1 Definitions of terms

Describe chunk, yield, take

2 Data Structures

2.1 Partition

Given input string is divided in to multiple chunks. This data structure describes an individual chunk. This says that data beginning from index 'firtIdx' till 'lastIdx' is owned by this chunk. It also holds that data of whether this chunk has any yield and has any take. 'lock' says whether this chuck is being processed by any worker or not. Initially lock is set to false.

- `firstIdx`
- `lastIdx`
- `hasYield`
- `hasTake`
- `lock`
- `nxt`

Constructor accepts firstIdx, lastIdx and initialises the object. 'nxt' is used as a reference to the connecting object. If 'nxt' is -1 that implies there is no connecting object.

2.2 Symbol

This holds an input string in inputStr. It's respective precedence type is taken from the operator precedence grammar matrix and stored in 'precedenceSymbol'. It only contains the following characters ('<', '>', '=', '|'). 'nxt' holds the reference to the next element it is pointing to. If 'nxt' is -1, that implies this is the last element in the sequence.

('<', '>', '=') is used for non-terminals and ('|') is used for terminal strings. In the algorithm we refer '<' as yield and '>' as a take.

- `inputStr`
- `precedenceSymbol`
- `nxt`

3 Required Variables

In this section we refer all the variables with its type definition and data it stores that are used in the algorithm.

3.1 inputSequence

Type: Vector

Description

Input given by the user is processed and for each input string respective Symbol object is created. Each element of the array represents the Symbol object added in relevant order. For each symbol initially 'nxt' is initialised with its next element's index. Only exceptional case is the last element which is initialised 'nxt' with -1.

3.2 partitionTable

Type: Vector

Description

Each element of the array is an reference to a Partition object no two elements in the partitionTable are responsible for the same index in inputSequence. This observation helps us to process the array in parallel since no two elements have a dependency. We use smart pointers to handle the references, so that even in case of a mistake in handling the dangling pointer the memory is properly cleaned. The class structure and kind of data it stores is defined below.

4 Functions Used

The purpose of this section is to define all the functions that are used in the algorithm.

4.1 Solver

Accepts: idx Returns: nothing

Description

'idx' refers to the index of the partitionTable. Solver contains the logics of a reducer which is applied on the idx-th index of the 'partitionTable'.

Algorithm 1: solver

4.2 merger

Description

Accepts: idx1, idx2

Returns: nothing

idx1 and idx2 represents two indices of partitionTable which are theoretical consecutive indices. idx1-th index of partitionTable is modified accordingly and passed to the solver. Before passing it to the solver, lock is applied so that no other process can pass it to solver.

Pseudo Code

input : idx1, idx2

1 partitionTable[$idx1$].$lastIdx$ ← partitionTable[$idx2$].$lastIdx$;
2 partitionTable[$idx1$].nxt ← partitionTable[$idx2$].nxt ;
3 **if** partitionTable *[idx1].hasYeilds and* partitionTable *[idx2].hasTakes* **then**
4 | partitionTable[$idx1$].$lock$ ← $True$;
5 | `async solver (idx1)` ;
6 | `// Solver is either an async or parallel process.`

Algorithm 2: merger

4.3 basicSetup

Description

Accepts: user input

Returns: nothing

User input is taken and 'inputSequence' is created out of it. Later inputSequence is broken in to multiple chunks. Each chunk represents a continuous subsequence of inputSequence. No two chunks have an overlap of data. Every chunk is independent to each other. Partition object is created from each chunk and every object is appended to the partitionTable.

Pseudo Code

```
input : User input
1  n ← length of userInput;
2  inputSequence ← initialise array to size n.
3  breakLength ← getChunkLength(n);
4  // getChunkLength returns a good value is chosen to break the
      precedence sequence into different chucks where the length of any
      two different chunks doesn't have a huge difference.
5  for i ← 1 to n do
6  |   adds new Symbol object returned by createSymbol(userInput [i]) to
   |    inputSequence;
7  |   // Can use async process to speed up.
8  end
9  // await until all async calls are successfully executed.
10 idx1 ← 1;
11 idx2 ← breakLength;
12 while idx2 ≤ n do
13 |   adds new Partition object returned by createPartition(idx1, idx2) to
   |    partitionTable;
14 |   // Can use async process to speed up.
15 |   idx1 ← idx1+ breakLength;
16 |   idx2 ← min(idx2 + breakLength, n) ;
17 end
```

Algorithm 3: basicSetup

4.4 workAssigner

Description

Accepts: user input

Returns: nothing

'partitionTable' produced by the basicSetup' is used in this routine. This tries to merge every two consecutively available chunks and reduces that newly created chunk if possible. This process goes until only one chunk is left and no more reductions are possible.

In case of any error or lack of possibility of reduction, appropriate area of the error in the input sequence is alerted.

Pseudo Code

```
input  : inputSequence and partitionTable are provided as input arguments
output : All the partitioned chunks are parsed appropriately and merged
         together until there is nothing else to parse.
1  n ← number of chunks (length of partitionTable);
2  cnt ← number of chunks (length of partitionTable);
3  // Represents number of available chunks.  If number of available
      chunks is 1, it implies that all the processing is completed.
4  while cnt ≠ 1 do
5  |  idx ← 1 ;
6  |  nxtIdx ← partitionTable[idx].nxt ;
7  |  while idx ≠ −1 do
8  |  |  if partitionTable [idx].lock is true then
9  |  |  |  do nothing ;
10 |  |  // This implies that the chunk is still being processed.
11 |  |  else if partitionTable [nxtIdx].lock is true then
12 |  |  |  do nothing ;
13 |  |  // This implies that the nxtIdx-th chunk is still being
          processed and merge can happen only with consecutive chunks
          (Technically idx and nxtIdx are consecutive) .
14 |  |  else
15 |  |  |  mergeChunks(idx, nxtIdx);
16 |  |  |  n ← n - 1;
17 |  |  idx ← partitionTable[idx].nxt ;
18 |  end
19 end
```

Algorithm 4: workAssigner