# Syntax Analysis

Robert Chen

August 31, 2021

# Contents

# 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 $r : A \to w$, 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 $r : A \to w$ 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 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 $A \to \varepsilon$ in $R$.
- There is a rule $A \to A_1 A_2 \cdots A_n$ in $R$, and for every $i = 1, \ldots, n$, $A_i \in V$ 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 $r : A \to x$, 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 $z = \varnothing$.

**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 $(s : B \to x) \in R$, we have

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

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

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

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

**Algorithm 1** LR(1)-State-Transiton-Graph

**function** LR(1)-State-Transition-Graph
  $\Sigma \leftarrow \Sigma \cup \{\$\}$ ▷ \$ represents the end-of-string symbol
  $V \leftarrow V \cup \{S'\}$
  $r \leftarrow (S' \to S)$
  $R \leftarrow R \cup \{r\}$
  $q_0 \leftarrow \text{closure}((r, \cdot S, \$))$ ▷ The initial state $q_0$ represents the state where it is expecting to derive $S$, followed by lookahead symbol \$
  $Q \leftarrow \{q_0\}$
  $\delta \leftarrow \varnothing$
  **for** $p \in Q$ **do**

```
        for x ∈ Σ ∪ V do
            q ← ∅
            for (r, w1 · w2, z) ∈ p do
                if w2 = xw3 then
                    q ← closure((r, w1x · w3, z))
                end if
            end for
            Q ← Q ∪ {q}
            δ ← δ ∪ {((p, x), q)}
        end for
    end for
    return (Q, q0, δ)
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

```
function LR(1)-State-Table(Q, q0, δ)
    T ← a two-dimensional array, all initialized to error
    for ((p, x), q) ∈ δ do
        if x ∈ Σ then
            T[p][x] ← shift(q)                          ▷ Represents a shift action
        else
            T[p][x] ← goto(q)                           ▷ Represents a goto action
        end if
    end for
    for p ∈ Q do
        for (r, w1 · w2, z) ∈ p do
            if w2 = ε then
                for a ∈ z do
                    if T[p][a] ≠ error and T[p][a] ≠ reduce(r) then
                        if T[p][a] is a shift action then
                            print("shift-reduce conflict on state p")
                        else if T[p][a] is a reduce action then
                            print("reduce-reduce conflict on state p")
                        end if
                        return null
                    else
                        T[p][a] ← reduce(r)             ▷ Represents a reduce action
                    end if
                end for
            end if
        end for
    end for
    return (q0, T)
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

```
function Parsing-Algorithm(q0, T, s)
    s ← s$
    Z ← an empty stack
    Z.push(q0)
    p ← q0
```

```
    for each character a of s in order do
        action ← T[p][a]
        if action = error then
            print("Syntax error")
            return false
        else if action = shift(q) for some q ∈ Q then
            Z.push(a)
            Z.push(q)
            p ← q
        else if action = reduce(r) for some r ∈ R then
            n ← the length of the RHS of r
            for 2n times do
                Z.pop()
            end for
            p ← Z.top()
            A ← LHS of r
            q ← T[p][A]
            Z.push(a)
            Z.push(q)
            p ← q
        end if
    end for
    return true
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

```
function LALR(1)-State-Table(q_0, T)
    newItems ← an empty dictionary
    for q ∈ Q do
        W ← ∅
        for (w, z) ∈ q do
            W ← W ∪ {w}
        end for
        if q = q_0 then
            W_0 ← W
        end if
        if newItems[W] exists then
            newItems[W] ← newItems[W] ∪ {q}
        else
            newItems[W] ← {q}
        end if
    end for
    Find q'_0 such that q_0 ∈ newItems[q'_0]
    T' ← a two-dimensional array, all initialized to error
    for p' ∈ newItems.values do
        for p ∈ p' do
            for x ∈ Σ ∪ V do
                action_new ← T[p][x]
                if action_new = shift(q) for some q ∈ Q then
                    Find q' such that q ∈ newItems[q']
                    action_new ← shift(q')
```

```
            else if action_new = goto(q) for some q ∈ Q then
                Find q' such that q ∈ newItems[q']
                action_new ← goto(q')
            end if
            if T'[p'][x] ≠ error and action_new ≠ error and T'[p'][x] ≠ action_new then
                if action_new is a shift action then
                    print("shift-reduce conflict on state p'")
                else if action_new is a reduce action then
                    print("reduce-reduce conflict on state p'")
                end if
                return null
            end if
            T'[p'][x] ← action_new
        end for
    end for
    end for
    return (q'_0, T')
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.

# 5 Empty Productions

The algorithms above only work when all rules in the CFG are not empty productions; in other words, all $r : A \to x$ in $R$ satisfies $x \neq \varepsilon$. However, having empty productions (i.e. $r : A \to \varepsilon$) can be very helpful in constructing the CFG for various programming language constructs. Therefore, here we show how to remove empty productions from a given CFG:

**Algorithm 5** Empty Production Removal

```
function EMPTY-PRODUCTION-REMOVAL(G = (Σ, V, R, S))
    changed ← true
    while changed do
        changed ← false
        for (r : A → x) ∈ R do
            if x = ε then
                for (s : B → y) ∈ R do
                    n ← the number of occurences of A in y
                    if n > 0 then
                        for m = 0, 1, ..., 2^n − 1 do
                            y' ← y with the i-th occurrence of A removed if the i-th bit of m is 1
                            R ← R ∪ {B → y'}
                            changed ← true
                        end for
                    end if
                end for
            end if
        end for
    end while
end function
```