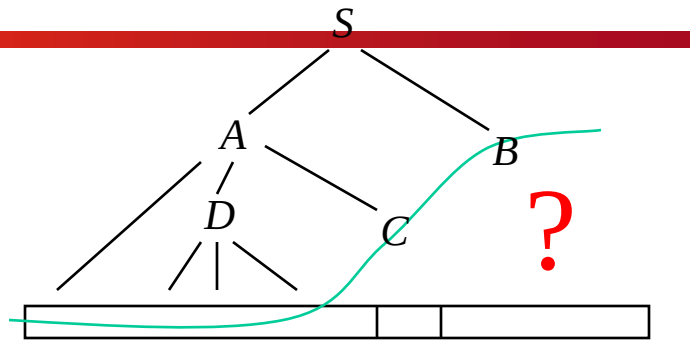
?



lookahead

input string

fringe of the parse tree



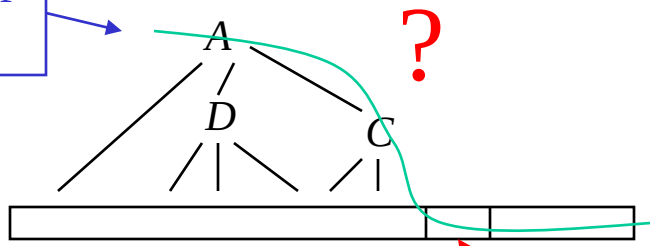
lookahead

left-most derivation

Bottom-up Parsing

S

upper fringe of the parse tree



right-most derivation in reverse

lookahead

Bottom-up Parsing

A general style of bottom-up syntax analysis, known as shift-reduce parsing.

Two types of bottom-up parsing:

1. Operator-Precedence parsing
2. LR parsing

Bottom Up Parsing

- "Shift-Reduce" Parsing
- Reduce a string to the start symbol of the grammar.
- At every step a particular sub-string is matched (in left-to-right fashion) to the right side of some production and then it is substituted by the non-terminal in the left hand side of the production.

Consider:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

abbcde
aAbcde
aAde
aABe
S

Reverse order

Rightmost Derivation:

$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$

Handles

- Handle of a string: Substring that matches the RHS of some production AND whose reduction to the non-terminal on the LHS is a step along the reverse of some rightmost derivation.
- **Formally:**
  - **handle of a right sentential form γ is <A → β, location of β in γ> ,** *that satisfies the above property.*
  - i.e. A → β is a handle of αβγ at the location immediately after the end of α, if:

$$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A \gamma \underset{rm}{\Rightarrow} \alpha\beta\gamma$$

- A certain sentential form may have many different handles.
- Right sentential forms of a non-ambiguous grammar have one *unique* handle

Example

Consider:

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

$$S \Rightarrow \underline{aABe} \Rightarrow aA\underline{d}e \Rightarrow a\underline{Abc}de \Rightarrow a\underline{b}bcde$$

It follows that:

$S \rightarrow aABe$ is a handle of $\underline{aABe}$ in location 1.

$B \rightarrow d$ is a handle of $aA\underline{d}e$ in location 3.

$A \rightarrow Abc$ is a handle of $a\underline{Abc}de$ in location 2.

$A \rightarrow b$ is a handle of $a\underline{b}bcde$ in location 2.

Handle Pruning

- A rightmost derivation in reverse can be obtained by "handle-pruning."
- Apply this to the previous example.

S → aABe

A → Abc | b

B → d

abbcde

Find the handle = b at loc. 2

aAbcde

b at loc. 3 is not a handle:

aAAcde

... blocked.

Also Consider:

E → E + E | E * E |

| (E) | id

Derive id+id*id
By two different Rightmost derivations

Handle-pruning, Bottom-up Parsers

The process of discovering a handle & reducing it to the appropriate left-hand side is called *handle pruning*.
Handle pruning forms the basis for a bottom-up parsing method.

To construct a rightmost derivation

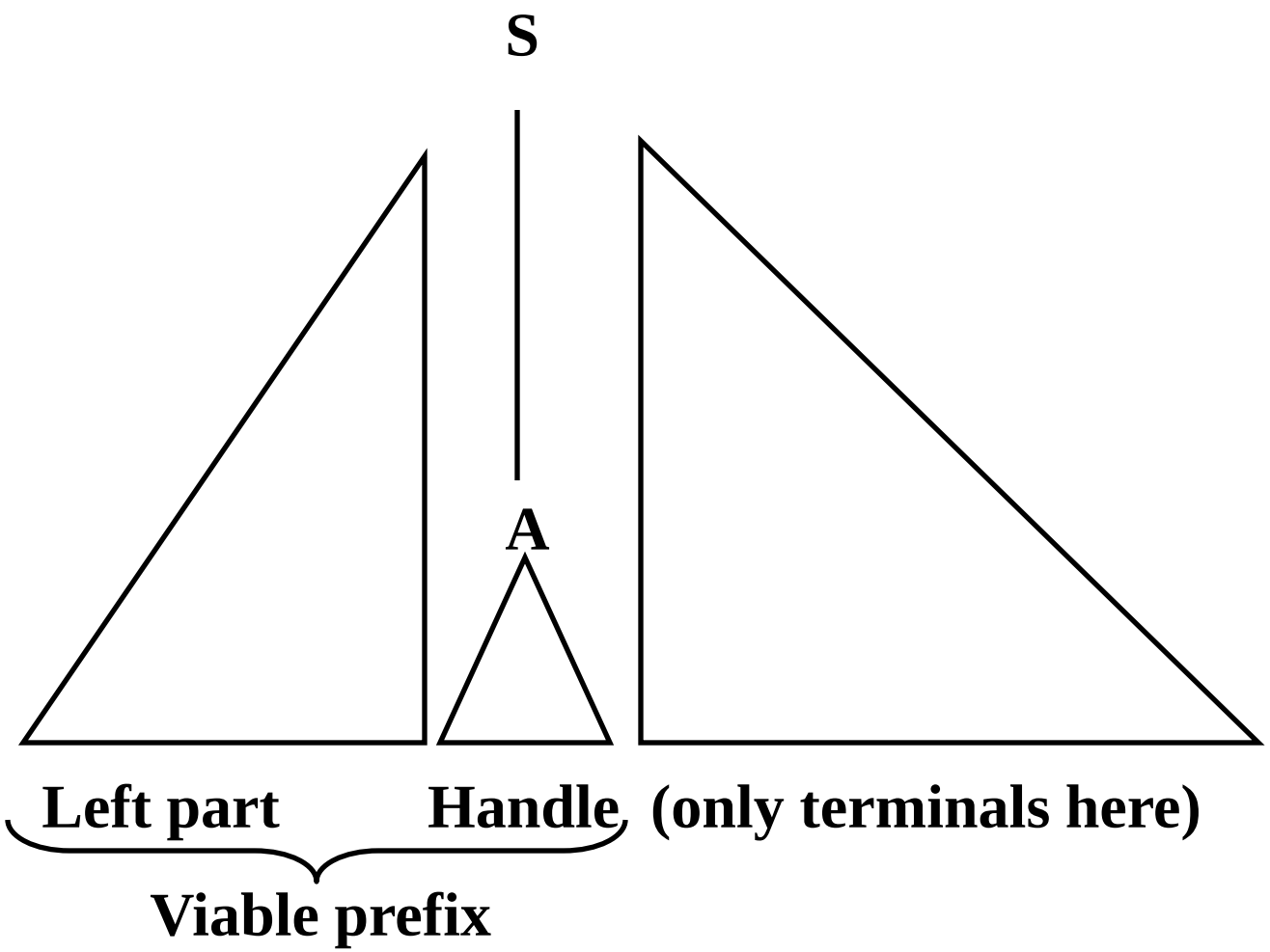
$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

Apply the following simple algorithm

for $i \leftarrow n$ *to* 1 *by* -1
    Find the handle $A_i \rightarrow \beta_i$ in γ_i
    Replace β_i *with* A_i *to generate* γ_{i-1}

Handle Pruning, II

- Consider the cut of a parse-tree of a certain right sentential form.



Example

1	*S*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* – *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	num
9		/	id

The expression grammar

Sentential Form	*Handle Prod'n , Pos'n*
S	—
Expr	1,1
Expr – *Term*	3,3
Expr – *Term* * *Factor*	5,5
Expr – *Term* * <id,y>	9,5
Expr – *Factor* * <id,y>	7,3
Expr – <num,2> * <id,y>	8,3
Term – <num,2> * <id,y>	4,1
Factor – <num,2> * <id,y>	7,1
<id,x> – <num,2> * <id,y>	9,1

Handles for rightmost derivation of input string:

x – 2 * y

Shift Reduce Parsing with a Stack

- Two problems:
  - locate a handle and
  - decide which production to use (if there are more than two candidate productions).

- General Construction: using a stack:
  - "shift" input symbols into the stack until a handle is found on top of it.
  - "reduce" the handle to the corresponding non-terminal.

  - other operations:
    - "accept" when the input is consumed and only the start symbol is on the stack, also: "error"

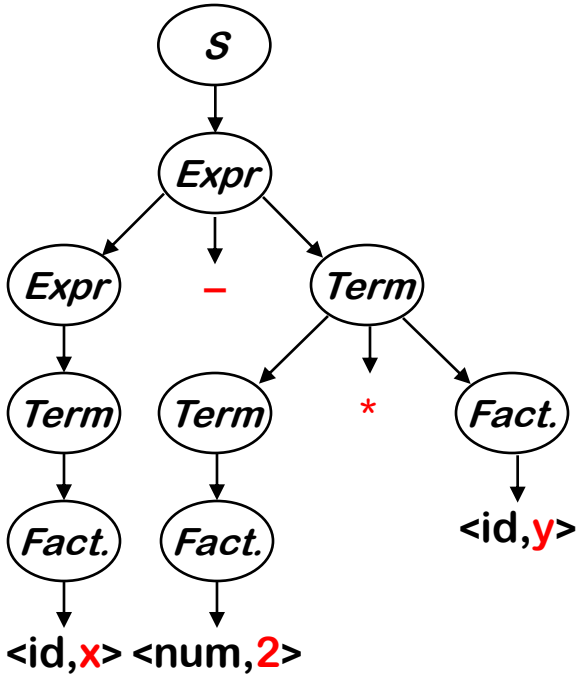
Example

STACK	INPUT	Remark
$	id + id * id$	Shift
$ id	+ id * id$	Reduce by E → id
$E	+ id * id$	

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

Example, Corresponding Parse Tree

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id		9,5	red. 9
$ *Expr* – *Term* * *Factor*		5,5	red. 5
$ *Expr* – *Term*		3,3	red. 3
$ *Expr*		1,1	red. 1
$ *S*		*none*	accept



1. Shift until top-of-stack is the right end of a handle
2. Pop the right end of the handle & reduce

5 shifts +
9 reduces +
1 accept

Shift-reduce Parsing

Shift reduce parsers are easily built and easily understood

A shift-reduce parser has just four actions

- *Shift* — next word is shifted onto the stack
- *Reduce* — right end of handle is at top of stack
  - Locate left end of handle within the stack
  - Pop handle off stack & push appropriate *lhs*
- *Accept* — stop parsing & report success
- *Error* — call an error reporting/recovery routine

Accept & *Error* are simple

Shift is just a push and a call to the scanner

Reduce takes |*rhs*| pops & 1 push

If handle-finding requires state, put it in the stack

Handle finding is key

- **handle is on stack**
- **finite set of handles**

⇒ **use a DFA !**

More on Shift-Reduce Parsing

Viable prefixes:

The set of prefixes of a right sentential form that can appear on the stack of a Shift-Reduce parser is called Viable prefixes.

Conflicts

"shift/reduce" or "reduce/reduce"

Example:

stmt → **if *expr* then *stmt***

| **if *expr* then *stmt* else *stmt***

| **other (any other statement)**

We can't tell whether it is a handle

Stack

if … then stmt

Input

else …

Shift/ Reduce Conflict

More Conflicts

stmt → id (*parameter-list*)

stmt → *expr* := *expr*

parameter-list → *parameter-list* , *parameter* | *parameter*

parameter → id

expr-list → *expr-list* , *expr* | *expr*

expr → id | id (*expr-list*)

Consider the string A(I,J)

Corresponding token stream is id(id, id)

After three shifts:

Stack = … id(id Input = , id)…

Reduce/Reduce Conflict … what to do?

(it really depends on what is A,
an array? or a procedure?

How the symbol third from the top of stack determines the reduction to be made, even though it is not involved in the reduction. Shift-reduce parsing can utilize information far down in the stack to guide the parse

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:

$E \rightarrow AB$	$E \rightarrow EOE$	$E \rightarrow E+E \mid$
$A \rightarrow a$	$E \rightarrow id$	$E*E \mid$
$B \rightarrow b$	$O \rightarrow +\mid*\mid/$	$E/E \mid id$
not operator grammar	not operator grammar	operator grammar

Precedence Relations

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

$a <\cdot b$	b has higher precedence than a
$a =\cdot b$	b has same precedence as a
$a \cdot> b$	b has lower precedence than a

- 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).

Using Operator-Precedence Relations

- The intention of the precedence relations is to find the handle of a right-sentential form,
  $\lessdot$ with marking the left end,
  $=\cdot$ appearing in the interior of the handle, and
  $\cdot>$ marking the right hand.

- 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).

Using Operator -Precedence Relations

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\hat{}E \mid (E) \mid -E \mid id$

The partial operator-precedence table for this grammar

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

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

$$\$ <\cdot \text{ id } \cdot> + <\cdot \text{ id } \cdot> * <\cdot \text{ id } \cdot> \$$$

To Find The Handles

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

$ <· 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 $
$ $		$ E $

Operator-Precedence Parsing Algorithm

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

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 $\lessdot$ b or a $\doteq$ b) **then** { /* SHIFT */
      push b onto the stack;
      advance p to the next input symbol;
    }
    else if (a $\gtrdot$ b) **then** /* REDUCE */
      repeat pop stack
      until (the top of stack terminal is related by $\lessdot$ to the terminal most recently popped);
    else error();
  }

Operator-Precedence Parsing Algorithm -- Example

stack	***input***	***action***		
$	id+id*id$	$ $\lessdot$ id	shift	
$id	+id*id$	id $\gtrdot$ +	reduce	E $\rightarrow$ id
$	+id*id$	shift		
$+	id*id$	shift		
$+id	*id$	id $\gtrdot$ *	reduce	E $\rightarrow$ id
$+	*id$	shift		
$+*	id$	shift		
$+*id	$	id $\gtrdot$ $	reduce	E $\rightarrow$ id
$+*	$	* $\gtrdot$ $	reduce	E $\rightarrow$ E*E
$+	$	+ $\gtrdot$ $	reduce	E $\rightarrow$ E+E
$	$	accept		

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

How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.

1. If operator θ_1 has higher precedence than operator θ_2,
   ➔ $\theta_1 \cdot\!> \theta_2$ and $\theta_2 <\!\cdot\ \theta_1$

2. If operator θ_1 and operator θ_2 have equal precedence,
   they are left-associative ➔ $\theta_1 \cdot\!> \theta_2$ and $\theta_2 \cdot\!> \theta_1$
   they are right-associative ➔ $\theta_1 <\!\cdot\ \theta_2$ and $\theta_2 <\!\cdot\ \theta_1$

3. For all operators θ, $\theta <\!\cdot$ id, id $\cdot\!> \theta$, $\theta <\!\cdot$ (, (<· θ, $\theta \cdot\!>$),) $\cdot\!> \theta$, $\theta \cdot\!> \$$, and $\$ <\!\cdot\ \theta$

4. Also, let

(=·)	$ <· (	id ·>)	) ·> $
(<· (	$ <· id	id ·> $	) ·>)
(<· id			

Operator-Precedence Relations

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

Handling Unary Minus

- Operator-Precedence parsing cannot handle the unary minus when we also have the binary minus in our grammar.
- The best approach to solve this problem, let the lexical analyzer handle this problem.
  - The lexical analyzer will 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.
- 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

Precedence Functions

- Compilers using operator precedence parsers do not need to store the table of precedence relations.
- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbols a and b.

  f(a) < g(b) whenever a $\lessdot$ b

  f(a) = g(b) whenever a $\doteq$ b

  f(a) > g(b) whenever a $\gtrdot$ b

Algorithm 4.6 **Constructing precedence functions**

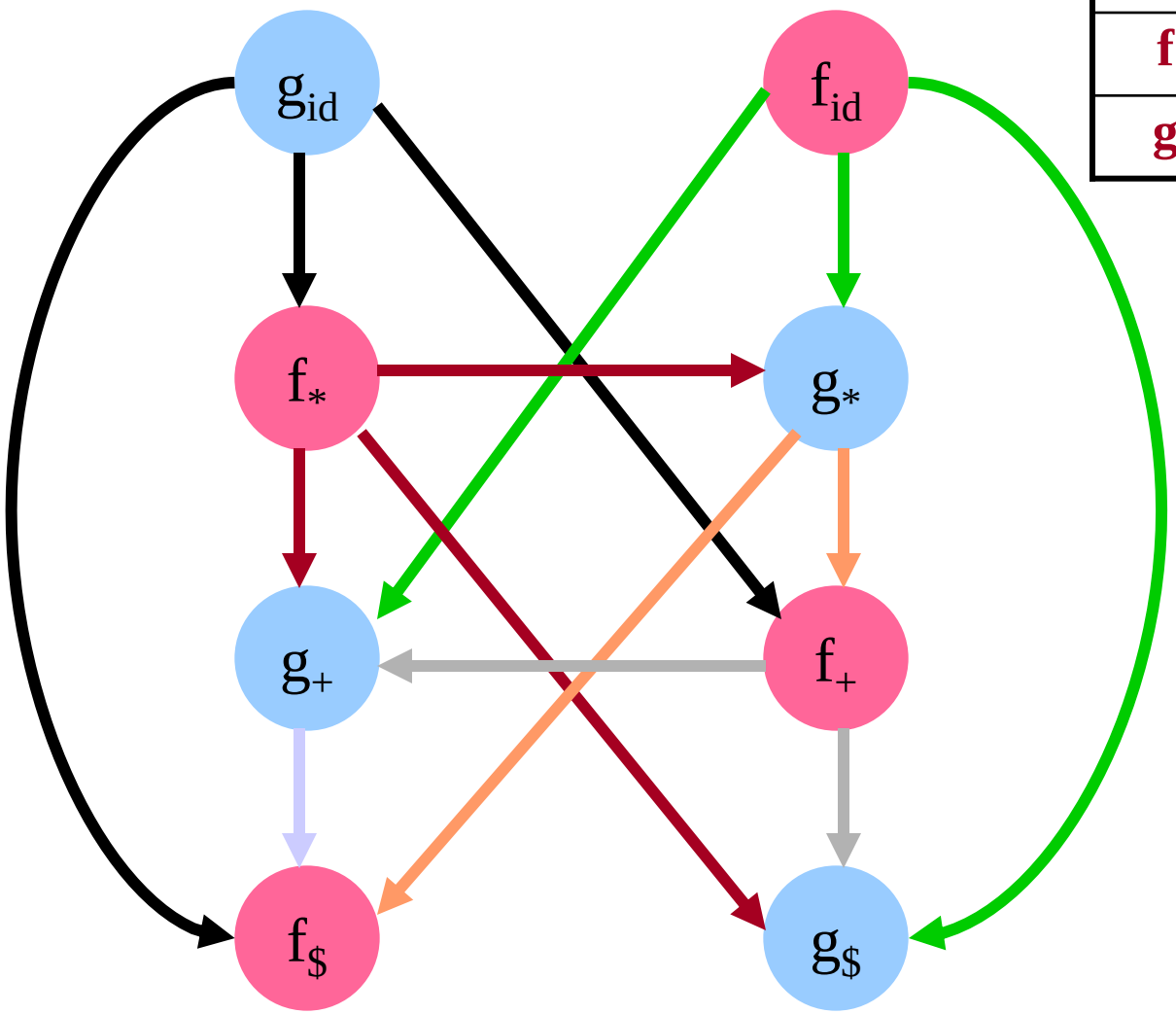
Constructing precedence functions

Method:

1. Create symbols $\mathbf{f_a}$ and $\mathbf{g_b}$ for each **a** that is a terminal or $.
2. Partition the created symbols into as many groups as possible, in such a way that if a =. b, then $\mathbf{f_a}$ and $\mathbf{g_b}$ are in the same group.
3. Create a directed graph whose nodes are the groups found in (2). For any a and b, if a <.b , place an edge from the group of $\mathbf{g_b}$ to the group of $\mathbf{f_a}$. Of a .> b, place an edge from the group of $\mathbf{f_a}$ to that of $\mathbf{g_b}$.
4. If the graph constructed in (3) 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

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



Disadvantages of Operator Precedence Parsing

- **Disadvantages**:
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

- **Advantages**:
  - simple
  - powerful enough for expressions in programming languages

Error Recovery in Operator-Precedence Parsing

Error Cases:

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found (reduction step), but there is no production with this handle as a right side

Error Recovery:

1. Each empty entry is filled with a pointer to an error routine.
2. Decides the popped handle “looks like” which right hand side. And tries to recover from that situation.

Handling Shift/Reduce Errors

When consulting the precedence matrix to decide whether to shift or reduce, we may find that no relation holds between the top stack and the first input symbol.

To recover, we must modify (insert/change)

1. Stack or
2. Input or
3. Both.

We must be careful that we don't get into an infinite loop.

Example

	id	(	)	$
id	**e3**	**e3**	·>	·>
(	<·	<·	=.	**e4**
)	**e3**	**e3**	·>	·>
$	<·	<·	e2	e1

e1: Called when : whole expression is missing

insert **id** onto the input

issue diagnostic: 'missing operand'

e2: Called when : expression begins with a right parenthesis

delete) from the input

issue diagnostic: 'unbalanced right parenthesis'

Example

	id	(	)	$
id	**e3**	**e3**	·>	·>
(	<·	<·	=.	**e4**
)	**e3**	**e3**	·>	·>
$	<·	<·	e2	e1

e3: Called when : id or) is followed by id or (

insert + onto the input

issue diagnostic: 'missing operator'

e4: Called when : expression ends with a left parenthesis

pop (from the stack

issue diagnostic: 'missing right parenthesis'

The End