1 Introduction

This note continues the discussion of parsing based on context free languages. Here we focus on
the question of how to convert a CFG into an implementation of a parser for the specified language.
In this note, we consider the class of languages that are recognizable by top-down (or predictive)
parsing methods.

2 Recursive descent parsing

The idea of recursive descent parsing is that a CFG can be directly mapped to a collection of mutually
recursive functions, with one function for each nonterminal symbol. For example, consider the
following grammar fragment:

$$
A \rightarrow bB \\
\rightarrow cC$

then we could write a C function definition for parsing instances of $A$ as follows:

```c
A *parseA ()
{
    switch (nextToken()) {
    case TOK_b: {
        advance (); /* consume b token */
        B *b = parseB();
        return mkAB(b);
    }
    case TOK_c: {
        advance (); /* consume c token */
        C *c = parseC();
        return mkAC(c);
    }
    default: ... error ... 
    }
}
```
In some cases, the grammar might not lend itself directly to this translation. For example, if we have the following two statement forms:

\[
\begin{align*}
Stm & \rightarrow \text{if } E \text{ then } Stm \\
Stm & \rightarrow \text{if } E \text{ then } Stm \text{ else } Stm
\end{align*}
\]

In this case, we need to left-factor the grammar to produce

\[
\begin{align*}
Stm & \rightarrow \text{if } E \text{ then } Stm \text{ Else} \\
\text{Else} & \rightarrow \epsilon \\
\text{Else} & \rightarrow \text{else } Stm
\end{align*}
\]

In practice, left factoring will often be handled by additional control logic in the parse routine for the left-hand-side nonterminal.

Another problem with using recursive descent for some grammars are left-recursive rules; i.e., rules where the left-hand-side nonterminal is the first symbol on the right-hand side. Direct implementation of a left-recursive rule results in an infinite recursion. For example, the following expression grammar has left-recursive productions for both the \( E \) and \( M \) nonterminal symbols:

\[
\begin{align*}
E & \rightarrow E + M \\
E & \rightarrow M \\
M & \rightarrow M \ast A \\
M & \rightarrow A \\
A & \rightarrow \text{num} \\
A & \rightarrow (E)
\end{align*}
\]

A naïve recursive descent parser would immediately go into an infinite recursion in the \texttt{parseE} function. In this case, we can break the recursion by converting the grammar to a regular right-part notation (i.e., allow regular expressions on the right-hand side of productions):

\[
\begin{align*}
E & \rightarrow M \ast (M)^* \\
M & \rightarrow A \ast (A)^* \\
A & \rightarrow \text{num} \\
A & \rightarrow (E)
\end{align*}
\]

We use iteration in parsing code to implement the closure operators. For example, the \texttt{parseE} function has the form:

```c
E *parseE ()
{
    M *m1 = parseM();
    while (nextToken() == PLUS) {
        advance (); /* consume `+` token */
        m = mkPlus(m, parseM());
    }
    return m;
}
```

2
Note that the parse tree constructed by this code reflects the left associativity of the “+” operator.

Notice that we can convert the regular expressions on the right-hand side to CFG rules by adding additional nonterminals. The resulting grammar also avoids the left recursion, but a direct implementation as a recursive descent parser would be less compact than the iterative version. Here is the converted grammar from our example:

\[
\begin{align*}
E & \rightarrow ME' \\
E' & \rightarrow \epsilon \\
E' & \rightarrow + ME' \\
M & \rightarrow AM' \\
M' & \rightarrow \epsilon \\
M' & \rightarrow * AM' \\
A & \rightarrow \text{num} \\
A & \rightarrow (E) 
\end{align*}
\]

### 3 LL(k) parsing

It is also possible to automatically generate table-driven top-down parsers. These parsers scan over the input stream using a prefix of tokens to identify which production applies. The languages accepted by these parsers are called LL($k$), where the $k$ is the length of the prefix. For the remainder of this discussion, we will consider the case of $k = 1$.

The construction of an LL(1) parser for a grammar $\langle N, T, S, \mathcal{P} \rangle$ requires first computing the following properties:

First($A$) = \{ $U \mid U \in T$ and $A \Rightarrow^* U \alpha$ \}

Follow($A$) = \{ $U \mid U \in T$ and $S \Rightarrow^* \alpha AU \beta$ \}

First($\alpha$) = \{ $U \mid U \in T$ and $\alpha \Rightarrow^* U \beta$ \}

Intuitively, First($A$) is the set of terminals that can start a string that is derivable from $A$, Follow($A$) is the set of terminals that can follow an occurrence of $A$, and First($\alpha$) is the set of terminals that can start a string that is derivable from $\alpha$. To compute these sets for a grammar, we need to define another property

nullable($A$) = \( \begin{cases} 
\text{true} & \text{if } A \Rightarrow^* \epsilon \\
\text{false} & \text{otherwise} 
\end{cases} \)

We say that a nonterminal $A$ is nullable, if nullable($A$) = true and that a terminal symbol is never
nullable. The algorithm for computing first and follow sets for a grammar as follows:

```plaintext
forall A ∈ N do
    First(A) = ∅
    Follow(A) = ∅
end
forall U ∈ T do
    First(U) = {U}
end
do
foreach A → X₁ . . . Xₙ ∈ P do
    foreach i ∈ [1 . . . n] do
        if X₁ , . . . , Xᵢ₋₁ are nullable then
            First(A) = First(A) ∪ First(Xᵢ)
        if Xᵢ₊₁ , . . . , Xₙ are nullable then
            Follow(Xᵢ) = Follow(Xᵢ) ∪ Follow(A)
        foreach j ∈ [i + 1 . . . n] do
            if Xᵢ₊₁ , . . . , Xⱼ₋₁ are nullable then
                Follow(Xᵢ) = Follow(Xᵢ) ∪ First(Xⱼ)
        end
        if all the Xᵢ are nullable then nullable(A) = true
    end
end
until First, Follow, and nullable do not change
```

Note that to simplify the presentation, we extend the notion of first and follow sets to terminal symbols (we can avoid doing so by adding extra conditional statements to the algorithm). We extend the notion of first sets to to include $\epsilon$ and define them for strings as follows:

```plaintext
First(ε) = {ε}
First(X α) = \begin{cases} X & \text{if } X ∈ T \\
    \text{First}(X) & \text{if } \text{nullable}(X) = \text{false} \\
    \text{First}(X) ∪ \text{First}(α) & \text{if } \text{nullable}(X) = \text{true} \end{cases}
```

Once we have computed the first and follow sets, we can construct an LL(1) parse table $M$ that maps pairs of nonterminals and terminals to productions using the following algorithm:

```plaintext
foreach A → α ∈ P do
    if $\epsilon$ ∈ First(α) then
        foreach U ∈ Follow(A) do
            add $A → α$ to $M[A, U]$
        end
    end
    foreach U ∈ First(α) do
        add $A → α$ to $M[A, U]$
    end
end
```

If the resulting table has at most one production per $(A, U)$ pair, then the grammar is LL(1).
As an example, consider the grammar from 3. The nullable, first, and follow sets are given in the following table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>nullable</th>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>false</td>
<td>{num, {}}</td>
<td>{}, {$}</td>
</tr>
<tr>
<td>$E'$</td>
<td>true</td>
<td>{+}</td>
<td>{}, {$}</td>
</tr>
<tr>
<td>$M$</td>
<td>false</td>
<td>{num, {}}</td>
<td>{}, {+, {$}</td>
</tr>
<tr>
<td>$M'$</td>
<td>true</td>
<td>{∗}</td>
<td>{}, {+, {$}</td>
</tr>
<tr>
<td>$A$</td>
<td>false</td>
<td>{num, {}}</td>
<td>{}, {+, ✕, {$}</td>
</tr>
</tbody>
</table>

Note that we have included the special “end-of-input” symbol ($$) in the follow sets. The parsing table for the grammar is

<table>
<thead>
<tr>
<th>Symbol</th>
<th>num</th>
<th>+</th>
<th>✕</th>
<th>(</th>
<th>)</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E \rightarrow M E'$</td>
<td>$E \rightarrow M E'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td>$E' \rightarrow + M E'$</td>
<td>$E' \rightarrow \epsilon$</td>
<td>$E' \rightarrow \epsilon$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$M$</td>
<td>$M \rightarrow A M'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$M'$</td>
<td>$M' \rightarrow \epsilon$</td>
<td>$M' \rightarrow * A M'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A$</td>
<td>$A \rightarrow$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table-driven parser for LL(1) grammars works uses a parsing table (constructed as above) and an auxiliary stack of grammar symbols. The table-driven parser works as follows:

```java
while stack is not empty do
    let X be the top symbol on the stack
    let U be the next input symbol
    if $X \in T$ then
        if $X = U$ then
            pop X off the stack and advance input
        else
            parsing error
    else
        parsing error
    elseif $M[X, U] = A \rightarrow Y_1 \cdots Y_n$ then
        pop X and push $Y_n, \ldots, Y_1$ onto the stack
    else
        parsing error
end
```

Here is a trace of running this algorithm on the string “1 + 2 * 3” using the above parsing table:
<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$1 + 2 \times 3 $</td>
<td>parse $E \rightarrow M M'$</td>
</tr>
<tr>
<td>$E' M$</td>
<td>$1 + 2 \times 3 $</td>
<td>parse $M \rightarrow A M'$</td>
</tr>
<tr>
<td>$E' M' A$</td>
<td>$1 + 2 \times 3 $</td>
<td>parse $A \rightarrow \text{num}$</td>
</tr>
<tr>
<td>$E' M' \text{num}$</td>
<td>$1 + 2 \times 3 $</td>
<td>advance input</td>
</tr>
<tr>
<td>$E' M'$</td>
<td>$+ 2 \times 3 $</td>
<td>parse $M' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$+ 2 \times 3 $</td>
<td>parse $E' \rightarrow \div M M'$</td>
</tr>
<tr>
<td>$E' M +$</td>
<td>$+ 2 \times 3 $</td>
<td>advance input</td>
</tr>
<tr>
<td>$E' M$</td>
<td>$2 \times 3 $</td>
<td>parse $M \rightarrow A M'$</td>
</tr>
<tr>
<td>$E' M' A$</td>
<td>$2 \times 3 $</td>
<td>parse $A \rightarrow \text{num}$</td>
</tr>
<tr>
<td>$E' M' \text{num}$</td>
<td>$2 \times 3 $</td>
<td>advance input</td>
</tr>
<tr>
<td>$E' M'$</td>
<td>$\times 3 $</td>
<td>parse $M' \rightarrow \div A M'$</td>
</tr>
<tr>
<td>$E' M' A \ast$</td>
<td>$\times 3 $</td>
<td>advance input</td>
</tr>
<tr>
<td>$E' M' A$</td>
<td>$3 $</td>
<td>parse $A \rightarrow \text{num}$</td>
</tr>
<tr>
<td>$E' M' \text{num}$</td>
<td>$3 $</td>
<td>advance input</td>
</tr>
<tr>
<td>$E' M'$</td>
<td>$$</td>
<td>parse $M' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$$</td>
<td>parse $E' \rightarrow \epsilon$</td>
</tr>
<tr>
<td></td>
<td>$$</td>
<td>done</td>
</tr>
</tbody>
</table>
