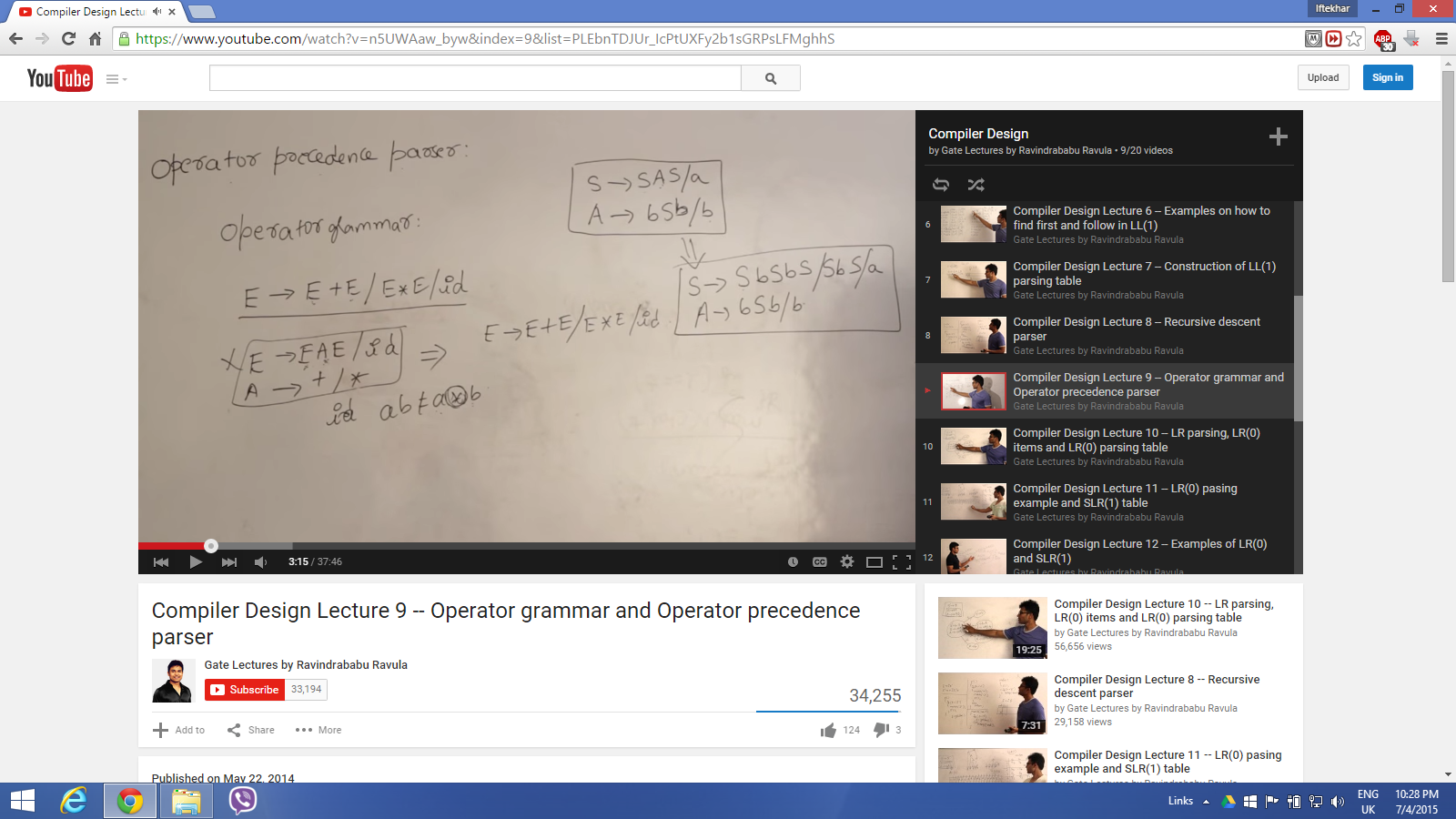
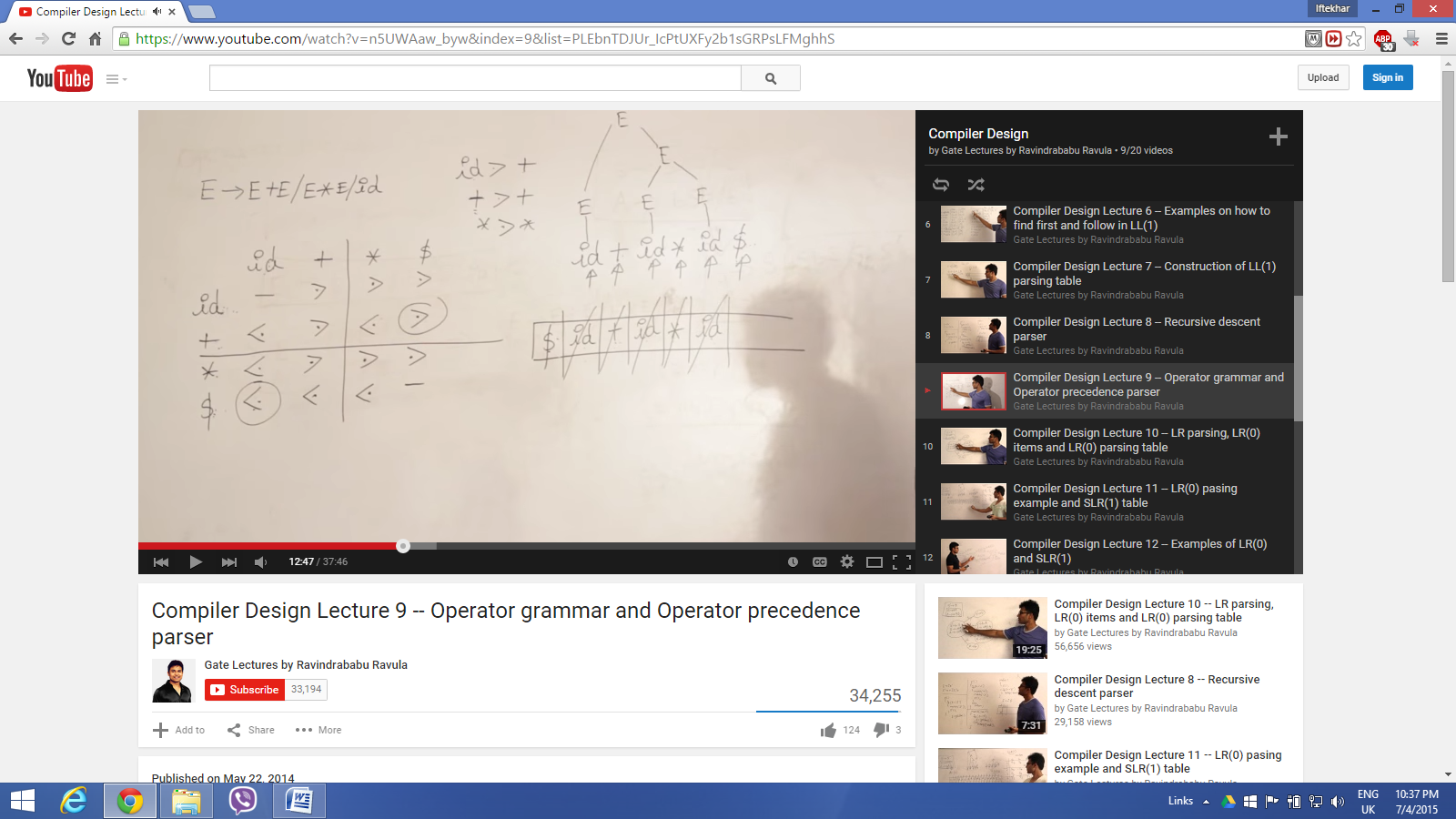
<https://www.youtube.com/watch?v=n5UWAaw_byw&index=9&list=PLEbnTDJUr_IcPtUXFy2b1sGRPsLFMghhS>

* If two variables are adjacent to each other in that case it's not an operator precedence grammar.
* From the following two grammars right grammar is not operator precedence grammar but the left top one is. The left below one is not an operator precedence grammar



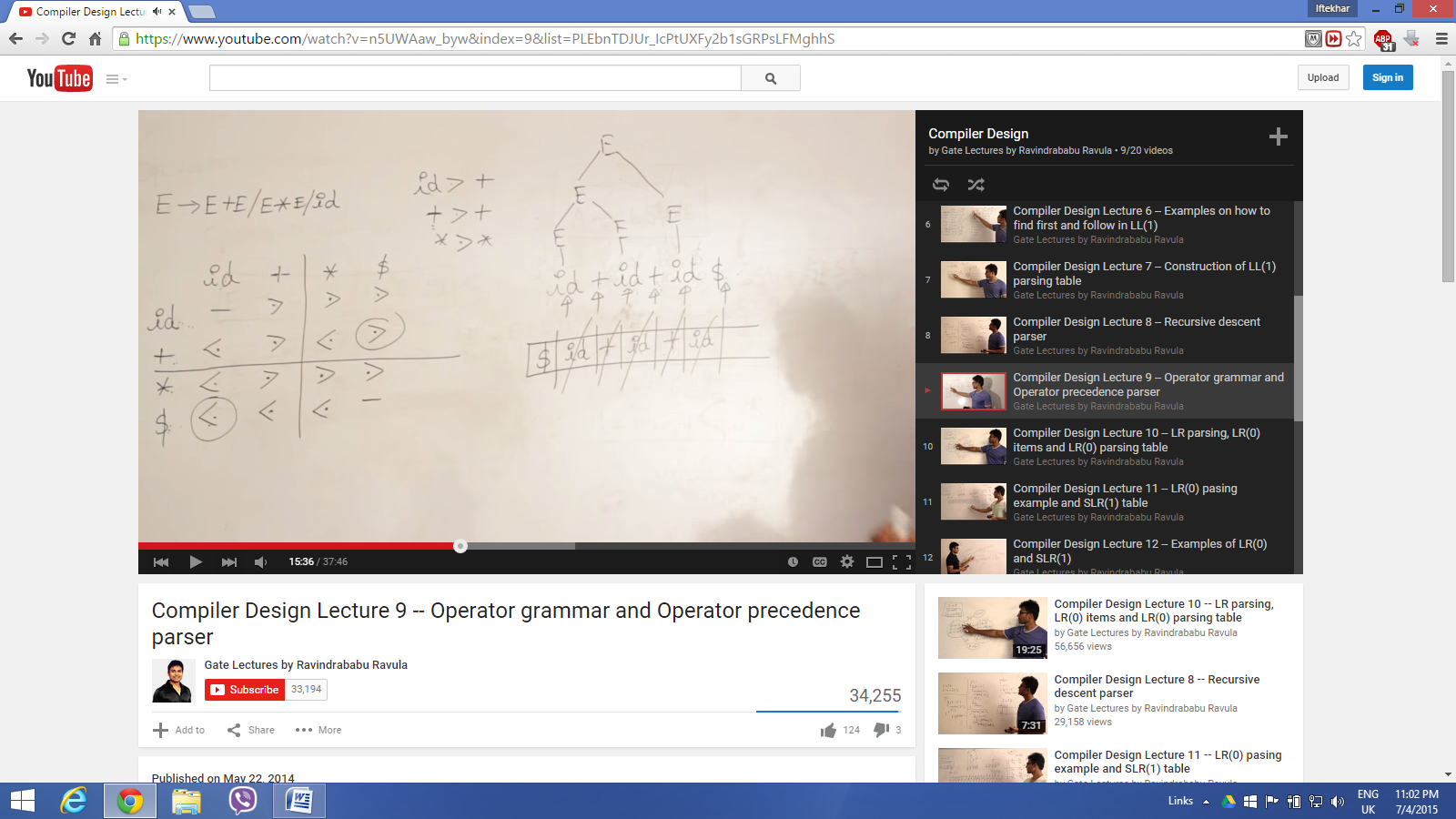


1. Identifier has the highest precedence, then operator and at the end dollar sign. That is how the table is created.
2. Always stack will be going to start with dollar sign,

**How the strings are matched**

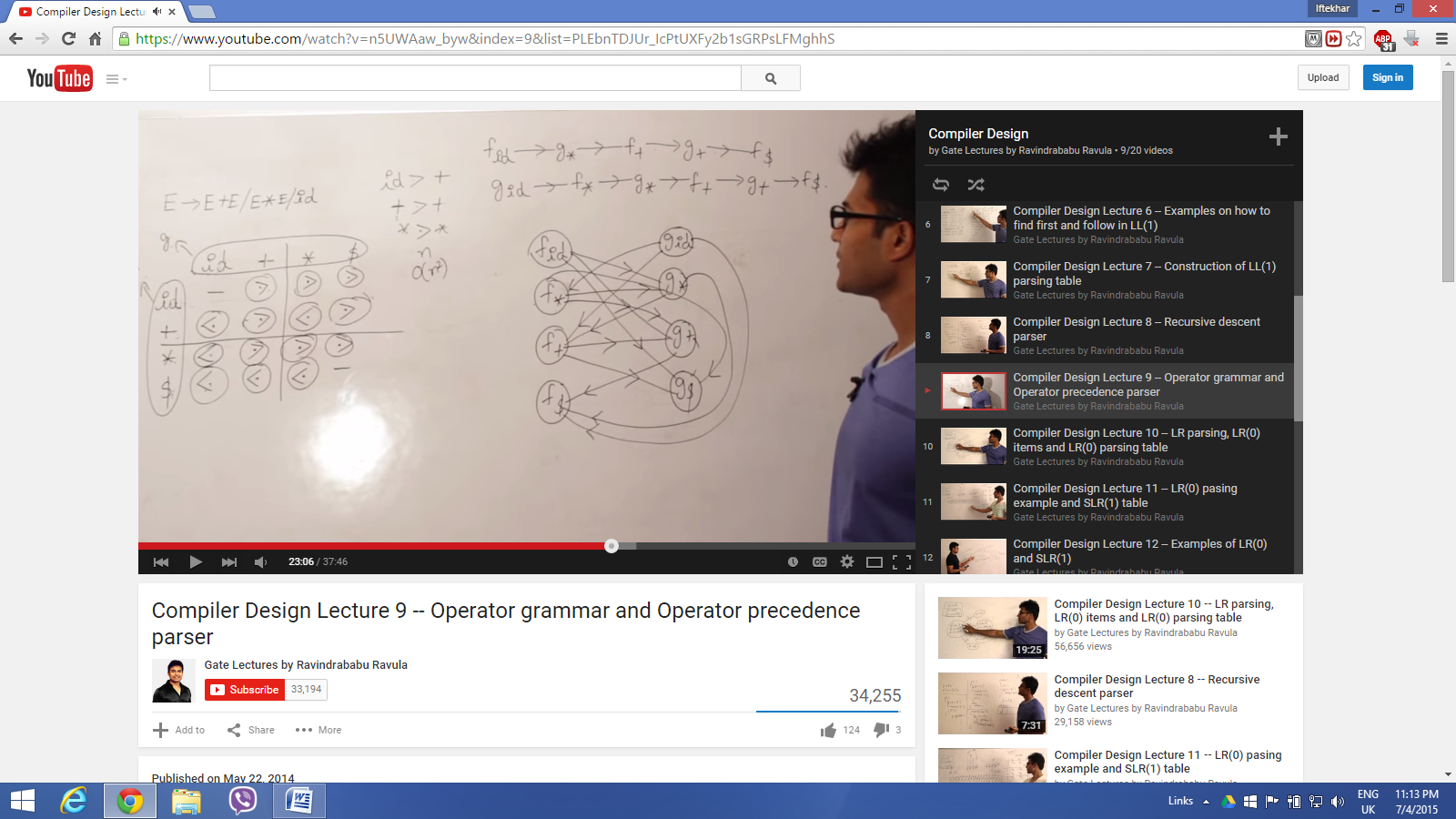
* Less than push and greater than pop

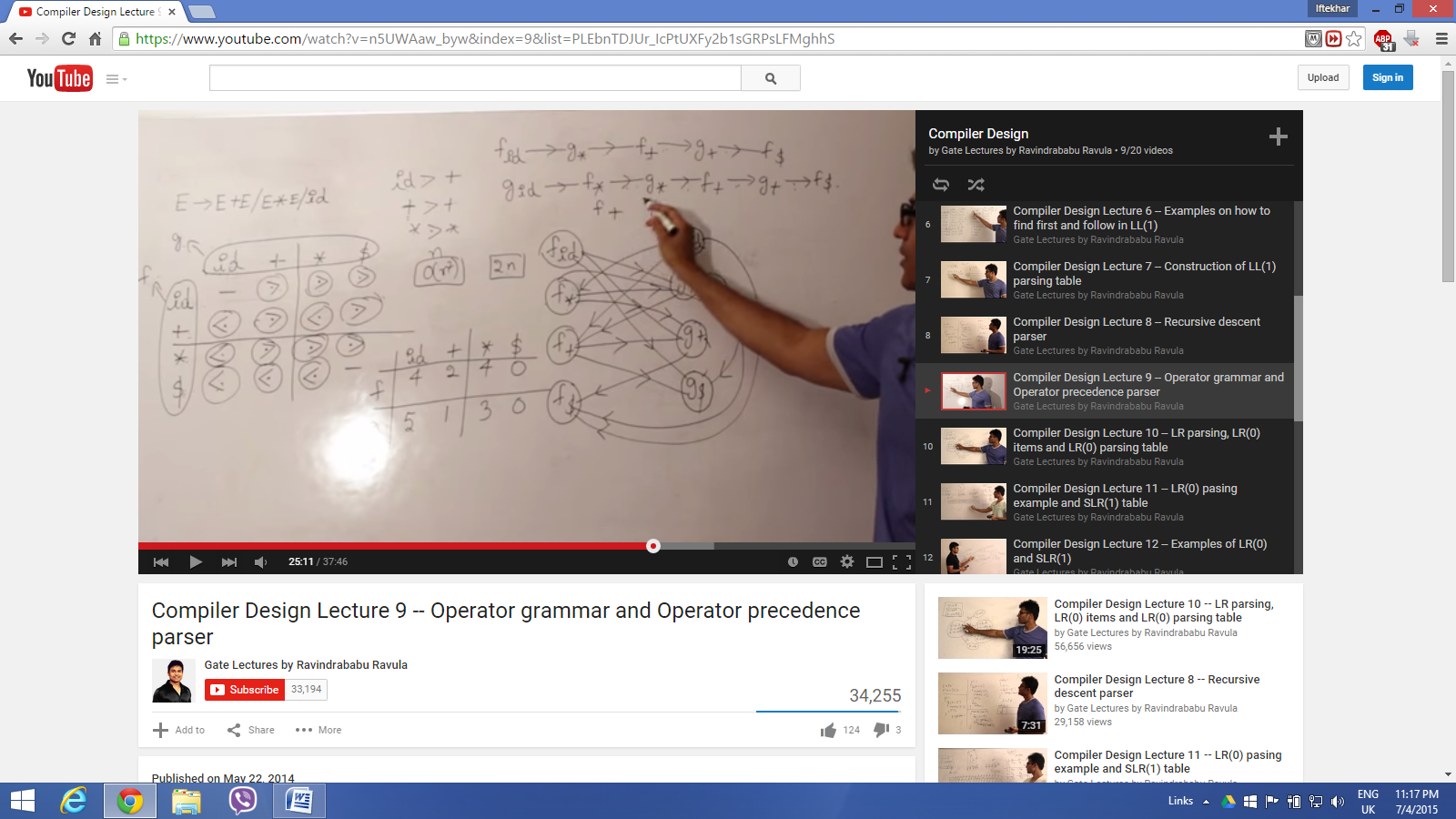
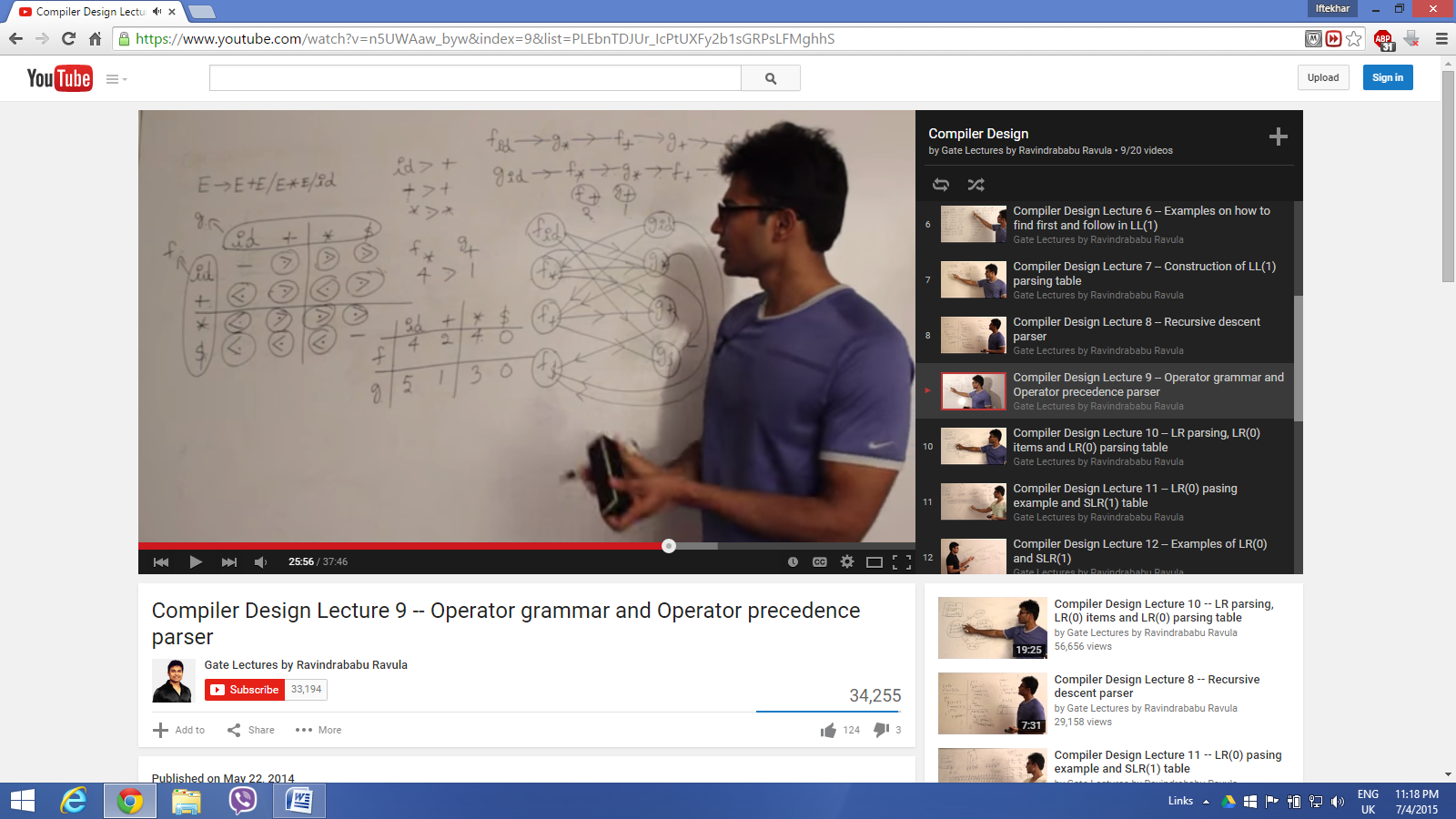
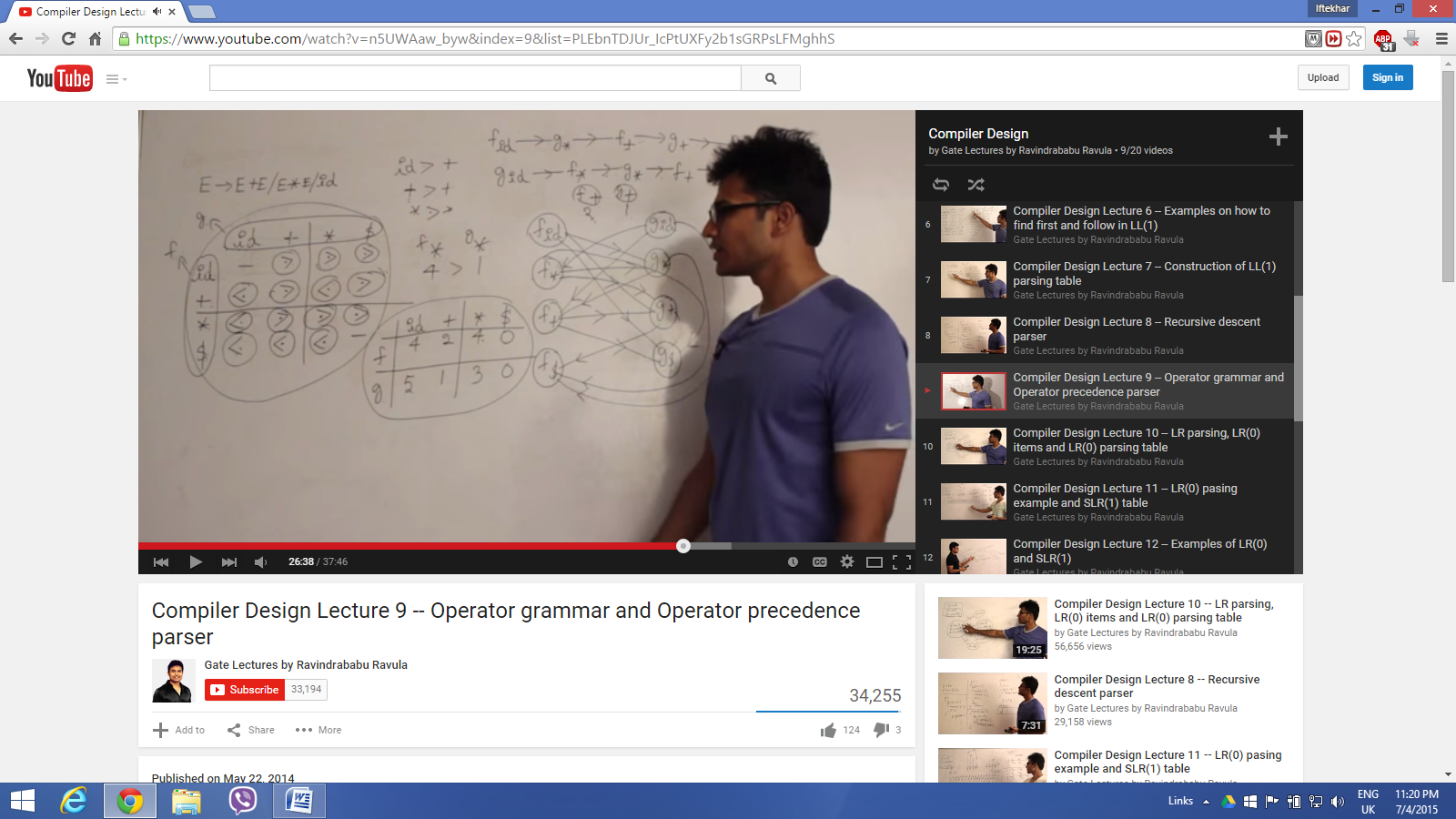
1. first compare dollar and id therefore push id symbol,
2. then id and + comparison, in where '+' showed less than hence push +,
3. If do popping anything in that case we will cross the symbol and draw a parsing tree leg using a corresponding terminal ,
4. At the end we will see that all the id is popped and +, \* remains on the stack then + will be compared with dollar and similarly \* will be compared with dollar as well and due to higher precedence going to pop and eventually we will get the whole parse tree

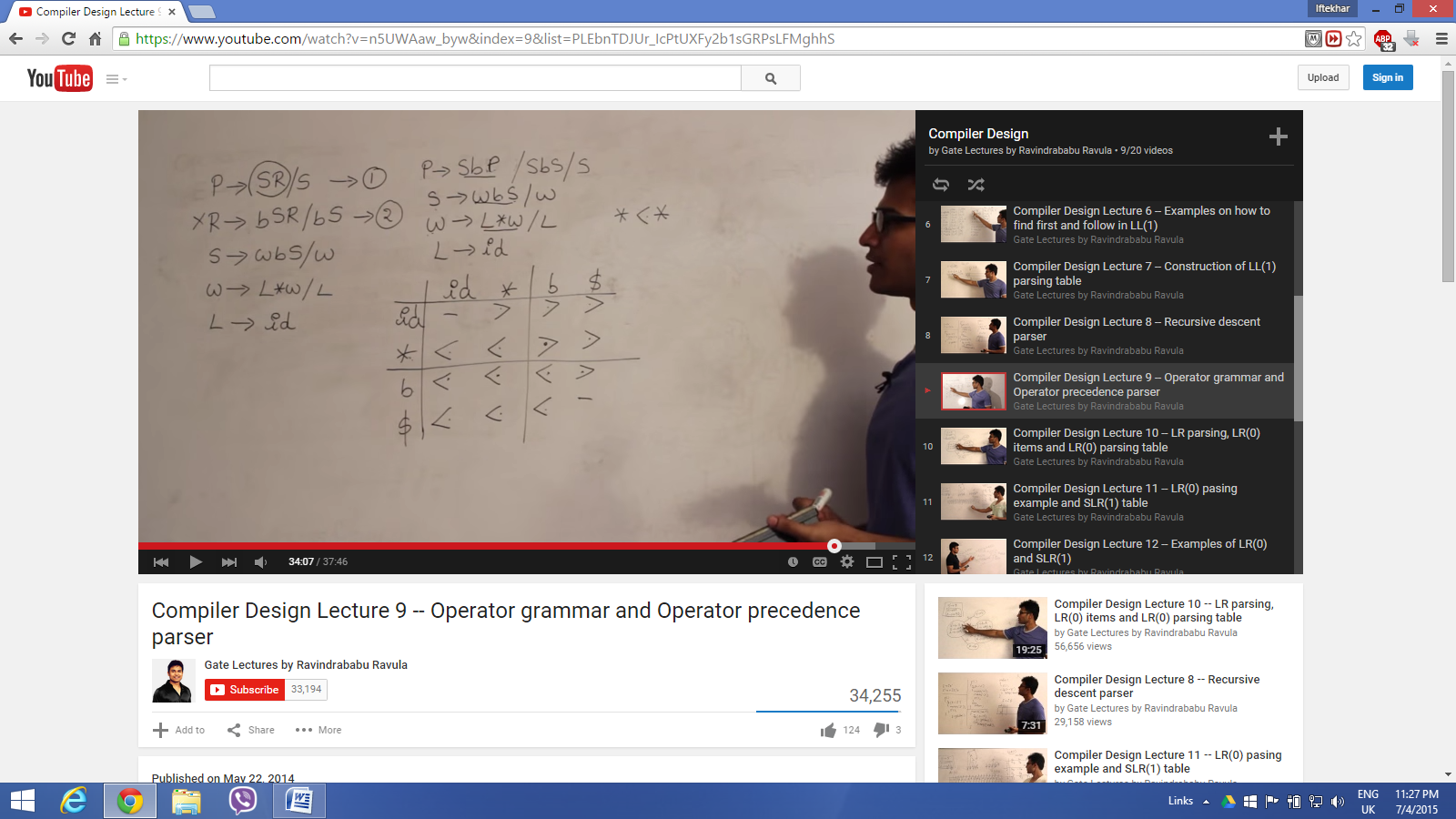


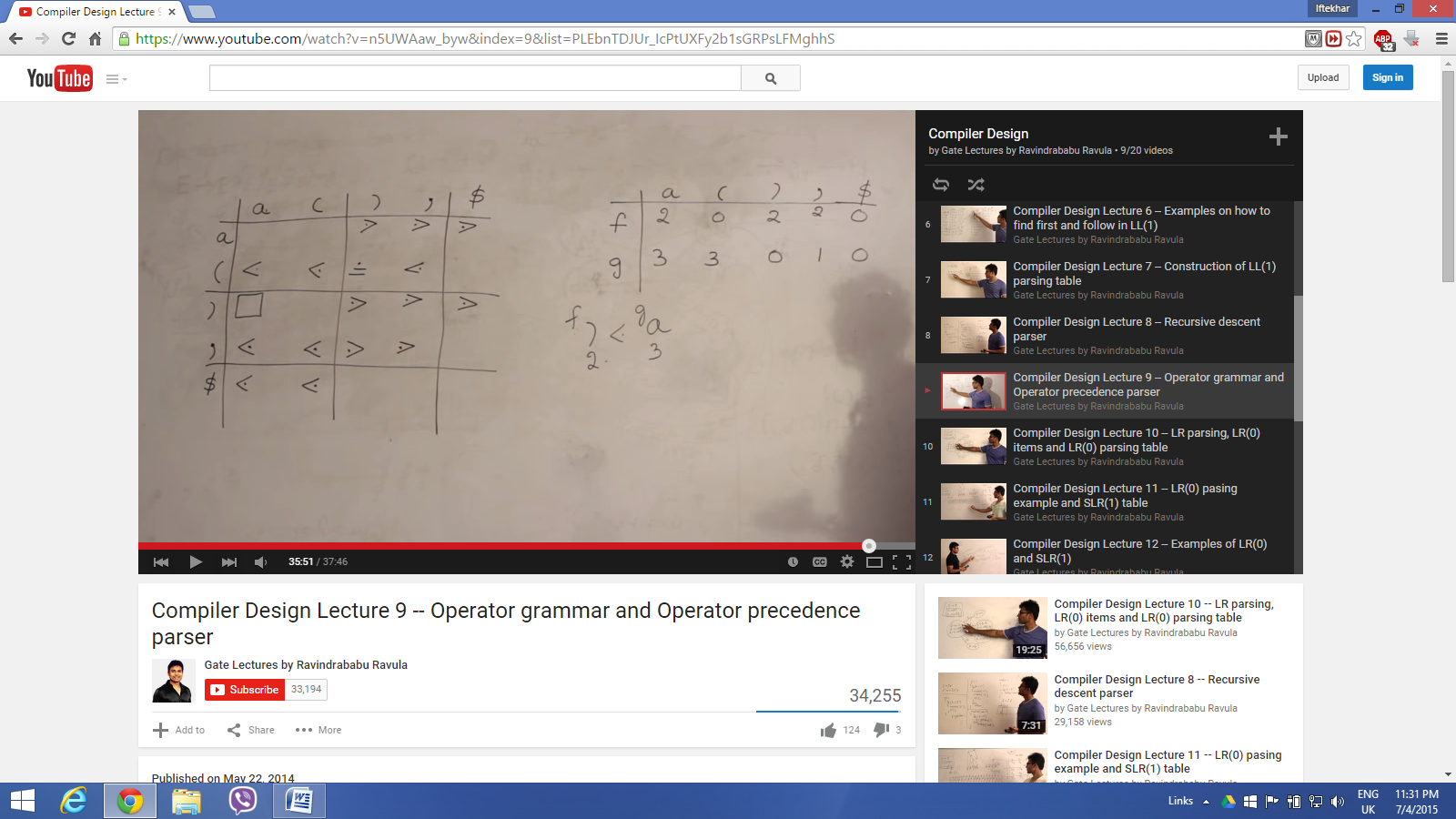
The above screen shot is another example**, one of the interesting issue with this parser is this will going to work even though it will be an ambiguous grammar.**

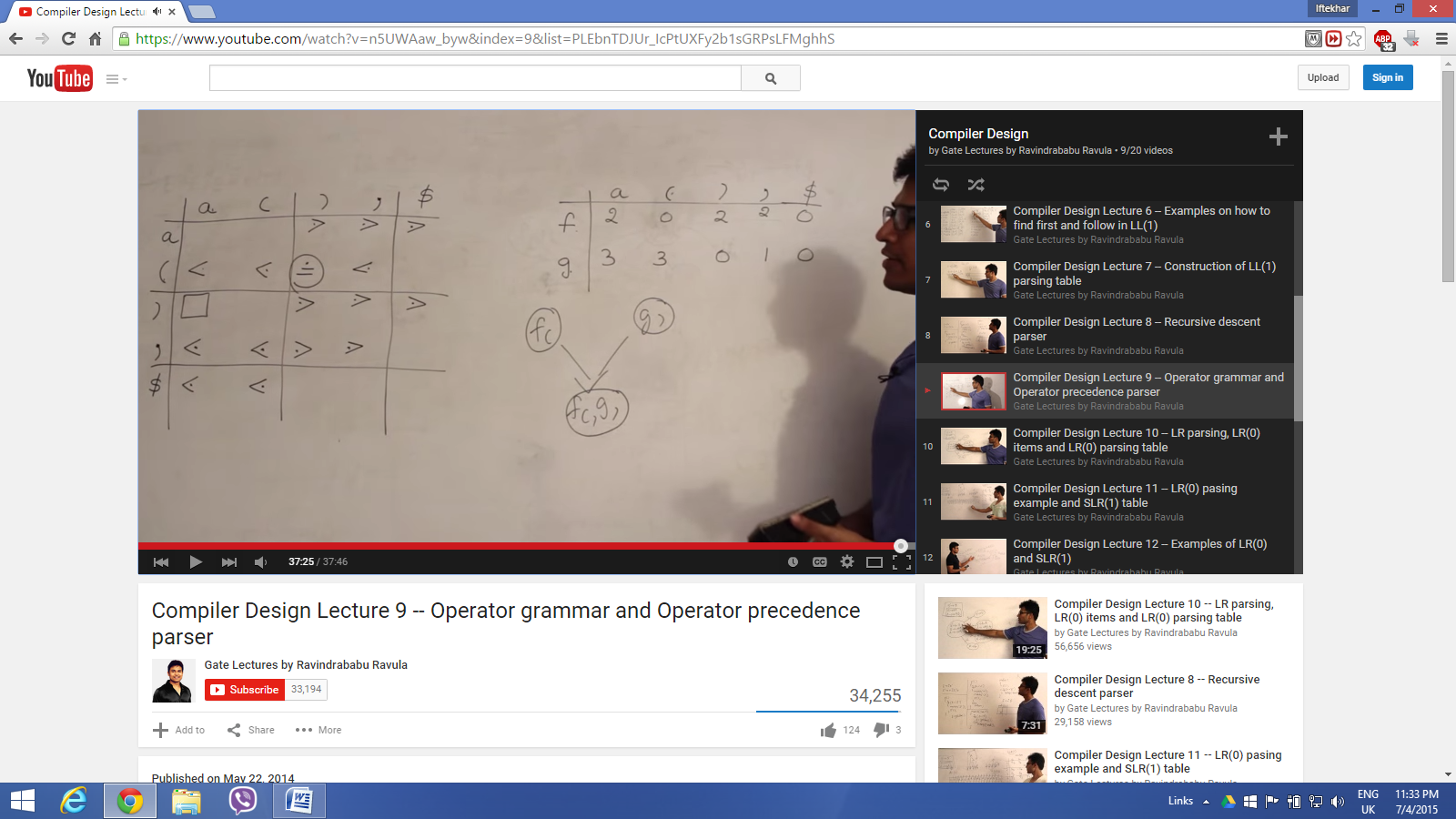
**But the disadvantage is for n number of operator we need n^2 number of cells.**











**Operator Precedence Parsing**

**1. Precedence Relations**

Bottom-up parsers for a large class of context-free grammars

can be easily developed using *operator grammars*.

*Operator grammars* have the property that no production right side

is empty or has two adjacent nonterminals. This property enables the

implementation of efficient *operator-precedence parsers*. These

parser rely on the following three precedence relations:

|  |  |
| --- | --- |
| Relation | Meaning |
| *a*  <· *b* | *a* yields precedence to *b* |
| *a* =· *b* | *a* has the same precedence as *b* |
| *a* ·> *b* | *a* takes precedence over *b* |

These operator precedence relations allow to delimit the handles

in the right sentential forms: <· marks the left end, =· appears in

the interior of the handle, and ·> marks the right end.

Let assume that between the symbols *a*i and *a*i+1 there is exactly

one precedence relation. Suppose that $ is the end of the string.

Then for all terminals we can write: $ <· *b* and *b* ·> $. If we

remove all nonterminals and place the correct precedence relation:

<·, =·, ·> between the remaining terminals, there remain strings

that can be analyzed by easily developed parser.

For example, the following operator precedence relations can

be introduced for simple expressions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **id** | + | \* | $ |
| **id** |  | ·> | ·> | ·> |
| + | <· | ·> | <· | ·> |
| \* | <· | ·> | ·> | ·> |
| $ | <· | <· | <· | ·> |

*Example*: The input string:

**id**1 + **id**2 **\* id**3

after inserting precedence relations becomes

$ <· **id**1 ·> + <· **id**2·> **\*** <· **id**3 ·> $

Having precedence relations allows to identify handles as follows:

- scan the string from left until seeing ·>

- scan backwards the string from right to left until seeing <·

- everything between the two relations <· and ·> forms the handle

Note that not the entire sentential form is scanned to find the handle.

**2. Operator Precedence Parsing Algorithm**

*Initialize*: Set *ip* to point to the first symbol of *w*$

*Repeat*: Let *X* be the top stack symbol, and *a* the symbol pointed to by *ip*

**if** $ is on the top of the stack and ip points to $  **then return**

**else**

Let *a* be the top terminal on the stack, and *b* the symbol pointed to by *ip*

**if** *a* <· *b* **or** *a* =· *b* **then**

push *b* onto the stack

advance *ip* to the next input symbol

**else if** *a* ·> *b* **then**

**repeat**

pop the stack

**until** the top stack terminal is related by <·

to the terminal most recently popped

**else** *error()*

**end**

**3. Making Operator Precedence Relations**

The operator precedence parsers usually do not store the precedence

table with the relations, rather they are implemented in a special way.

Operator precedence parsers use precedence functions that map

terminal symbols to integers, and so the precedence relations

between the symbols are implemented by numerical comparison.

Not every table of precedence relations has precedence functions

but in practice for most grammars such functions can be designed.

**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* =· *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* <· *b*, otherwise if *a* ·> *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.

*Example*: Consider the above table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **id** | + | \* | $ |
| **id** |  | ·> | ·> | ·> |
| + | <· | ·> | <· | ·> |
| \* | <· | ·> | ·> | ·> |
| $ | <· | <· | <· | ·> |

Using the algorithm leads to the following graph:

fig6-1.wmf

from which we extract the following precedence functions:

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