**Bottom Up Parsing Handout**

Compiled by: Nomair A. Naeem  
Additional Material by: Adriel Dean-Hall and Brad Lushman  
This handout is intended to accompany material covered during lectures and is not considered a replacement for lectures. Dissemination of this handout (including posting on a website) is explicitly prohibited unless permission is obtained. Please report any errors to nanaeem@uwaterloo.ca.

1 Introduction

In bottom up parsing we start from the input string $w$ and create a reverse derivation to the start symbol $S$  
\[ w \leftarrow \alpha_k \leftarrow \alpha_{k-1} \leftarrow \ldots \leftarrow \alpha_1 \leftarrow S \]

2 Example illustrating bottom-up parsing

Using the augmented grammar:

```
S' → ⊢ S ⊣  
S → AyB  
A → ab  
A → cd  
B → z  
B → wx
```

INPUT $s$ is: \( ⊢ a b y w x \ ⊣ \)

We will use the stack to store partially reduced info read so far:

At each step, the algorithm has two choices:

**SHIFT:** consume the next input symbol and push it on the stack  
**REDUCE:** POP the RHS of a rule off the stack and push its LHS
<table>
<thead>
<tr>
<th>Consumed Input</th>
<th>Remaining Input</th>
<th>Stack</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>l- abywx -l</td>
<td></td>
<td>shift l-</td>
</tr>
<tr>
<td>l- abywx -l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l- a bywx -l</td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>l- ab ywx -l</td>
<td>b a l-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l- ab ywx -l</td>
<td>A l-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l- aby wx -l</td>
<td>y A l-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l- abyw x -l</td>
<td>w y A l-</td>
<td></td>
<td>shift x</td>
</tr>
<tr>
<td>l- abywx -l</td>
<td>x w y A l-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l- abywx -l</td>
<td>B y A l-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l- abywx -l</td>
<td>S l-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l- abywz -l</td>
<td>ε l-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l- abywz -l</td>
<td>ε S'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Check if contents on the stack represent some RHS of a rule if not shift next

SHIFTING a

Again check if we have RHS on stack
shift b

Check Again. This time we have ab on TOS
Can reduce using A -> ab
Pop b and a
Push A

Check if contents on the stack represent some RHS of a rule if not shift next
shift y

Check if we have a RHS on TOS.
We have wx on TOS
Can reduce using B -> wx
Pop x and w
Push B

Check if we have a RHS on TOS.
We have AyB on TOS
Can reduce using S -> AyB
Pop B y and A
Push S

Check if we have a RHS on TOS.
We DON'T shift l-

Check if we have a RHS on TOS.
We have l- S -l on TOS
Can reduce using S' -> l- S -l
Pop l- S l-
Push S'
The accepting condition is reached when stack contains $S'$ and remaining input is $\epsilon$

Since we only shift if we cannot reduce, the key challenge is to decide when to reduce.

We use the notion of items which can be viewed as tracking what portion of the RHS of a rule is on the stack.

**USING THE SIMPLER GRAMMAR**

\[
S' \rightarrow \epsilon \ E \ | \\
E \rightarrow E + T \\
E \rightarrow T \\
T \rightarrow \text{id}
\]

ITEM: An item is a production with a bookmark (represented by a dot) somewhere between the symbols in the RHS.

- e.g. $E \rightarrow \cdot E + T$ (a FRESH item)
- e.g. $E \rightarrow E + T \cdot$ (a REDUCIBLE item)

### IDEA:
- To decide on an action (shift/reduce) to perform, run the contents of the stack through the LR(0) DFA and see which state we end up in.
- If we end up in a state which has a reducible item we will reduce, if not we will shift the next symbol from the unconsumed portion of the input.
EXAMPLE RUN: Parse: \( \vdash \text{id} + \text{id} + \text{id} \) 

<table>
<thead>
<tr>
<th>Symbols Stack</th>
<th>Read Input</th>
<th>Unread Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \epsilon )</td>
<td>( \text{id} )</td>
<td>( \text{id} + \text{id} + \text{id} - )</td>
<td>Pass the contents of the stack through the DFA and see what state we end up in. Since stack is empty we are in state 1.</td>
</tr>
<tr>
<td>( \text{id} )</td>
<td>( \text{id} )</td>
<td>( \text{id} + \text{id} + \text{id} - )</td>
<td>Again run the DFA with current stack contents. We end up in state 2.</td>
</tr>
<tr>
<td>( \text{id} )</td>
<td>( \text{id} )</td>
<td>( \text{id} + \text{id} + \text{id} - )</td>
<td>Pass contents through DFA. We reach: State 6.</td>
</tr>
<tr>
<td>( + )</td>
<td>( \text{id} )</td>
<td>( \text{id} + \text{id} - )</td>
<td>Pass contents through DFA. We reach: State 5.</td>
</tr>
<tr>
<td>\text{And so on}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.1 Making the algorithm efficient
The algorithm discussed above is inefficient, because after each action we pass the entire contents of the stack through the DFA.

We can keep track of the state using a separate (or same) stack.

The key is to ensure that the Top of the State Stack always contains the state we would end up in if we ran the symbol stack contents through the DFA.
<table>
<thead>
<tr>
<th>Symbols Stack</th>
<th>State Stack</th>
<th>Read Input</th>
<th>Unread Input</th>
<th>Action</th>
</tr>
</thead>
</table>
|               | 1           | ε          | I- id + id + id - | * Check Top of State Stack  
|               |             |            |              | * State 1 has no reducible item  
|               |             |            |              | * Shift next symbol AND SHIFT CORRESPONDING STATE on state stack |
| I-            | 2           | id         | I- id + id + id - | Top of State stack is state 2  
|               | 1           |            |              | No reducible item  
|               |             |            |              | Shift next symbol and push corresponding state to state stack |
| id            | 6           | I- id      | + id + id - | * Top of state stack is 6  
|               | 2           |            |              | * It has a reducible item  
|               | 1           |            |              | * Symbol Stack: Reduce by popping id and pushing T  
|               |             |            |              | * State Stack: Pop equal number of state off the state stack and push corresponding state |
| T             | 5           | I- id      | + id + id - | * Top of state stack is 5  
|               | 2           |            |              | * It has a reducible item  
|               | 1           |            |              | * Symbol Stack: reduce by popping T and pushing E  
|               |             |            |              | * State Stack: pop 5 and push 3 |
| E             | 3           | I- id      | + id + id - | * Top of state stack is 3  
|               | 2           |            |              | * Has no reducible item  
|               | 1           |            |              | * Shift + on symbol stack  
|               |             |            |              | * Shift corresponding state to state stack |
| +             | 7           | I- id      | id + id - | * Top of state stack is 7  
|               | 3           |            |              | * Has no reducible item  
|               | 2           |            |              | * Shift id on symbol stack  
|               | 1           |            |              | * Shift corresponding state to state stack |
| id            | 5           | I- id      | id + id - | Top of state stack is 6  
|               | 7           |            |              | Reduce |
| +             | 8           | I- id      | id + id - | Top of state stack is 8  
|               | 7           |            |              | Reduce |
| E             | 3           | I- id      | id + id - | * Top of state stack is 3  
|               | 2           |            |              | * Shift + on symbol stack  
|               | 1           |            |              | * Shift corresponding state to state stack |
| +             | 7           | I- id      | id + id - | Top of state stack is 7  
|               | 3           |            |              | Has no reducible item  
|               | 2           |            |              | Shift id on symbol stack  
|               | 1           |            |              | shift corresponding state to state stack |
| id            | 6           | I- id      | id + id + id - | Top of state stack is 6  
|               | 7           |            |              | Reduce |
| +             | 8           | I- id      | id + id + id - | Top of state stack is 8  
|               | 7           |            |              | Reduce |
| E             | 3           | I- id      | id + id - | * Top of state stack is 3  
|               | 2           |            |              | * Has no reducible item  
|               | 1           |            |              | * Shift - on symbol stack  
|               |             |            |              | * Shift corresponding state to state stack |
| -             | 4           | I- id      | id + id - | Top of state stack is 4  
|               | 3           |            |              | Reduce |
| E             | 5           |             |            |     |
|               | 4           |             |            |     |
|               | 3           |             |            |     |
|               | 2           |             |            |     |
|               | 1           |             |            |     |
2.2 Summary

The method described above is called LR parsing

- Left to Right scan
- Right most derivation

The DFA we used did not look at the next input symbol. Hence the exact method is LR(0)

For WLP4 you will be using SLR(1) / LR(1) parsing i.e. you will need to look at the next input symbol in order to make the shift/reduce decisions

Bottom Up Parsing Invariant: symbol stack + unread input = α_

2.3 Possible Errors

What if a DFA state looks like this:

Do we try to shift the next character (as suggested by A → α ⋆ c β) or do we reduce by B → γ (as suggested by B → γ ⋆). We cannot decide: this is called a shift-reduce conflict (error).

What if a DFA state looks like this:

Do we reduce by A → α or do we reduce by B → β? We cannot decide and this is a reduce-reduce conflict

If any reducible item occurs in a state in which it is NOT alone, then this is a shift-reduce conflict and the grammar is not LR(0)
Suppose input starts with ⊢ \text{id} ..... 

Then, once we shift \text{id}, we will be in state 6. Reducing would lead us to state 5 which has a shift-reduce conflict i.e. Should we reduce using \text{E} \rightarrow \text{T} or wait for a + symbol

The answer depends on what is next.
If the input is \text{⊢ \text{id} +} then YES we should reduce using \text{E} \rightarrow \text{T}
However, if the input is \text{⊢ \text{id} + ...} then, we should shift +

2.5 SLR(1)

- We can add a lookahead to the automaton to do this.
  - For each, reducible item \text{A} \rightarrow \alpha \circ attach the Follow(A) to the item.
  - Recall: Follow(A) = \{ b \mid S' \Rightarrow^* \alpha \text{Ab}\beta \text{ for some } \alpha, \beta\}
  - i.e. starting from the START SYMBOL if we can get to a situation where we have b after A then b is in the follow set of A
  - Essentially, what the follow set is checking is that if we do reduce using \text{A} \rightarrow \alpha and we know that the next symbol is in the follow set, we know there is at least some derivation still possible.

For the example
- Follow(E) = \{ + \}
- Follow(T) = \{ +, \text{⊣} \}

The example had a shift-reduce conflict. We can extend the LR(0) DFA by using follow sets:

\[
\begin{align*}
\text{E} & \rightarrow \text{T} \\
\text{E} & \rightarrow \text{T} \; \text{∙} \; \text{E}
\end{align*}
\]

becomes

\[
\begin{align*}
\text{E} & \rightarrow \text{T} \\
\text{E} & \rightarrow \text{T} \; : \text{⊣}
\end{align*}
\]

- For each reducible item in the LR(0) DFA, add an indication that we will reduce only if the next input symbol is in the FOLLOW set

We read such a state as: if we reach this state and the next input symbol is in the follow set (in this case \{\text{⊣}\}) only then will we reduce otherwise we will shift.

- Notice that the conflict is resolved.
- This construction is called an SLR(1) parser: Simplified LR(1) parser and can resolve many conflicts that arise in an LR(0) parser.

3 LR(1) parsing

- LR(1) parsers are strictly more powerful than LR(0) and SLR(1) parsers.
- Use the next input symbol to make the decision rather than the weaker condition that the next input symbol should be in the follow set.
- ADVANTAGE: parse more grammars
- DISADVANTAGE: complex automaton (expensive to store)
4 Algorithm

The SLR(1), LALR(1) and LR(1) parsing algorithms are the same. The only difference is the DFA used by the algorithm.

LR(1) algorithm (concrete, linear time, input DFA($\Sigma,Q,q_0,\text{Trans},A$))

\[
\begin{align*}
\text{symStack}.\text{push} \vdash \\
\text{stateStack}.\text{push} \text{Trans}[q_0, \vdash] \\
\text{for each symbol } a \text{ in input } \vdash \text{ from left to right} \\
\quad \text{while } \text{Reduce}[\text{stateStack}.\text{top}, a] \text{ is some production } A \rightarrow \gamma \text{ do} \\
\qquad \text{symStack}.\text{pop} \text{ symbols in } \gamma \text{ (right end first)} \\
\qquad \text{stateStack}.\text{pop} \text{ |$\gamma$| states} \\
\qquad \text{symStack}.\text{push} A \\
\qquad \text{stateStack}.\text{push} \text{Trans}[\text{stateStack}.\text{top}, A] \\
\quad \text{end while} \\
\text{symStack}.\text{push} a \\
\text{reject if } \text{Trans}[\text{stateStack}.\text{top}, a] \text{ is undefined (ERROR)} \\
\text{stateStack}.\text{push} \text{Trans}[\text{stateStack}.\text{top}, a] \\
\text{end for} \\
\text{accept (symStack is necessarily $\vdash S$ $\vdash$)}
\end{align*}
\]

A key thing to notice is that the Reduce function looks at the top of the state stack. If it contains reducible items, the choice to reduce is made if the reduce is indicated when the next symbol is $a$.

Notice also how the symbol stack and state stack are always kept in sync, any updates to the symbol stack lead to a corresponding update to the state stack.

5 Hints for the last question of assignment 7

- We have seen that the LR(1) parsing algorithm will give us a reversed rightmost derivation. In A7P4 you will implement this algorithm perhaps using the pseudocode given above.

- In A7P5, you can use the same implementation this time using the lr(1) machine for WLP4, also provided (aren’t we nice!). This will give you a reversed rightmost derivation for the input WLP4 program.

- The challenge is that the required output (.wlp4i format) needs a leftmost derivation whereas you have a rightmost derivation. Note that wlp4i is more than just a print of the leftmost derivation since it also requires you to print terminals (and their lexemes).
• But doing this is fairly straightforward if you think about it for a second. Using the reversed rightmost derivation you can create a parse tree. Recall the property that a derivation uniquely specifies a parse tree. Therefore, since you have a rightmost derivation for the input you should be able to create a parse tree for the input.

• Now: recall the property that a parse tree has a unique leftmost derivation. Therefore, once you have created the parse tree (using the rightmost derivation) you can now traverse this tree to generate a leftmost derivation.

Look at the sample code given in CFGrl (Racket/C/C++ versions are available). This code has functions to create a tree from a rightmost derivation and traversal of a tree to get a leftmost derivation. What the code does NOT have is a way to interleave the leftmost derivation output with information about the terminals you are encountering (these are the leaf nodes in the tree). You will have to implement this functionality on your own.