

# Syntax Analysis

Robert Chen

December 30, 2022

# Contents

<b>1</b>	<b>Context Free Grammar</b>	<b>2</b>
<b>2</b>	<b>Determinism and Ambiguity</b>	<b>2</b>
<b>3</b>	<b>LR(1) Parsers</b>	<b>3</b>
<b>4</b>	<b>LALR(1) Parsers</b>	<b>5</b>

# 1 Context Free Grammar

The following is the definition of a *context free grammar*:

**Definition.** A *context free grammar* (CFG for short) is a 4-tuple  $\mathcal{G} = (\Sigma, V, R, S)$  where

- $\Sigma$  is a finite alphabet.
- $V$  is a set of variables.
- $R$  is a set of rules, where each rule  $r$  is in the form  $r : A \rightarrow w$ , with  $A \in V$  and  $w \in (V \cup \Sigma)^*$ .  
We denote  $A$  as the *left-hand-side (LHS)* and  $w$  as the *right-hand-side (RHS)*.
- $S \in V$ : the starting variable.

A CFG also defines a language. First let's see the definition of a *derivation*:

**Definition.** Let  $\mathcal{G} = (\Sigma, V, R, S)$  be a CFG,  $u = u_1 A u_2$  where  $u_1, u_2 \in (V \cup \Sigma)^*$  and  $A \in V$ , and  $r$  be a rule  $r : A \rightarrow w$  in  $R$ . A *derivation* on  $u$  is denoted as

$$u_1 A u_2 \xrightarrow{r} u_1 w u_2$$

which represents changing a variable in  $u$  to the RHS of rule  $r$ .

Then the definition of the language of a CFG follows:

**Definition.** Let  $\mathcal{G} = (\Sigma, V, R, S)$  be a CFG and  $L(\mathcal{G})$  denote the language that  $\mathcal{G}$  accepts, then

$$L(\mathcal{G}) = \{w \in \Sigma^* : \exists \text{ a finite sequence of derivations from } S \text{ to } w\}$$

## 2 Determinism and Ambiguity

*Remark.* This section is mainly to provide some context, so most of the content is not complete. Many interesting theoretical results are not included, as they are not the focus of this documentation.

Similar to regexes, CFGs also have a class of machines that accept the languages of CFGs, called the *push down automaton* (PDA for short). Similar to finite automata, PDAs also have non-deterministic and deterministic (denoted as DPDA) variants. Unfortunately, DPDAs are not as powerful as general PDAs, meaning there are CFGs that cannot be parsed by DPDAs but can be parsed by PDAs. Thus, the notion of *deterministic CFGs* come in:

**Definition.** A CFG  $\mathcal{G}$  is *deterministic* there is a corresponding DPDA that derives  $\mathcal{G}$ .

Deterministic CFGs (DCFG for short) are often preferred because the DPDAs that derive them yields better run-time performance (similar idea to DFA). Another property of CFG is its *ambiguity*. For any CFG  $\mathcal{G} = (\Sigma, V, R, S)$ , every  $w \in L(\mathcal{G})$  can be obtained from a finite sequence of derivations on  $S$ . Even if we limit the sequence of derivations to derive the leftmost variable first, the language  $L(\mathcal{G})$  is still unchanged. This kind of derivation is called a *leftmost-derivation*. The definition of the *ambiguity* of a CFG follows:

**Definition.** A CFG  $\mathcal{G} = (\Sigma, V, R, S)$  is *ambiguous* if there is a  $w \in L(\mathcal{G})$  that can be obtained from two different finite sequences of leftmost derivations on  $S$ .

Ambiguous CFGs are also something we want to avoid, since it may result in a program that can be parsed in two different ways, and thus how the program should be executed becomes unclear. Fortunately, DCFGs are always unambiguous (however, the converse is not true), thus if we restrict the CFGs to be deterministic, we don't have to worry about ambiguity.

Therefore, we now focus solely on how to parse DCFGs. We introduce the LR(1) and LALR(1) parsing tables, which are driven by the parsing algorithm to produce parsers. The LR(1) parser comes from a family of LR( $k$ ) parsers, which are a set of DPDAs. Theoretical results show that the family of LR( $k$ ) parsers are actually **exactly** the set of DPDAs, and that all LR( $k$ ) parsers can be transformed into LR(1) parsers. In other words, LR(1) parsers can parse all DCFGs. LALR(1) is a method to reduce the number of states produced by LR(1) parsers.

### 3 LR(1) Parsers

Before getting into the definition of a LR(1) parser, we have to first look at a few definitions. The following definition all uses the DCFG  $\mathcal{G} = (\Sigma, V, R, S)$ .

**Definition.** An *LR(1) item* is a 3-tuple  $w' = (r, i, z)$  where

- $r$  is a rule in  $R$ .
- $w \in (\Sigma \cup V \cup \{\cdot\})^*$  comes from the RHS of  $r$  with a  $\cdot$  inserted at any position.
- $z \subseteq \Sigma$  is a set of lookahead symbols.

**Definition.** For any variable  $A \in V$ ,  $A$  is called *nullable* if one of the following is satisfied:

- There is a rule  $A \rightarrow \varepsilon$  in  $R$ .
- There is a rule  $A \rightarrow A_1 A_2 \cdots A_n$  in  $R$ , and for every  $i = 1, \dots, n$ ,  $A_i \in V$  and  $A_i$  is nullable.

**Definition.** For any string  $w \in (\Sigma \cup V)^*$ ,  $w$  is called *nullable* if  $w = V_1 V_2 \dots V_n$ , and for every  $i = 1, \dots, n$ ,  $A_i \in V$  and  $A_i$  is nullable.

**Definition.** The *first-set* of a string  $w \in (\Sigma \cup V)^*$ , denoted by  $\text{first}(w)$ , is a set  $z \subseteq \Sigma$  constructed as follows:

- If  $w = aw_1$  where  $a \in \Sigma$  and  $w_1 \in (\Sigma \cup V)^*$ , then  $a \in z$ .
- If  $w = Aw_1$  where  $A \in V$  and  $w_1 \in (\Sigma \cup V)^*$ , then for every rule  $r : A \rightarrow x$ , we have  $\text{first}(x) \subseteq z$ .
- If  $w = Aw_1$  where  $A \in V$  and  $w_1 \in (\Sigma \cup V)^*$  and  $A$  is nullable, then  $\text{first}(w_1) \subseteq z$ .
- If  $w = \varepsilon$ , then  $z = \emptyset$ .

**Definition.** Given  $w \in (\Sigma \cup V)^*$  and  $z \subseteq \Sigma$ , the *follow-set* of  $w$  and  $z$ , denoted by  $\text{follow}(w, z)$ , is defined as

$$\text{follow}(w, z) = \begin{cases} z & \text{if } w = \varepsilon \\ \text{first}(w) \cup z & \text{if } w = Aw_1 \text{ where } A \in V \text{ and } A \text{ is nullable} \\ \text{first}(w) & \text{otherwise} \end{cases}$$

**Definition.** A *closure* of an LR(1) item  $w'$ , denoted by  $\text{closure}(w')$ , is a set  $q'$  of LR(1) items where a set  $q$  is first constructed as follows:

- $w' \in q$ .
- If  $(r, w_1 \cdot Bw_2, z) \in q$ , where  $w_1, w_2 \in (\Sigma \cup V)^*$  and  $B \in V$ , then for every rule  $(s : B \rightarrow x) \in R$ , we have

$$(s, \cdot x, \text{follow}(w_2, z)) \in q$$

Then, the final set  $q'$  is constructed from  $q$  with the following:

$$q' = \left\{ (r, w, z) : (r, w, z_1), \dots, (r, w, z_n) \in q \text{ and } z = \bigcup_{i=1}^n z_i \right\}$$

With these definitions, we can construct the *LR(1) state transition graph* using the following algorithm:

---

**Algorithm 1** LR(1)-STATE-TRANSITION-GRAPH

---

- 1: **function** LR(1)-STATE-TRANSITION-GRAPH
- 2:    $\Sigma \leftarrow \Sigma \cup \{\$$  ▷ \$ represents the end-of-string symbol
- 3:    $V \leftarrow V \cup \{S'\}$
- 4:    $r \leftarrow (S' \rightarrow S)$
- 5:    $R \leftarrow R \cup \{r\}$
- 6:    $q_0 \leftarrow \text{closure}((r, \cdot S, \$))$  ▷ The initial state  $q_0$  represents the state where it is expecting to derive  $S$ , followed by lookahead symbol \$
- 7:    $Q \leftarrow \{q_0\}$

```

8:    $\delta \leftarrow \emptyset$ 
9:   for  $p \in Q$  do
10:     for  $(r, w_1 \cdot w_2, z) \in p$  do
11:       if  $w_2 = xw_3$  then
12:          $q \leftarrow \text{closure}((r, w_1x \cdot w_3, z))$ 
13:          $Q \leftarrow Q \cup \{q\}$ 
14:          $\delta \leftarrow \delta \cup \{(p, x), q\}$ 
15:       end if
16:     end for
17:   end for
18:   return  $(Q, q_0, \delta)$ 
19: end function

```

---

After constructing the state-transition graph, we can use it to construct the *LR(1) state table* using the following algorithm:

---

**Algorithm 2** LR(1)-STATE-TABLE

---

```

1: function LR(1)-STATE-TABLE( $Q, q_0, \delta$ )
2:    $T \leftarrow$  a two-dimensional array, all initialized to error
3:   for  $((p, x), q) \in \delta$  do
4:     if  $x \in \Sigma$  then
5:        $T[p][x] \leftarrow \text{shift}(q)$  ▷ Represents a shift action
6:     else
7:        $T[p][x] \leftarrow \text{goto}(q)$  ▷ Represents a goto action
8:     end if
9:   end for
10:  for  $p \in Q$  do
11:    for  $(r, w_1 \cdot w_2, z) \in p$  do
12:      if  $w_2 = \varepsilon$  then
13:        for  $a \in z$  do
14:          if  $T[p][a] \neq \text{error}$  and  $T[p][a] \neq \text{reduce}(r)$  then
15:            if  $T[p][a]$  is a shift action then
16:              print("shift-reduce conflict on state  $p$ ")
17:            else if  $T[p][a]$  is a reduce action then
18:              print("reduce-reduce conflict on state  $p$ ")
19:            end if
20:          return null
21:          else
22:             $T[p][a] \leftarrow \text{reduce}(r)$  ▷ Represents a reduce action
23:          end if
24:        end for
25:      end if
26:    end for
27:  end for
28:  return  $(q_0, T)$ 
29: end function

```

---

Note that we introduced the concept of *conflicts* — states where different actions can be taken. If conflicts exist, then it is not possible to build an LR(1) parser using the above algorithm.

With the LR(1) state table constructed, we can finally run the parsing algorithm on any word  $s \in \Sigma^*$ :

---

**Algorithm 3** PARSING ALGORITHM

---

```

1: function PARSING-ALGORITHM( $q_0, T, s$ )
2:    $s \leftarrow s\$$ 
3:    $Z \leftarrow$  an empty stack
4:    $Z.\text{push}(q_0)$ 
5:    $p \leftarrow q_0$ 
6:    $i \leftarrow 1$ 

```

```

7:  while  $i \leq |s|$  do
8:       $a \leftarrow s_i$ 
9:       $\text{action} \leftarrow T[p][a]$ 
10:     if  $\text{action} = \text{error}$  then
11:         print("Syntax error")
12:         return false
13:     else if  $\text{action} = \text{shift}(q)$  for some  $q \in Q$  then
14:          $Z.\text{push}(a)$ 
15:          $Z.\text{push}(q)$ 
16:          $p \leftarrow q$ 
17:          $i \leftarrow i + 1$ 
18:     else if  $\text{action} = \text{reduce}(r)$  for some  $r \in R$  then
19:          $n \leftarrow$  the length of the RHS of  $r$ 
20:         for  $2n$  times do
21:              $Z.\text{pop}()$ 
22:         end for
23:          $p \leftarrow Z.\text{top}()$ 
24:          $A \leftarrow$  LHS of  $r$ 
25:          $q \leftarrow T[p][A]$ 
26:          $Z.\text{push}(a)$ 
27:          $Z.\text{push}(q)$ 
28:          $p \leftarrow q$ 
29:     end if
30: end while
31: return true
32: end function

```

---

## 4 LALR(1) Parsers

LALR(1) parsers are exactly the same as LR(1) parsers except for the construction of the state table. The following algorithm shows how to construct an LALR(1) state table from an LR(1) state table:

---

### Algorithm 4 LALR(1) STATE TABLE

---

```

1: function LALR(1)-STATE-TABLE( $q_0, T$ )
2:      $\text{newItems} \leftarrow$  an empty dictionary
3:     for  $q \in Q$  do
4:          $W \leftarrow \emptyset$ 
5:         for  $(w, z) \in q$  do
6:              $W \leftarrow W \cup \{w\}$ 
7:         end for
8:         if  $q = q_0$  then
9:              $W_0 \leftarrow W$ 
10:        end if
11:        if  $\text{newItems}[W]$  exists then
12:             $\text{newItems}[W] \leftarrow \text{newItems}[W] \cup \{q\}$ 
13:        else
14:             $\text{newItems}[W] \leftarrow \{q\}$ 
15:        end if
16:    end for
17:    Find  $q'_0$  such that  $q_0 \in \text{newItems}[q'_0]$ 
18:     $T' \leftarrow$  a two-dimensional array, all initialized to error
19:    for  $p' \in \text{newItems.values}$  do
20:        for  $p \in p'$  do
21:            for  $x \in \Sigma \cup V$  do
22:                 $\text{action}_{\text{new}} \leftarrow T[p][x]$ 
23:                if  $\text{action}_{\text{new}} = \text{shift}(q)$  for some  $q \in Q$  then

```

```

24:         Find  $q'$  such that  $q \in \text{newItems}[q']$ 
25:          $\text{action}_{\text{new}} \leftarrow \text{shift}(q')$ 
26:     else if  $\text{action}_{\text{new}} = \text{goto}(q)$  for some  $q \in Q$  then
27:         Find  $q'$  such that  $q \in \text{newItems}[q']$ 
28:          $\text{action}_{\text{new}} \leftarrow \text{goto}(q')$ 
29:     end if
30:     if  $T'[p'][x] \neq \text{error}$  and  $\text{action}_{\text{new}} \neq \text{error}$  and  $T'[p'][x] \neq \text{action}_{\text{new}}$  then
31:         if  $\text{action}_{\text{new}}$  is a shift action then
32:             print("shift-reduce conflict on state  $p'$ ")
33:         else if  $\text{action}_{\text{new}}$  is a reduce action then
34:             print("reduce-reduce conflict on state  $p'$ ")
35:         end if
36:         return null
37:     end if
38:      $T'[p'][x] \leftarrow \text{action}_{\text{new}}$ 
39: end for
40: end for
41: end for
42: return  $(q'_0, T')$ 
43: end function

```

---

Note that the construction of the LALR(1) parsing table may introduce new conflicts, which means that even if a CFG can be used to construct an LR(1) parser, the same CFG may not necessarily be able to be used to construct an LALR(1) parser.