Lecture Scribe

Operator Precedence Functions and Overview of LR-Parsing Techniques

Date:04/09/13

Author:Sikhar Patranabis

1. Building the operator precedence parsing table:

We have seen that the operator precedence parser requires a parsing table that it must refer to while parsing an input string.

We consider the following grammar:

E-> E+E | E*E | id

For this grammar the precedence table looks something like this:

	id	*****	**+**	**$**
id		$\dot{>}$	$\dot{>}$	$\dot{>}$
*****	$<\dot{}$	$\dot{>}$	$\dot{>}$	$\dot{>}$
+	$<\dot{}$	$<\dot{}$	$\dot{>}$	$\dot{>}$
$	$<\dot{}$	$<\dot{}$	$<\dot{}$	

The table size is n*n where n is the number of terminal symbols in the grammar.

The storage space however can be efficiently managed if we can build an efficient data structure to represent adequately the precedence relationships between the terminals.

For this we can construct two functions *f* and *g* where the following hold:

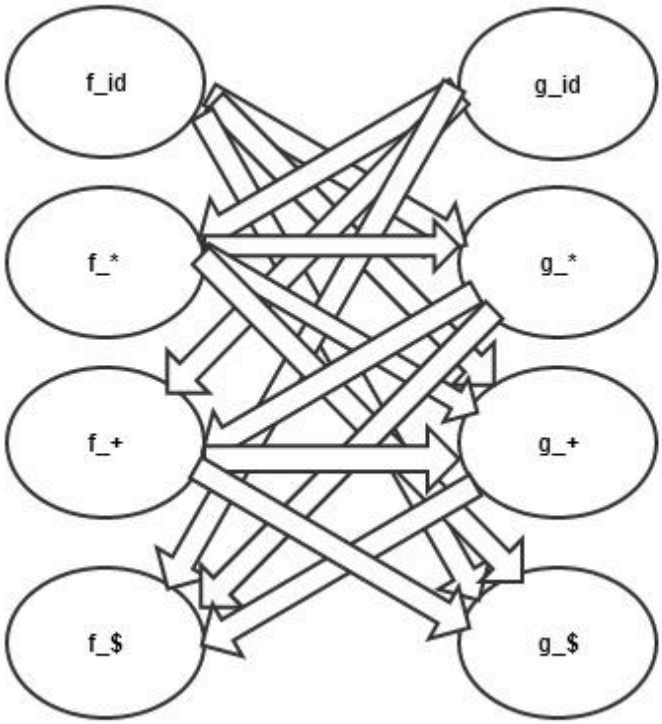
1. If $a\dot{>}b$ then $f(a)>g(b)$
2. If $a<\dot{}b$ then $f(a)<g(b)$

3. If a$\doteq$b then f(a)=g(b)

To construct these functions we create a directed graph with nodes f_i and g_i where i is the i^{th} terminal, obeying the following rules:

- If a$\doteq$b then f_a and f_b are merged, and g_a and g_b are merged
- If a$\dot{>}$b then a directed edge is introduced from f_a to g_b
- If a$\dot{<}$b then a directed edge is introduced to f_a from g_b

Following these rules, the precedence graph for the above mentioned precedence table looks something like this:



Now we can easily construct a table for the f and g values of each terminal by denoting f(a) as the longest possible path in the graph starting from f_a such that each node on this path has lower precedence than its immediate preceding node. Similarly for g(a) as well.

In this case we get the following table:

	f	**g**
id	4	5

*	4	3
+	2	1
$	0	0

Clearly now the storage space is only n*2 where n is the number of terminals. Thus this construction of a directed graph with the nodes representing functional entities allows saving a lot of space.

Now using this table as reference the operator precedence parser can parse any operator string fed to it as input.

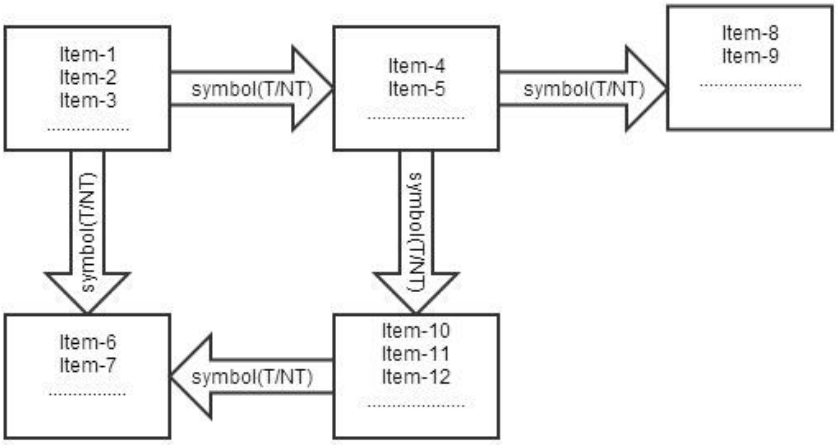
2. LR Parsers:

The operator precedence parser that we have discussed so far uses the shift/reduce parsing paradigm to parse the input string in bottom-up fashion. However, owing to its inherent limitations in terms of the range of languages that an operator precedence parser may parse, it is not really a commercially viable option for use as a standard parser. Most programming language constructs do not have such well defined precedence relations and the applicability of an operator-precedence parser is limited to operator expressions only. So in order to be able to parse more complicated grammars, we use an LR parser.

The term LR comes from left-scanning and reverse of rightmost derivation i.e. the parser scans the input string from left to right and builds the parse tree such that it traces the reverse of a rightmost derivation, i.e. at any point of time the stack contains a valid prefix of a right sentential form.

The additional information that the LR parser uses is a finite state machine (FSM) like structure where each state represents a collection of items. As in any FSM this machine too has a transition function that takes a state and a symbol as input (terminal/non-terminal) and generates a new state as output. This transition function is given a special name- **goto()**.

A schematic diagram of a FSM for an LR parser is shown here:



Basic Algorithm of shift/reduce parsing using transition table:

- We maintain 2 stacks instead of one- one containing states and the other containing symbols read from the input string.
- We check the state at the top of the stack and the current input symbol in the string
- We search for an appropriate item in the current state that will allow a transition from the current state to some other state for that particular input symbol. If we have such a production we move onto the next state and push that new state onto the state stack, and the input symbol is shifted to the symbol stack.
- If we do not get the shift item, we look for some other item in the current state that may allow us to reduce a possible handle at the top of the symbol stack using an appropriate production rule. Then we perform the reduction and back track to the previous state.
- If we reach an accepting configuration, we accept.
- If there is no possible action, we report an error.

Construction of the FSM:

In order to construct the FSM, we must perform certain steps, including:

1. Augmentation of the grammar
2. Identification of items from productions

Augmentation of grammar involves adding a new start symbol (a non-terminal obviously) to the grammar along with a new production where the new start symbol derives the old one.

For example, if S be the current start symbol of the grammar G, then the augmented grammar G' has a new start symbol S' and a new production S'=>S.

This new start symbol is necessary to uniquely characterize an accepting configuration, as we shall see in subsequent lectures.

Derivation of items is another trick that is applied to validate the portion of the string read so far as possible expansion of a viable prefix of some production, i.e. to identify whether in future we can use some production to reduce a portion of the contents of the stack.

Let us take an example:

A=>XYZ be a production in the augmented grammar G'.

We derive the following items from it:

a) A=>.XYZ
b) A=>X.YZ
c) A=>XY.Z
d) A=>XYZ.

Now we analyze the implication of the 4 items. Imagine my input string is $w_1w_2…w_kaw_{k+1}….w_{n.}$ Also assume that 'a' is the current input symbol pointed to by the pointer. Then item (a) implies that we have not yet identified any prefix of the production but expect to do so from the next character onwards. Item (b) implies we have matched prefix derived from X in the scanned part and expect to start scanning a substring derivable from Y. Item (c) implies we have matched X and Y and expect to match Z. Item (d) implies we have matched an entire substring derivable from A and we are ready to reduce the appropriate handle using this production, provided my current input symbol is in FOLLOW (A).

This gives us an idea of how to resolve a shift-reduce conflict while parsing. If the current state has got items like (a), (b) or (c), we may shift in the hope of reaching (d) after scanning a finite number of symbols. On the other hand if we have already reached a state where an item like (d) exists, we can reduce using an appropriate production.

However to be able to perform these tasks without any conflict, we need to design an appropriate action table which we will study in subsequent lectures.