Compiler course

Chapter 4

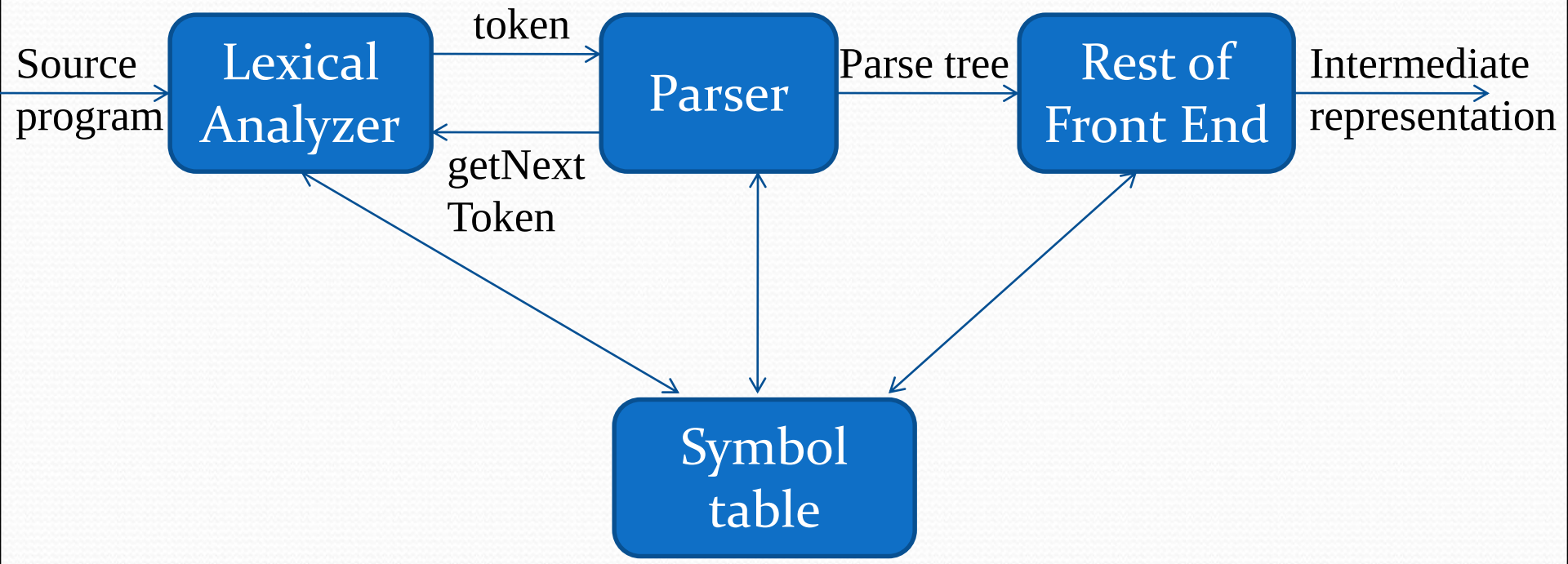
Syntax Analysis

Outline

- Role of parser
- Context free grammars
- Top down parsing
- Bottom up parsing
- Parser generators

- Checking the structure of source program .
- Known as parsing
- 2nd phase of compiler
- Checks the syntax and produced IC code for the representation of source program.
- Why CFG used for syntax checking?

The role of parser



Uses of grammars

E -> E + T | T
T -> T * F | F
F -> (E) | **id**

E -> TE’
E’ -> +TE’ | ε
T -> FT’
T’ -> *FT’ | ε
F -> (E) | **id**

Error handling

- Common programming errors
  - Lexical errors
  - Syntactic errors
  - Semantic errors
- Error handler goals
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct programs

Error recovery:

Panic Mode Recovery

- In this method, successive characters from input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are deli-meters such as ; or }
- Advantage is that its easy to implement and guarantees not to go to infinite loop
- Disadvantage is that a considerable amount of input is skipped without checking it for additional errors

Statement Mode recovery

- In this method, when a parser encounters an error, it performs necessary correction on remaining input so that the rest of input statement allow the parser to parse ahead.
- The correction can be deletion of extra semicolons, replacing comma by semicolon or inserting missing semicolon.
- While performing correction, atmost care should be taken for not going in infinite loop.

Error production

- If user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.
- If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
- Disadvantage is that its difficult to maintain.

Global correction

- The parser examines the whole program and tries to find out the closest match for it which is error free.
- The closest match program has less number of insertions, deletions and changes of tokens to recover from erroneous input.
- Due to high time and space complexity, this method is not implemented practically.

- For Given incorrect input X and grammar G, it will find parse tree for related string y such that no of insertions, deletions and changes of tokens required to transform x into y is as small as possible

Context free grammars

- Terminals
- Nonterminals
- Start symbol
- productions

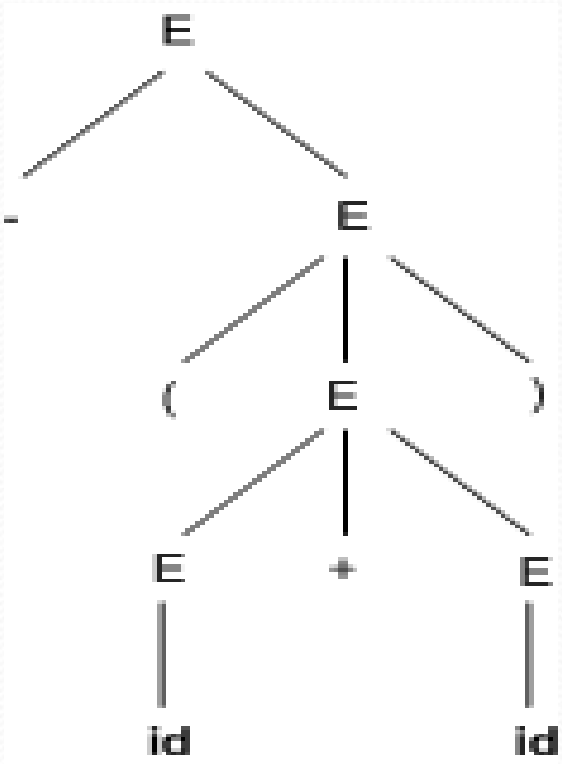
expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term / factor
term -> factor
factor -> (expression)
factor -> **id**

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
  - E -> E + E | E * E | -E | (E) | **id**
  - Derivations for **–(id+id)**
    - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)

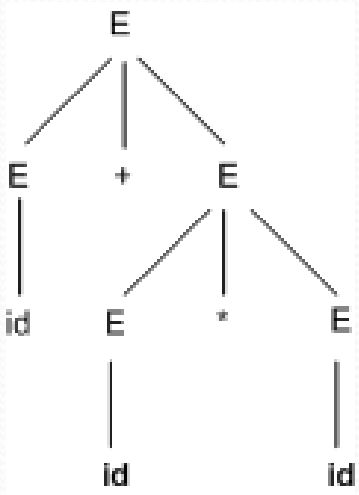
Parse trees

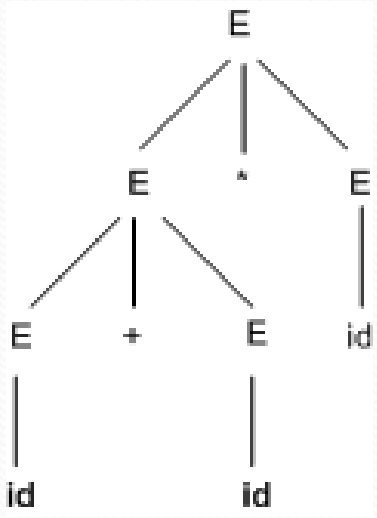
- **-(id+id)**
- E => **-**E => **-**(E) => **-**(E+E) => **-(id**+E)=>**-(id+id)**



Ambiguity

- For some strings there exist more than one parse tree
- Or more than one leftmost derivation
- Or more than one rightmost derivation
- Example: id+id*id





Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation $A =>^{+} A\ \alpha$
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - $A \rightarrow A\ \alpha \mid \beta$
  - We may replace it with
    - $A \rightarrow \beta\ A'$
    - $A' \rightarrow \alpha\ A' \mid \varepsilon$

Eliminate left recursion

1. $A \rightarrow ABd / Aa / a$

   $B \rightarrow Be / b$

2. $S \rightarrow (L) / a$

   $L \rightarrow L , S / S$

3. $A \rightarrow Ba / Aa / c$

   $B \rightarrow Bb / Ab / d$

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
  - Stmt -> **if** expr **then** stmt **else** stmt
  - | **if** expr **then** stmt
- On seeing input **if** it is not clear for the parser which production to use
- We can easily perform left factoring:
  - If we have A-> α β 1 | α β 2 then we replace it with
    - A -> α A'
    - A' -> β 1 | β 2

Left factoring (cont.)

- Example:

  1.S -> I E t S | i E t S e S | a

  E -> b

  2. A → aAB / aBc / aAc

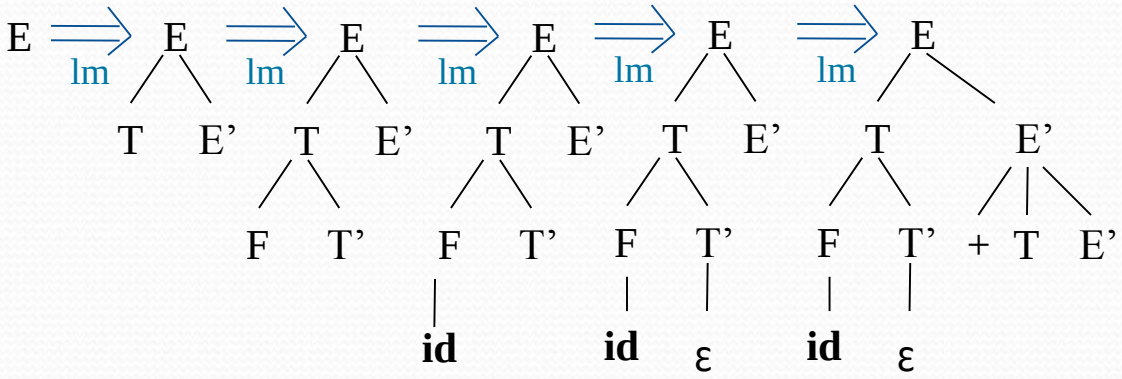
  3. S → bSSaaS / bSSaSb / bSb / a

Top Down Parsing

Introduction

- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example: id+id*id

E -> TE’
E’ -> +TE’ | ε
T -> FT’
T’ -> *FT’ | ε
F -> (E) | **id**



Recursive descent parsing

- Consists of a set of procedures, one for each nonterminal
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A() {
        choose an A-production, A->X1X2..Xk
        for (i=1 to k) {
                if (Xi is a nonterminal
                        call procedure Xi();
                else if (Xi equals the current input symbol a)
                        advance the input to the next symbol;
                else /* an error has occurred */
        }
}
```

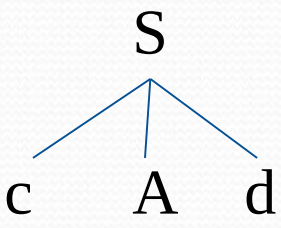
Recursive descent parsing (cont)

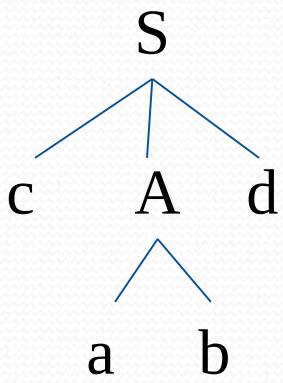
- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking
- In general form it cant choose an A-production easily.
- So we need to try all alternatives
- If one failed the input pointer needs to be reset and another alternative should be tried
- Recursive descent parsers cant be used for left-recursive grammars

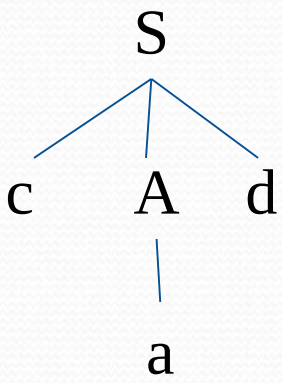
Example

S->cAd
A->ab | a

Input: cad







1. E -> iE’

   E’ -> +iE’|null

2. S -> AB

   A -> c|aB

   B -> d.

To understand first and follow

- Each time a predictive parser makes a decision, it needs to determine which production rule to apply to the leftmost non-terminal in an intermediate form, based on the next terminal

- **FIRST?**
- We saw the need of backtrack in the previous article of on Introduction to Syntax Analysis, which is really a complex process to implement. There can be easier way to sort out this problem:
- If the compiler would have come to know in advance, that what is the “first character of the string produced when a production rule is applied”, and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.

- S -> aAb
- A -> a | <epsilon>

- **FOLLOW?**
  The parser faces one more problem. Let us consider below grammar to understand this problem.
- A -> aBb
- B -> c | ε

First and Follow

- First() is set of terminals that begins strings derived from
- If $\alpha \stackrel{*}{\Rightarrow} \varepsilon$ then is also in First(ε)
- In predictive parsing when we have A-> $\alpha \mid \beta$, if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input
- Follow(A), for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form
  - If we have $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ for some α and β then a is in Follow(A)
- If A can be the rightmost symbol in some sentential form, then $ is in Follow(A)

Computing First

- To compute First(X) for all grammar symbols X, apply following rules until no more terminals or ε can be added to any First set:
  1. If X is a terminal then First(X) = {X}.
  2. If X is a nonterminal and X->Y1Y2…Yk is a production for some k>=1, then place a in First(X) if for some i, a is in First(Yi) and ε is in all of First(Y1),…,First(Yi-1) that is $Y1\ldots Yi\text{-}1 \overset{*}{\Rightarrow} \varepsilon$. if ε is in First(Yj) for j=1,…,k then add ε to First(X).
  3. If X-> ε is a production then add ε to First(X)
- Example! *

Rules for First Sets

1. If X is a terminal then First(X) is just X!
2. If there is a Production X → ε then add ε to first(X)
3. If there is a Production X → Y1Y2..Yk then add first(Y1Y2..Yk) to first(X)
4. First(Y1Y2..Yk) is either
    1. First(Y1) (if First(Y1) doesn't contain ε)
    2. OR (if First(Y1) does contain ε) then First (Y1Y2..Yk) is Everything in First(Y1) <except for ε > as well as everything in First(Y2..Yk)
    3. If First(Y1) First(Y2)..First(Yk) all contain ε then add ε to First(Y1Y2..Yk) as well.

Computing follow

- To compute First(A) for all nonterminals A, apply following rules until nothing can be added to any follow set:
  1. Place $ in Follow(S) where S is the start symbol
  2. If there is a production A-> α B β then everything in First(β) except ε is in Follow(B).
  3. If there is a production A->B or a production A-> α B β where First(β) contains ε, then everything in Follow(A) is in Follow(B)
- Example!

Rules for Follow Sets

1.First put $ (the end of input marker) in Follow(S) (S is the start symbol)

2. If there is a production A → aBb, (where a can be a whole string) then everything in FIRST(b) except for ε is placed in FOLLOW(B).

3. If there is a production A → aB, then everything in FOLLOW(A) is in FOLLOW(B)

4.If there is a production A → aBb, where FIRST(b) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

Examples

1)

S -> aAB|bA|NULL

A-> aAb|NULL

B->bB|c

2)

S->iCtSS’|a

S’-> eS|NULL

C->b

3.

S -> aBDh

B -> cC

C -> bC | NULL

D -> EF

E -> g | NULL

F -> f | NULL

LL(1) Grammars

- Predictive parsers are those recursive descent parsers needing no backtracking
- Grammars for which we can create predictive parsers are called LL(1)
  - The first L means scanning input from left to right
  - The second L means leftmost derivation
  - And 1 stands for using one input symbol for lookahead
- A grammar G is LL(1) if and only if whenever A-> $\alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - For no terminal a do α and β both derive strings beginning with a
  - At most one of α or β can derive empty string
  - If $\alpha \Rightarrow^{*} \varepsilon$ then β does not derive any string beginning with a terminal in Follow(A).

Construction of predictive parsing table

- For each production A-> α in grammar do the following:
  1. For each terminal a in First(α) add A-> in M[A,a]
  2. If ε is in First(α), then for each terminal b in Follow(A) add A-> ε to M[A,b]. If ε is in First(α) and $ is in Follow(A), add A-> ε to M[A,$] as well
- If after performing the above, there is no production in M[A,a] then set M[A,a] to error

Example

E -> TE’
E’ -> +TE’ | ε
T -> FT’
T’ -> *FT’ | ε
F -> (E) | **id**

	First	Follow
F	{(,id}	{+, *,), $}
T	{(,id}	{+,), $}
E	{(,id}	{), $}
E’	{+,ε}	{), $}
T’	{*,ε}	{+,), $}

Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’ -> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> **id**			F -> (E)		

Another example

S -> iEtSS' | a
S' -> eS | ε
E -> b

Non-terminal	Input Symbol					
	a	b	e	i	t	$
S	S -> a			S -> iEtSS'		
S'			S' -> ε S' -> eS			S' -> ε
E		E -> b				

Bottom-up Parsing

OPERATOR PRECEDENCE PARSING

- One of the bottom up parser
- It is mainly use to define the mathematical operators.

- What is operator grammar?
- A grammar which is use to define generally mathematical operations it is called as operator grammar.
- With some restrictions.?

Operator Grammar

- No Ɛ-transition.
- No two adjacent non-terminals.

  Eg.

  E → E op E | id

  op → + | *

  The above grammar is not an operator grammar but:

  E → E + E | E* E | id

A-> ABA/a

B->bAb/b

Operator Precedence

- If a has higher precedence over b; a .> b
- If a has lower precedence over b; a <. b
- If a and b have equal precedence; a =. b

Note:

- id has higher precedence than any other symbol
- $ has lowest precedence.
- if two operators have equal precedence, then we check the **Associativity** of that particular operator.

Precedence Table

	id	+	*	$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Example: w= $id + id * id$

$<.id.>+<.id.>*<.id.>$

BASIC PRINCIPLE

- Scan input string left to right, try to detect .> and put a pointer on its location.
- Now scan backwards till reaching <.
- String between <. And .> is our handle.
- Replace handle by the head of the respective production.
- REPEAT until reaching start symbol.

Operator-Precedence 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();
}
```

EXAMPLE

STACK	INPUT	ACTION/REMARK
$	id + id * id$	$ <. Id
$ id	+ id * id$	id >. +
$	+ id * id$	$ <. +
$ +	id * id$	+ <. Id
$ + id	* id$	id .> *
$ +	* id$	+ <. *
$ + *	id$	* <. Id
$ + * id	$	id .> $
$ + *	$	* .> $
$ +	$	+ .> $
$	$	accept

PRECEDENCE FUNCTIONS

- Operator precedence parsers use **precedence functions** that map terminal symbols to integers.

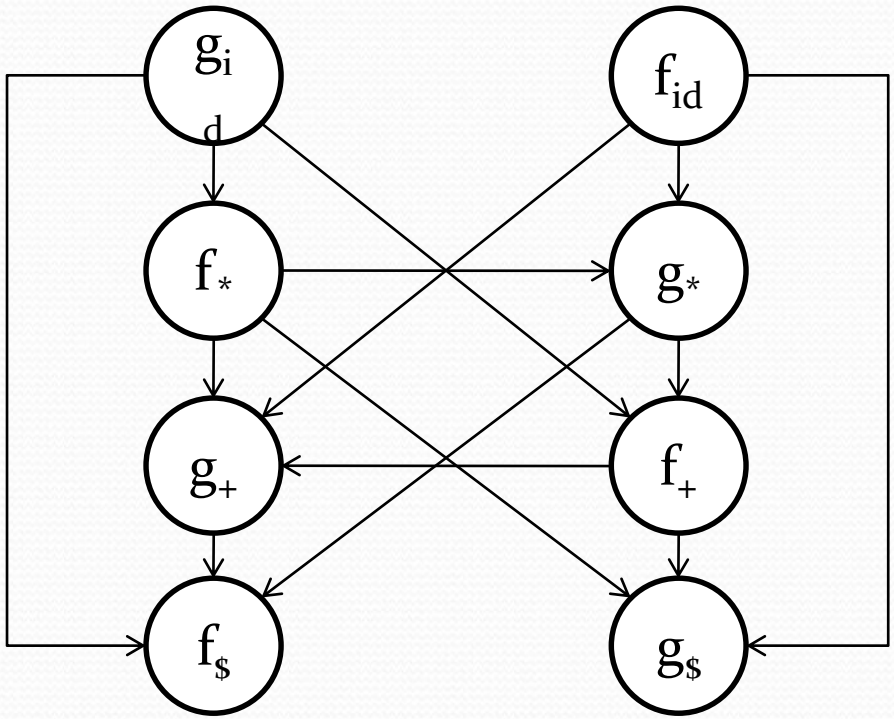
Algorithm for Constructing Precedence Functions

1. Create functions f_a for each grammar terminal a and for the end of string symbol.
2. Partition the symbols in groups so that f_a and g_b are in the same group if $a =\cdot b$ (there can be symbols in the same group even if they are not connected by this relation).
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of g_b to the group of f_a if $a <\cdot b$, otherwise if $a \cdot> b$ place an edge from the group of f_a to that of g_b.
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.

- Consider the following table:

	id	+	*	$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

- Resulting graph:



- From the previous graph we extract the following precedence functions:

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