PARSING ALGORITHMS FOR LL PARSERS

Table-driven predictive parsing algorithm (for LL parsers)

Input: A string of tokens and a parsing table $M$ for grammar $G$. ($S$ denotes the start symbol of grammar $G$, and $\$ \$ denotes the end-of-file symbol)

```
push($$);
push($S$);
lookahead = get_next_token();
repeat
    $X$ = top_of_stack();
    if ($X$ is a terminal or $X$ == $\$$) then
        if ($X$ == lookahead) then
            pop($X$);
            lookahead = get_next_token();
        else
            error();
    else
        if ($M[X, \text{lookahead}] = X \rightarrow Y_1 Y_2 \cdots Y_k$) then
            pop($X$);
            push($Y_k$); push($Y_{k-1}$); \cdots; push($Y_1$);
        else
            error();
until ($X$ == $\$$)
```
The FIRST set construction

To build FIRST($X$) for all grammar symbols $X$, apply the following rules until no more terminals or $\epsilon$ can be added to any FIRST set:

1. if $X$ is a terminal, FIRST($X$) is \{X\}
2. if $X \rightarrow \epsilon$ is a production then add $\epsilon$ to FIRST($X$)
3. if $X$ is a nonterminal and $X \rightarrow Y_1Y_2\cdots Y_k$ is a production then put every symbol in FIRST($Y_1$) other than $\epsilon$ to FIRST($X$)
4. if $X$ is a non-terminal and $X \rightarrow Y_1Y_2\cdots Y_k$ is a production, then put terminal $a$ in FIRST($X$) if $a$ is in FIRST($Y_i$) and $\epsilon$ is in FIRST($Y_j$) for all $1 \leq j < i$
5. if $X$ is a non-terminal and $X \rightarrow Y_1Y_2\cdots Y_k$ is a production, then put $\epsilon$ to FIRST($X$) if $\epsilon$ is in FIRST($Y_i$) for all $1 \leq i \leq k$

To construct the FIRST set for any string of grammar symbols $X_1X_2\ldots X_n$ (given the first sets for each of the symbols $X_1$, $X_2$, $\ldots$, $X_n$) apply the following rules:

FIRST($X_1X_2\ldots X_n$) contains:

- Any symbol in FIRST($X_1$) other than $\epsilon$
- Any symbol in FIRST($X_i$) other than $\epsilon$, if $\epsilon$ is in FIRST($X_j$) for all $1 \leq j < i$
- $\epsilon$, if $\epsilon$ is in FIRST($X_i$) for all $1 \leq i \leq n$
**The FOLLOW set construction**

To build FOLLOW(A) for all nonterminal symbols apply the following rules until nothing can be added to any FOLLOW set:

1. place $ in FOLLOW(S) ($ is the end-of-file symbol, and S is the start symbol).
2. if there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST($\beta$) except $\epsilon$ is placed in FOLLOW($B$)
3. if there is a production $A \rightarrow \alpha B$ then everything in FOLLOW($A$) is placed in FOLLOW($B$)
4. if there is a production $A \rightarrow \alpha B \beta$ and $\epsilon$ is in FIRST($\beta$) then everything in FOLLOW($A$) is placed in FOLLOW($B$)

**LL(1) parse table construction**

1. For all productions $A \rightarrow \alpha$, perform the following steps:
   (a) For each terminal symbol $a$ in FIRST($\alpha$), add $A \rightarrow \alpha$ to $M[A,a]$
   (b) if $\epsilon$ is in FIRST($\alpha$), add $A \rightarrow \alpha$ to $M[A,b]$ for each terminal symbol $b$ in FOLLOW($A$)
   (c) if $\epsilon$ is in FIRST($\alpha$) and $\$ is in FOLLOW($A$), add $A \rightarrow \alpha$ to $M[A,\$]$
2. Set each undefined entry of $M$ to error
LR PARSING ALGORITHMS

Table-driven shift-reduce parsing algorithm (for LR parsers)

*Input:* A string of tokens and a parsing table (that consists of two parts `ACTION` and `GOTO`) for grammar $G$

```plaintext
push initial state onto the stack;
token = next_token();
repeat forever
    s = top of stack;
    if ACTION[s,token] = "shift $s_i$" then
        push token;
        push $s_i$;
        token = next_token();
    else if ACTION[s,token] = "reduce $A \rightarrow \beta$" then
        pop 2 * $|\beta|$ symbols;
        s = top of stack;
        push $A$;
        push GOTO[s,$A$];
    else if ACTION[s, token] = "accept" then
        return;
    else error();
```
Closure and goto operations for LR(1) items

Algorithm for computing closure(I) where I is a set of LR(1) items:

```plaintext
function closure(I)
    repeat
        new_item ← false
        for each item [A → α • Bβ, a] in I,
            each production B → γ in G,
            and each terminal b in FIRST(βa),
            if [B → •γ, b] not in I then
                add [B → •γ, b] to I
                new_item ← true
            endif
        until (new_item = false)
    return I
```

Algorithm for computing goto(I, X) where I is a set of LR(1) items and X is a grammar symbol:

```plaintext
function goto(I, X)
    J ← set of items [A → αX • β, a]
        such that [A → α • Xβ, a] is in I
    J' ← closure(J)
    return J'
```
Constructing canonical collection of sets of LR(1) items

We start the construction of the collection of sets of LR(1) items with the item \([S' \rightarrow \bullet S, \$]\), where \(S'\) is the start symbol.

The algorithm for computing the canonical collection of sets of LR(1) items:

```
procedure items(G')
    I_0 ← closure({[S' \rightarrow \bullet S, \$]})
    Items ← { I_0 }
    ToDo ← { I_0 }
    while ToDo not empty do
        remove I_i from ToDo
        for each grammar symbol X do
            I_new ← goto(I_i, X)
            if I_new is not empty and I_new is not in Items then
                Items ← Items ∪ { I_new }
                ToDo ← ToDo ∪ { I_new }
            endif
        endfor
    endwhile
    return Items
```

We can construct a DFA with states corresponding to members of the constructed Items set, and the transition function corresponding to goto function defined above. This DFA recognizes the handles. Each state in this DFA corresponds to a set of LR(1) items (each member of Items set is a set of LR(1) items).

LR(1) parse table construction

The Algorithm

1. construct the collection of sets of LR(1) items for \(G\).
2. State \(i\) of the parser is constructed from \(I_i\).
   (a) if \([A \rightarrow \alpha \bullet \alpha, b]\) is in \(I_i\) and goto(I_i, a) = I_j, then set action[i, a] to “shift \(j\)”.
   (a must be a terminal)
   (b) if \([A \rightarrow \alpha \bullet, a]\) is in \(I_i\), then set action[i, a] to “reduce \(A \rightarrow \alpha\)”.
   (c) if \([S' \rightarrow S \bullet, \$]\) is in \(I_i\), then set action[i, \$] to “accept”.
3. If goto(I_i, A) = I_j, then set goto[i, A] to \(j\).
4. All other entries in action and goto are set to “error”
5. The initial state of the parser is the state constructed from the set containing the item \([S' \rightarrow \bullet S, \$]\).