1 Bottom-up Parsing

1.1 Definitions

- LR(1) stands for \textit{Left-to-right} scan of input, \textit{Right-canonical} derivations produced, \textit{1 symbol of lookahead}. The S in SLR(1) stands for \textit{Simplified}.
- An LR(1) parser is a \textit{bottom-up} parser; it begins with the input string and replaces right hand sides of rules with left hand sides until the start symbol is produced.
- A grammar that can be parsed using the LR(1) algorithm is called an LR(1) grammar.
- An \textit{item} is a production with a dot (\textbullet) somewhere on the right-hand side, and represents a partially-completed rule.
- A \textit{shift-reduce conflict} occurs when a state in the LR machine has both completed and partially completed rules with overlapping follow sets.
- A \textit{reduce-reduce conflict} occurs when a state in the LR machine has two different completed rules with overlapping follow sets.

1.2 The SLR(1) Parsing Algorithm

\begin{verbatim}
push q0
for each a in ⊢ input ⊣
    while (Reduce[stack.top, a] = A → γ)
        pop 2*|γ| times
        state ← stack.top
        push A
        push Trans[state, A]
        state ← stack.top
        if (Trans[state, a] = ERROR) reject
        push a
    push Trans[state, a]
accept
\end{verbatim}

How would you modify the above algorithm to construct a parse tree while parsing?
1.3 Bottom-up Parsing Using SLR(1)

Consider the following context-free grammar $G_3$ and its corresponding SLR(1) shift/reduce machine, given in both bubble diagram and table form:

1.3.1 $G_3$

$$
\begin{align*}
S' &\rightarrow \vdash S \dashv \\
S &\rightarrow Sab \quad (1) \\
S &\rightarrow XY \quad (2) \\
X &\rightarrow pX \quad (3) \\
X &\rightarrow \epsilon \quad (4) \\
Y &\rightarrow q \quad (5) \\
Y &\rightarrow \epsilon \quad (6)
\end{align*}
$$

1.3.2 $G_3$ SLR(1) Machine Table

<table>
<thead>
<tr>
<th></th>
<th>⊢</th>
<th>shift 1</th>
<th>2</th>
<th>a</th>
<th>reduce 2</th>
<th>5</th>
<th>a</th>
<th>reduce 4</th>
<th>8</th>
<th>a</th>
<th>shift 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>reduce 4</td>
<td>2</td>
<td>⊳</td>
<td>reduce 2</td>
<td>5</td>
<td>p</td>
<td>shift 5</td>
<td>8</td>
<td>⊳</td>
<td>shift 6</td>
</tr>
<tr>
<td>1</td>
<td>p</td>
<td>shift 5</td>
<td>3</td>
<td>a</td>
<td>reduce 6</td>
<td>5</td>
<td>q</td>
<td>reduce 4</td>
<td>9</td>
<td>a</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>q</td>
<td>reduce 4</td>
<td>3</td>
<td>q</td>
<td>shift 7</td>
<td>5</td>
<td>⊳</td>
<td>reduce 4</td>
<td>9</td>
<td>q</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>⊳</td>
<td>reduce 4</td>
<td>3</td>
<td>⊳</td>
<td>reduce 6</td>
<td>5</td>
<td>X</td>
<td>shift 9</td>
<td>9</td>
<td>⊳</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>shift 8</td>
<td>3</td>
<td>Y</td>
<td>shift 2</td>
<td>7</td>
<td>a</td>
<td>reduce 5</td>
<td>10</td>
<td>a</td>
<td>reduce 1</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>shift 3</td>
<td>4</td>
<td>b</td>
<td>shift 10</td>
<td>7</td>
<td>⊳</td>
<td>reduce 5</td>
<td>10</td>
<td>⊳</td>
<td>reduce 1</td>
</tr>
</tbody>
</table>

1.3.3 $G_3$ SLR(1) Machine Bubble Diagram
1.3.4 **Problem**

Use the shift/reduce table and grammar given above to parse the string ⊢ pqab ⊣. Write the reversed right-canonical derivation for the string as well as the parse tree.

# Error Detection in C and WLP4

## 2.1 Error Detection in WLP4

For each WLP4 program below, point out the error in the program and state whether it is a syntax error (i.e. something the scanner or parser would catch), semantic error (something semantic analysis would catch) or runtime error (any errors occurred while running the program).

<table>
<thead>
<tr>
<th>Program</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>int wain(int x, int y) { return x^y; }</td>
<td>Syntax error</td>
</tr>
<tr>
<td>int wain(int x, int y) { int a = 100; int y = 0; // initialize y y = a*x; return y; }</td>
<td>Semantic error</td>
</tr>
<tr>
<td>int wain(int* a, int n) { // loop to get the last index while (idx &lt; n) { idx = idx + 1; } return *(a + idx); }</td>
<td>Runtime error</td>
</tr>
<tr>
<td>int wain(int a, int b) { int *c = NULL; c = &amp;a; int *d = NULL; d = &amp;b; return (c - d); }</td>
<td>Syntax error</td>
</tr>
<tr>
<td>int wain(int x, int y) { int a = ‘a’; return a + x; }</td>
<td>Semantic error</td>
</tr>
<tr>
<td>int wain(int x, int y) { int a = 0; while (a &lt; 10) { x = x + y; } return x;</td>
<td>Runtime error</td>
</tr>
</tbody>
</table>
int f(int a, int b) {
    return g(a) + g(b);
}
int g(int a) {
    return a + 32;
}
int wain(int x, int y) {
    return f(y, x);
}

2.2 Error Detection in C

For each C program below, point out the error in the program and state whether it is a syntax error (i.e. something the scanner or parser would catch) or a semantic error (something semantic analysis would catch).

float triple(float a) {
    return a * 3.0;
}

int main() {
    int* x, y;
    int a, b;

    a = triple(4.4);
    x = &a;
    y = &b;
    b = *x;
    return *y;
}

int main() {
    double a = 2.0 * .4 / getRandom();
    int b;
    b = 2;
    return b;
}