Syntactical Analysis

May 13, 2020

Syntactical analysis, also known as *parsing*, is the task of taking a text, checking whether it belongs to a given context-free language described by a grammar, and reconstructing its derivation in that grammar. The topic is mostly associated with compiler construction but can be applied to any case where one must process non-regular input, such as mathematical formulae.

Technically speaking, parsing unites theory from regular and from context-free languages and consists of two tasks: *lexical analysis* and *syntactical analysis*.

Example 1 *Consider this grammar G:*

$$E \to \mathbf{int} \mid E + E \mid E * E \mid (\ E\ )$$

And this stream of input symbols: `10␣*␣(6␣+␣5)`

To see that the input is a valid word generated by G, it first needs to be split into a stream of terminal symbols that make sense to G, such as this:

$$\mathbf{int}\ *\ (\ \mathbf{int}\ +\ \mathbf{int}\ )$$

This step is called *lexical analysis*. One then wishes to reconstruct a derivation that generates this word, such as this:

$$E \to E{*}E \to E{*}(E) \to E{*}(E{+}E) \to E{*}(E{+}\mathbf{int}) \to E{*}(\mathbf{int}{+}\mathbf{int}) \to \mathbf{int}{*}(\mathbf{int}{+}\mathbf{int})$$

or alternatively, a parse tree:

$$\begin{tikzcd}
 & E \arrow[d] & & & \\
E \arrow[d] & * & E \arrow[dr] & & \\
\mathbf{int} & & (& E \arrow[d] &) \\
 & & E \arrow[d] & + & E \arrow[d] \\
 & & \mathbf{int} & & \mathbf{int}
\end{tikzcd}$$

This step is called *syntactic analysis*. The parse tree allows to semantically evaluate the expression.

Sophisticated tools exist to automatize these tasks: *lex* / *flex* for lexical analysis and *yacc* / *bison* for syntactic analysis.

1 Lexical analysis

This first step is the easier of the two, and we will treat it less formally. Recall that the goal of lexical analysis is to transform a sequence of input symbols (bytes or characters) into a sequence of *terminals*. Each terminal will be either a single input symbol (such as `+` or `*`) or a sequence such as `10` (an **int**) or keywords like `if` and `then`. The same terminal may take different forms, e.g. `10` and `5` are both instances of **int**. Likewise, in a programming language, an identifier could be any alphanumerical sequence.

In common practice, every terminal is represented by a regular expression. In the case of keywords, these admit a single match, in the case of **int** they would admit non-empty sequences of digits ($R_{\mathbf{int}} := \{\mathtt{0}, \ldots, \mathtt{9}\}^+$), etc. This leaves two problems:

- The input stream `10` could correspond to either one single occurrence of $R_{\mathbf{int}}$ or two seperate ones. The convention here is to match a maximal part of the input, so this would translate into one occurrence.

- Two different regular expressions could match the same maximal input. P.ex., `then` could be either a keyword or an identifier. One therefore orders the expressions in decreasing priority (e.g., prioritising keywords).

The lexical-analysis problem is then solved as follows:

– Input: Regular expressions $R_1, \ldots, R_k$ and input $a_1 \cdots a_n$.

– Output: A sequence of terminals (identified as $1, \ldots, k$).

1. Set $i := 1$ and translate the regular expressions into deterministic complete automata $A_1, \ldots, A_k$.

2. While $i \leq n$, repeat the following:

   (a) Find the longest word $a_i \cdots a_j$ accepted by some A_l. If no such word exists, report a syntax error.

   (b) Find the least l such that A_l accepts $a_i \cdots a_j$.

   (c) Output l and set $i := j + 1$.

Whitespace in the input is usually handled by an additional regular expression whose occurrences are omitted from the output. The running time of the algorithm is "usually" linear but worst-case quadratic (for pathological R_i). The tool *flex* allows one to perform the algorithm above for a given set of regular expressions. Each expression is associated with C code to be executed when an occurrence is found.

2 Syntaxtic analysis with PDA

This is a more complex topic, we will treat it more formally. In this section, we will develop a pushdown automaton that can parse context-free grammars. We will then consider how to determinize and optimize it in Section 3. However, the concepts presented in this section already go a long way towards understanding syntactic-analysis tools in practice.

2.1 Grammars and parse trees

We will assume that the theory surrounding context-free languages and grammars is well-known. The following definitions are merely to establish notations.

Definition 2 *A (context-free)* grammar *is a tuple* $G = \langle \Sigma, V, P, S' \rangle$*, where* Σ *is a finite alphabet of* terminals*,* V *is a finite set of* variables*,* P *is a set of* productions*, and* S' *is a starting symbol. Here, a production is of the form* $X \to \alpha$*, where* $X \in V$ *and* $\alpha \in (\Sigma \cup V)^*$*. We assume that* S' *only appears in a single production* $P_0 := S' \to S$*.*

A *derivation* for $w \in \Sigma^*$ is a sequence of productions transforming S' into w, and a derivation can be associated with a *parse tree*, see Example 1. In a leftmost/rightmost derivation, each production is applied to the leftmost/rightmost remaining terminal. For instance, the derivation shown in Example 1 is rightmost. The word w is in the language of G ($w \in \mathcal{L}(G)$) if there exists a derivation for w. A grammar is said to be *unambiguous* is each $w \in \mathcal{L}(G)$ possesses a unique parse tree. For instance, the grammar G in Example 1 is ambiguous.

The *syntactic-analysis problem* is to test whether $w \in \mathcal{L}(G)$, and if so, construct a corresponding parse tree. In general, the syntactic- analysis problem can be solved in time $\mathcal{O}(|w|^3)$, e.g. using the algorithm of Cooke, Younger, and Kasami (CYK), and said algorithm also allows to construct all possible parse trees for w. However, a cubic running time is unacceptable for many applications, like compilers. Also, a programming language with an ambiguous grammar cannot have a well-defined semantics. Thus, we will be interested in grammars G with the following properties:

(i) The syntactic-analysis problem for G must be solvable in $\mathcal{O}(|w|)$.

(ii) G must be unambiguous.

Given a word w, we will read it *from left to right.* There are two fundamental ways to construct a parse tree for w. In both cases, the parser keeps a state $\gamma \in (\Sigma \cup V)^*$.

Top-down parsing: Initially $\gamma := S'$. It then either expands the state or consumes a symbol:

- Expansion: If $\gamma = X\delta$ for some variable X, choose a production $X \to \alpha$ and set $\gamma := \alpha\delta$.

- Consumption: If $\gamma = a\delta$ for some terminal a, and a is the next input symbol is a, consume a and set $\gamma := \delta$.
- Accept if $\gamma = \varepsilon$ at the end of w.

This generates the parse tree from top to bottom, and the order of expansions corresponds to a leftmost derivation.

Bottom-up parsing: Initially $\gamma := \varepsilon$. There are two actions, Shift and Reduce:

- Shift: If a is the next input symbol, set $\gamma := \gamma a$.
- Reduce: If $\gamma = \delta\alpha$ and there is a production $X \to \alpha$, set $\gamma := \delta X$.
- Accept if $\gamma = S'$ at the end of w.

This generates the parse tree from bottom to top, and the order of reductions is the reverse of a rightmost derivation.

Exercise: Apply both methods to Example 1.

Both variants possess rich theories. For time reasons, we shall concentrate on the bottom-up approach, which also happens to be the one implemented by the tools we consider.

2.2 Pushdown automata

Definition 3 *A* pushdown automaton *(PDA) is a tuple $\mathcal{A} = \langle Q, \Sigma, Z, T, q_0, F\rangle$, where Q is a finite set of* states*, Σ, Z are finite alphabets of input and stack symbols, respectively, T are the transitions, q_0 is an initial state, and $F \subseteq Q$ are the final states.*

Σ' denotes $\Sigma \cup \{\varepsilon\}$. We make two departures from the standard notation:

- Configurations will be noted with the *top of the stack to the right.* And for convenience, we shall also place the state there, so a configuration is a tuple $\langle w, q\rangle \in Z^* \times Q$. The initial configuration is $\langle \varepsilon, q_0\rangle$.
- Transitions can either push or pop a symbol:

  $$T \subseteq (Q \times \Sigma' \times Z \times Q) \cup (Z \times Q \times \Sigma' \times Q)$$

  where (i) $\langle w, q\rangle \xrightarrow{a} \langle wz, q'\rangle$ if $\langle q, a, z, q'\rangle \in T$, and (ii) $\langle wz, q\rangle \xrightarrow{a} \langle w, q'\rangle$ if $\langle z, q, a, q'\rangle \in T$.

As usual, a PDA accepts when reaching a state of F at the end of the input.

The bottom-up parser from Section 2.1 works like a pushdown automaton (with $Z := \Sigma \cup V$), except that reductions allow to look at multiple symbols on the stack. This is a special case of PDA with regular stack conditions:

Definition 4 *A* regular PDA *is a PDA $\mathcal{A} = \langle Q, \Sigma, Z, T, q_0, F\rangle$, where the set of transitions T is*

$$T \subseteq (Rec(Z^*) \times Q \times \Sigma' \times Z \times Q) \cup (Rec(Z^*) \times Q \times \Sigma' \times Q)$$

where (i) $\langle w, q\rangle \xrightarrow{a} \langle wz, q'\rangle$ if $\langle L, q, a, z, q'\rangle \in T$ and $w \in L$ (push), and (ii) $\langle wz, q\rangle \xrightarrow{a} \langle w, q'\rangle$ if $\langle L, q, a, q'\rangle \in T$ and $wz \in L$ (pop).

The following result was shown in the TP:

Proposition 5 *Let $\mathcal{A}$ be a regular PDA. One can construct a (normal) PDA $\mathcal{A}'$ accepting the same language, as follows. Let k be the number of rules in T and $\mathcal{A}_i = \langle Q_i, Z, \delta_i, \iota_i, F_i\rangle$, for $i = 1, \ldots, k$, deterministic complete automata for the regular languages used in T. Denote $\mathcal{Q} := Q_1 \times \cdots \times Q_k$, $\iota := \langle \iota_1, \ldots, \iota_k\rangle$, $\mathcal{F}_i := \{\, \langle q_1, \ldots, q_k\rangle \in \mathcal{Q} \mid q_i \in F_i \,\}$, and $\delta : \mathcal{Q} \times Z \to \mathcal{Q}$ with $\delta(\langle q_1, \ldots, q_k\rangle, z) := \langle \delta_1(q_1, z), \ldots, \delta_k(q_k, z)\rangle$. We then construct $\mathcal{A}' := \langle \mathcal{Q} \times Q, \Sigma, \mathcal{Q} \times Z, T', \langle \iota, q_0\rangle, \mathcal{Q} \times F\rangle$, where:*

- *(push) for each $\langle L_i, q, a, z, q'\rangle \in T$ and $f \in \mathcal{F}_i$ we will have a tuple $\langle\langle f, q\rangle, a, \langle f, z\rangle, \langle \delta(f, z), q'\rangle\rangle \in T'$;*
- *(pop) for each $\langle L_i, q, a, q'\rangle \in T$, $z \in Z$, $q'' \in \mathcal{Q}$, and $f \in F_i$ we will have $\langle\langle q'', z\rangle, \langle f, q\rangle, a, \langle q'', q'\rangle\rangle \in T'$.*

This construction maintains an invariant where if $\mathcal{A}$ reaches a configuration $\langle z_1 \cdots z_n, q\rangle$, then $\mathcal{A}'$ reaches the configuration $\langle\langle q'_0, z_1\rangle \cdots \langle q'_{n-1}, z_n\rangle, \langle q'_n, q\rangle\rangle$, where $q'_0 = \iota$ and $q'_{i+1} = \delta(q'_i, z_{i+1})$ for $i = 0, \ldots, n-1$. In other words, $\mathcal{A}'$ uses its stack to record the unique path that the stack contents of $\mathcal{A}$ take in the (finite) automata $\mathcal{A}_1, \ldots, \mathcal{A}_k$.

2.3 The Shift/Reduce automaton

We can now specify the behaviour of the bottom-up parser as a regular PDA.

Definition 6 *Let $G = \langle \Sigma, V, P, S'\rangle$ be a grammar. The* items *of a production $X \to \alpha$ are $Items(X \to \alpha) = \{\, X \to \beta.\gamma \mid \alpha = \beta\gamma \,\}$. The* items *of G are the items of all its productions. We let $\mathcal{I}_G := 2^{Items(G)}$ and write $\mathcal{I}$ if G is understood.*

Example 7 *Let $G_1 = \langle \{a, b, c\}, \{S', S, T, U\}, \{P_0, P_1, P_2, P_3, P_4\}, S\rangle$, with*

$$P_0 := S' \to S \quad P_1 := S \to TU \quad P_2 := T \to aTb \quad P_3 := T \to ab \quad P_4 := U \to c$$

Then $Items(G) = \{S' \to .S, S' \to S., S \to .TU, S \to T.U, S \to TU., T \to .aTb, T \to a.TB, \ldots\}$.

A grammar G can be recognized by a regular PDA $\langle Q, \Sigma, Z, T, q_0, F\rangle$:

- $Q := \{\bot, \top\} \cup Items(G)$

- $Z := \Sigma \cup V$
- $q_0 := \bot$
- $F := \{\top\}$
- $T := T_{shift} \cup T_{reduce} \cup T_{accept}$ with
  - $T_{shift} = \{\, \langle Z^*, \bot, a, a, \bot \rangle \mid a \in \Sigma \,\}$;
  - $T_{reduce} = \{\, \langle Z^*\alpha, \bot, \varepsilon, X \to \alpha. \rangle \mid X \to \alpha \in P \,\}$
    $\cup \{\, \langle Z^*, X \to \alpha z.\beta, \varepsilon, X \to \alpha.z\beta \rangle \mid X \to \alpha\beta \in P \,\}$
    $\cup \{\, \langle Z^*, X \to .\alpha, \varepsilon, X, \bot \rangle \mid X \to \alpha \in P \setminus \{P_0\} \,\}$;
  - $T_{accept} = \{\langle \{S\}, \bot, \varepsilon, \top \rangle\}$.

With T_{shift} one simply consumes an input symbol and pushes it onto the stack, remaining in state $\bot$. With T_{reduce}, one replaces α by X on the stack if $X \to \alpha \in P$; here the items are simply used as temporary control states. State $\top$ is used for accepting when the stack only contains S'. Most of the transitions use Z^* as their regular language (so there is nothing to check), the only exceptions are the conditions $Z^*\alpha$ in T_{reduce}; these are simply checking whether the last symbols on the stack correspond to a right-hand side of a production. For P_0 we instead check whether the stack contains only S and accept right away.

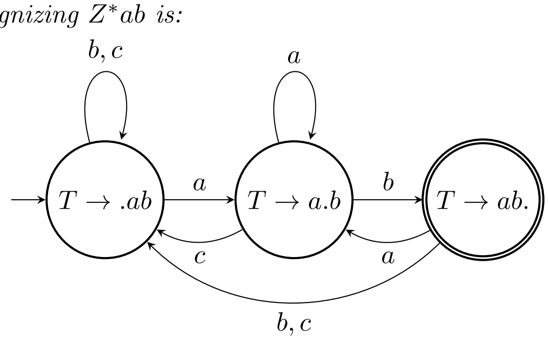
Let us see how an *ordinary* PDS for G looks like, according to the construction of Proposition 5. We need automata recognizing $Z^*\alpha$, for any right-hand side α in $P \setminus P_0$. The following proposition is given without proof:

Proposition 8 *Let $X \to \alpha$ be a production. The minimal deterministic complete automaton recognizing $Z^*\alpha$ is (isomorphic to)*

$$\langle \mathit{Items}(X \to \alpha), Z, \delta, X \to .\alpha, X \to \alpha. \rangle$$

and $\delta(X \to .\alpha, w) = X \to \beta.\gamma$ such that β is the longest prefix of α that is also a suffix of w.

Example 9 *For $P_3 := T \to ab$, the minimal deterministic complete automaton recognizing Z^*ab is:*



It follows from Proposition 8 that the PDA $\mathcal{A}'$ constructed from Proposition 5 has states $\mathcal{I} \times Q$ and stack alphabet $I \times Z$. A configuration of that PDA is a tuple $\langle\langle I_0, z_1\rangle \cdots \langle I_{n-1} z_n\rangle, \langle I_n, q\rangle\rangle$. If we concentrate on configurations having state $\perp$, then we can more conveniently denote the state as a path between of item sets, linked by stack symbols. Also, we will ignore the items for P_0 since they are treated specially.

Example 10 *Consider G_1 from Example 7 on the input aabbc. An accepting run of $\mathcal{A}'$ is as follows:*

The initial configuration consists of only the initial items:

$$I_0 := \{S \to .TU, T \to .aTb, T \to .ab, U \to .c\}.$$

We can then shift the first a (leaving abbc), going to configuration $I_0 \xrightarrow{a} I_1$ with

$$I_1 := \{S \to .TU, T \to a.Tb, T \to a.b, U \to .c\}.$$

Next, we shift a again (leaving bbc), going to $I_0 \xrightarrow{a} I_1 \xrightarrow{a} I_1$.
We shift b (leaving bc) and go to $I_0 \xrightarrow{a} I_1 \xrightarrow{a} I_1 \xrightarrow{b} I_2$ with

$$I_2 := \{S \to .TU, T \to .aTb, T \to ab., U \to .c\}.$$

Since $T \to ab.$ is a final state, we can reduce the corresponding rule P_3, replacing ab by T on the stack. This involves multiple steps of $\mathcal{A}'$, ending up with $I_0 \xrightarrow{a} I_1 \xrightarrow{T} I_3$, where:

$$I_3 := \{S \to T.U, T \to aT.b, T \to .ab, U \to .c\}.$$

Shifting b we obtain $I_0 \xrightarrow{a} I_1 \xrightarrow{T} I_3 \xrightarrow{b} I_4$, where:

$$I_4 := \{S \to .TU, T \to aTb., T \to .ab, U \to .c\}.$$

We not reduce P_2, going to $I_0 \xrightarrow{T} I_5$:

$$I_5 := \{S \to T.U, T \to .aTb, T \to .ab, U \to .c\}.$$

Shifting c we obtain $I_0 \xrightarrow{T} I_5 \xrightarrow{c} I_6$:

$$I_6 := \{S \to .TU, T \to .aTb, T \to .ab, U \to c.\}.$$

Reducing P_4, we end up at $I_0 \xrightarrow{T} I_5 \xrightarrow{U} I_7$:

$$I_7 := \{S \to TU., T \to .aTb, T \to .ab, U \to .c\}.$$

Reducing P_1 gives $I_0 \xrightarrow{S} I_8$, from which we can use P_0 and accept.
The inverse order of the rules that were applied is P_0, P_1, P_4, P_2, P_3; one can verify that this is indeed a rightmost derivation from S' to aabbc.

This automaton still has two weak points:

1. It is non-deterministic. Indeed when $\mathcal{A}'$ has the chance to apply a reduction, it can always decide to shift the next symbol instead. This is called a **shift/reduce conflict**. Moreover, $\mathcal{A}'$ may have the choice between two different reductions. This is called a **reduce/reduce conflict**. For instance, if P_3 was replaced by $T \to \varepsilon$, then a P_3-reduction could happen at any time during the run.

2. The item sets are unnecessarily large, increasing the size of the automaton. Intuitively, at the beginning of the run, it seems unnecessary to track the progress of production P_4, which can only intervene near the end. Likewise, near the end it seems unnecessary to track the progression of P_2 and P_3.

In Section 3, we will study how to address both weak points at the same time. Indeed, they are related: if we manage to track only "relevant" items, this automatically reduces the number of shift/reduce and reduce/reduce conflicts.

Before continuing, make sure that you have understood the concepts presented in this section, in particular the concept of a bottom-up parser, what *shift*, *reduce* and associated conflicts mean, and what items are. Indeed, armed with this knowledge you will already be largely operational to use *bison*. Indeed, that tool builds a parser following these concepts, and its states will be sets of items. These sets can be inspected with the option `-v`.

3 LR parsing

In this section, we will consider several variants of *deterministic* automata that implement bottom-up parsing. All of them accept only a subset of unambiguous grammars, and they represent different trade-offs between in how large a subset of grammars they can accept vs how much memory they need. The names of all these parsers contain the letters **LR**, standing for:

- **L**: the input is read from *left* to right;
- **R**: the parser produces a *rightmost* derivation.

All variants of LR parsers work with some form of *lookahead*: they can inspect the next k unconsumed symbols in the input before deciding what to do next. This behaviour can be simulated by a PDA: the PDA can read the k next symbols into its control state, then perform an ε-transition to simulate a shift or reduce. However, the notion of lookahead is more convenient. Other than that, LR parsers behave very similarly to the shift-reduce automaton from Section 2.3.

In practice, most parsers use a lookahead of $k = 1$. We shall discuss three such parsers, SLR, LR(1), and LALR. They represent different trade-offs in terms of the class of grammars they can handle and their memory requirements.

3.1 First and Follow

We first introduce some easy-to-understand concepts that are common to all parsers that we consider.

Definition 11 *Let $k \geq 0$ and $G = \langle \Sigma, V, P, S \rangle$ a grammar.*

- *For $w = a_1 \cdots a_l \in \Sigma^*$, let $First_k(w) := w$ if $l \leq k$ and $First_k(w) := a_1 \cdots a_k$ otherwise.*
- *For $L \subseteq \Sigma^*$, let $First_k(L) := \{ First_k(w) \mid w \in L \}$.*
- *For $\alpha \in (\Sigma \cup V)^*$, let $First_k(\alpha) := First_k(\mathcal{L}_G(\alpha))$.*

In other words, $First_k(\alpha)$ is the set of words up to length k that can be derived from α.

Example 12 *In the grammar from Example 1, we have*

$$First_2(E) := \{((, (\mathbf{int}, \mathbf{int}+, \mathbf{int}*, \mathbf{int}\}.$$

Definition 13 *Let $k \geq 0$, $G = \langle \Sigma, V, P, S \rangle$ a grammar, and $X \in V$. Then*

$$Follow_k(X) := \{ w \in \Sigma^* \mid \exists S' \to^* \gamma X \delta \wedge w \in First_k(\delta) \}.$$

Thus, $Follow_k(X)$ contains all the terminal words (up to length k) that may follow an occurrence of X in a derivation of G.

Example 14 *In the grammar from Example 1, we have $Follow_1(E) := \{\varepsilon,), +, *\}$.*

3.2 SLR parser

SLR stands for *Simple LR*. In general, this type of parser can work with a lookahead of any k symbols, denoted SLR(k); when k is not specified, it is assumed to be 1. In the following, we present the SLR(1) parser.

While being a bottom-up parser, SLR tries to identify 'useful' productions to track with a top-down approach: the goal of the parser is to apply the reduction $P_0 = S' \to S$, but in the beginning, S is not yet on the stack. Thus, one starts with the item $S' \to .S$ and then defines a closure operation:

Definition 15 *Let $I \subseteq Items(G)$. Then $clot(I) \subseteq Items(G)$ is the least $J \supseteq I$ satisfying the following condition:*

If $X \to \alpha.Y\beta \in J$, $Y \in V$, and $Y \to \gamma \in P$, then $Y \to .\gamma \in J$.

Example 16 *In Example 7, we have $clot(\{S' \to .S\}) = \{S' \to .S, S \to .TU, T \to .aTb, T \to .ab\}$. E.g., in order to obtain S', one must first obtain S, and therefore T. However, U is not useful in this context.*

One now defines the function *goto*, which is similar of δ from Proposition 5 but working only on items that are still interesting:

Definition 17 *Let $I \subseteq Items(G)$ and $z \in \Sigma \cup V$. Then*

$$goto(I, z) := clot(\{ X \to \alpha z.\beta \mid X \to \alpha.z\beta \in I \}).$$

Example 18 *Let $J := clot(\{S' \to .S\})$ from Example 16. Then*

$$goto(J, a) = \{T \to a.Tb, T \to a.b, T \to .aTb, T \to .ab\}.$$

Construction of the SLR parser

The states of an SLR parser are those that are reachable from the initial state $q_0 := clot(\{S' \to .S\})$ by means of *goto*. The parser assigns to each state q and to each possible lookahead $u \in \Sigma'$ a set of actions $actions(q, u)$. Those actions can be:

- **shift** (s): shift the next input symbol: if that symbol is $a \in \Sigma$, push $\langle q, a\rangle$ onto the stack and go to $goto(q, a)$.
- **reduce**$_{P_i}$ (r_i): apply the reduction for $P_i = X \to \alpha$ by removing α from the top of the stack, going back to some state q', then push $\langle q', X\rangle$ on the stack and $goto(q', X)$.
- **accept** (a): does what it says

More precisely:

- **shift** is in $actions(q, a)$ if $a \in \Sigma$ and q contains an item $X \to \alpha.a\beta$;
- **reduce**$_{X\to\alpha}$ is in $actions(q, u)$ if q contains $X \to \alpha.$, $u \in Follow_1(X)$ and $X \neq S'$;
- **accept** is in $actions(q, \varepsilon)$ if q contains $S' \to S$.

Definition 19 *A grammar G is said to be SLR if $|actions(q, u)| \leq 1$ for all reachable states q and lookaheads u.*

Example 20 *We construct the tables for actions and goto for the grammar from Example 7.*

state	*actions*				*goto*					
	a	b	c	ε	a	b	c	S	T	U
q_0	s				q_1			q_8	q_3	
q_1	s	s			q_1	q_2			q_4	
q_2		r_3	r_3							
q_3			s				q_5			q_7
q_4		s				q_6				
q_5				r_4						
q_6		r_2	r_2							
q_7				r_1						
q_8				a						

We have $Follow_1(S) = \{\varepsilon\}$, $Follow_1(T) = \{b, c\}$, $Follow_1(U) = \{\varepsilon\}$.
The states represent the following item sets:

- $q_0 := \{S' \to .S, S \to .TU, T \to .aTb, T \to .ab\}$
- $q_1 := \{T \to a.Tb, T \to a.b, T \to .aTb, T \to .ab\}$
- $q_2 := \{T \to ab.\}$
- $q_3 := \{S \to T.U, U \to .c\}$
- $q_4 := \{T \to aT.b\}$
- $q_5 := \{U \to c.\}$
- $q_6 := \{T \to aTb.\}$
- $q_7 := \{S \to TU.\}$
- $q_8 := \{S' \to S.\}$

Example 21 *Let us run the SLR parser on the input aabbc.*

$$q_0$$

$$\downarrow \textit{shift } a$$

$$q_0 \xrightarrow{a} q_1$$

$$\downarrow \textit{shift } a$$

$$q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_1$$

$$\downarrow \textit{shift } b$$

$$q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_1 \xrightarrow{b} q_2$$

$$\downarrow \textit{reduce } P_3 \ (T \rightarrow ab)$$

$$q_0 \xrightarrow{a} q_1 \xrightarrow{T} q_4$$

$$\downarrow \textit{shift } b$$

$$q_0 \xrightarrow{a} q_1 \xrightarrow{T} q_4 \xrightarrow{b} q_6$$

$$\downarrow \textit{reduce } P_2 \ (T \rightarrow aTb)$$

$$q_0 \xrightarrow{T} q_3$$

$$\downarrow \textit{shift } c$$

$$q_0 \xrightarrow{T} q_3 \xrightarrow{c} q_5$$

$$\downarrow \textit{reduce } P_4 \ (U \rightarrow c)$$

$$q_0 \xrightarrow{T} q_3 \xrightarrow{U} q_7$$

$$\downarrow \textit{reduce } P_1 \ (S \rightarrow TU)$$

$$q_0 \xrightarrow{S} q_8$$

$$\downarrow \textit{accept}$$

3.3 LR(1) parser

LR(1) parsers can handle a larger class of grammars than SLR. Like SLR, they do the parsing in linear time and with a lookahead of one character. The price to pay is a larger state table and a more complicated construction. One weak point of SLR is its reduction rule:

> $\mathbf{reduce}_{X\rightarrow\alpha}$ is in $\mathit{actions}(q,u)$ if q contains $X \rightarrow \alpha.$, $u \in \mathit{Follow}_1(X)$ and $X \neq S'$;

The lookahead u is compared to the characters that may follow X. Now, X may appear in multiple places in the grammar, and according to the context it appears in, different characters may follow. However, the SLR items ignore the context in which an item appears. This may cause unnecessary conflicts, as the following example shows.

Example 22 *Consider the following grammar G_2:*

$$P_0 : S' \to S \quad P_1 : S \to TTb \quad P_2 : S \to U \quad P_3 : T \to a \quad P_4 : U \to ab$$

*We have $Follow_1(T) = \{a, b\}$ and $Follow_1(S) = Follow_1(U) = \{\varepsilon\}$. The initial state of the SLR parser would be $q_0 := \{S' \to .S, S \to .TTb, S \to .U, T \to .a, U \to .ab\}$, and $goto(q_0, a) = \{T \to a., U \to a.b\} =: q_1$. Now, $actions(q_1, b)$ contains both **shift** (because of $U \to a.b$) and $\mathbf{reduce}_{T\to a}$ (because of $T \to a.$ and $b \in Follow_1(T)$). Therefore, G_2 is not SLR.*

The shift/reduce conflict in G_2 exists because the second T in $P_1 = S \to TTb$ is followed by b; but the item $T \to a.$ in q_1 corresponds to the first T. LR(1) parsers remember this context and hence allow for a more precise reduction rule.

Definition 23 *Let $G = \langle \Sigma, V, P, S' \rangle$ be a grammar. A* 1-item *of G is a tuple $[X \to \beta.\gamma, u]$ such that $X \to \beta\gamma \in P$ and $u \in \Sigma^{\leq 1}$. The set of 1-items of G is denoted $Items_1(G)$.*

Intuitively, a 1-item $[X \to \beta.\gamma, u]$ represents a situation where X appears in a derivation where it can be followed by u. The initial state of an LR(1) parser is therefore $clot(\{[S' \to .S, \varepsilon]\})$, where *clot* is defined as follows:

Definition 24 *Let $I \subseteq Items_1(G)$. Then $clot(I) \subseteq Items_1(G)$ is the least $J \supseteq I$ satisfying the following condition:*

> *If $[X \to \alpha.Y\beta, u] \in J$, $Y \to \gamma \in P$, and $v \in First_1(\beta u)$, then $[Y \to .\gamma, v] \in J$.*

Example 25 *We have $clot(\{[S' \to .S, \varepsilon]\}) = \{[S' \to .S, \varepsilon], [S \to .TTb, \varepsilon], [S \to .U, \varepsilon], [T \to .a, a], [U \to .ab, \varepsilon]\}$.*

The function *goto* is straightforward to adapt:

Definition 26 *Let $I \subseteq Items_1(G)$ and $z \in \Sigma \cup V$. Then*

$$goto(I, z) := clot(\{\, [X \to \alpha z.\beta, u] \mid [X \to \alpha.z\beta, u] \in I \,\}).$$

Construction of the LR(1) parser

We can now use 1-items to refine the reduction rule.

- **shift** is in $actions(q, a)$ if $a \in \Sigma$ and q contains an item $[X \to \alpha.a\beta, u]$;
- $\mathbf{reduce}_{X\to\alpha}$ is in $actions(q, u)$ if q contains $[X \to \alpha., u]$ and $X \neq S'$;
- **accept** is in $actions(q, \varepsilon)$ if q contains $[S' \to S., \varepsilon]$

Definition 27 *A grammar G is said to be LR(1) if $|actions(q,u)| \leq 1$ for all reachable states q and lookaheads u.*

Example 28 *We construct the LR(1) parse table for grammar G_2 from Example 22. For each state, an example stack content is given; the details of each state are given in the second table.*

state	*stack*	*actions*			*goto*				
		a	b	ε	a	b	S	T	U
q_0	ε	s			q_1		q_6	q_3	q_4
q_1	a	r_3	s			q_2			
q_2	ab			r_4					
q_3	T	s			q_5			q_7	
q_4	U			r_2					
q_5	Ta		r_3						
q_6	S			a					
q_7	TT		s			q_8			
q_8	TTb			r_1					

q_0	$[S' \to .S, \varepsilon], [S \to .TTb, \varepsilon], [S \to .U, \varepsilon], [T \to .a, a], [U \to .ab, \varepsilon]$
q_1	$[T \to a., a], [U \to a.b, \varepsilon]$
q_2	$[U \to ab., \varepsilon]$
q_3	$[S \to T.Tb, \varepsilon], [T \to .a, b]$
q_4	$[S \to U., \varepsilon]$
q_5	$[T \to a., b]$
q_6	$[S' \to S., \varepsilon]$
q_7	$[S \to TT.b, \varepsilon]$
q_8	$[S \to TTb., \varepsilon]$

No action contains more than two choices, so the language is LR(1).

3.4 LALR parser

The LALR parser is the most frequently implemented variant of bottom-up parsers; in particular it is supported by *bison*. It represents a compromise between the higher expressive power of LR(1) and the lower memory requirements of SLR. Its idea is very simple: One first generates the LR(1) parsing table. Then, one drops the lookaheads from all 1-items, effectively turning each state into a collection of items as in Definition 2. One then merges those lines of the parse table that have identical item sets.

Example 29 *Transforming the states from Example 28 yields the following:*

q_0	$[S' \to .S], [S \to .TTb], [S \to .U], [T \to .a], [U \to .ab]$
q_1	$[T \to a.], [U \to a.b]$
q_2	$[U \to ab.]$
q_3	$[S \to T.Tb], [T \to .a]$
q_4	$[S \to U.]$
q_5	$[T \to a.]$
q_6	$[S' \to S.]$
q_7	$[S \to TT.b]$
q_8	$[S \to TTb.]$

No two states are identical, so the parse table remains unchanged. In particular, the G_2 is LALR.

3.5 Extensions and relations

Let us briefly go through some results that are of a more theoretical interest; we will mention them without proof.

LR(1) parsing can be generalised to LR(k) parsing, i.e. with a lookahead of k characters ($k \geq 0$), in the obvious way: the states will be sets of k-items of the form $[X \to \alpha.\beta, u]$, where $u \in \Sigma^{\leq k}$, which can then be used to construct the parse table. A grammar is said to be LR(k) if its LR(k)-parse table contains no conflicts.

For any $k \geq 0$, there exists a grammar that is LR($k+1$) but not LR(k):

$$S \to ab^k c \mid Ab^k d, \qquad A \to a$$

We then have the following relationship between classes of grammars:

$$\text{LR}(0) \subseteq \text{SLR} \subseteq \text{LALR} \subseteq \text{LR}(1) \subseteq \text{LR}(2) \subseteq \cdots$$

The following characterization exists:

Proposition 30 *Let G be grammar and let $\to_r$ denote a rightmost derivation step in G. Then G is LR(k) if and only if any two derivations $S' \to_r^* \delta X w \to_r \delta\alpha w$ and $S' \to_r^* \gamma \to_r \delta\alpha w'$ with $First_k(w) = First_k(w')$ satisfy $\gamma = \delta X w$.*

By definition, an LR(k) grammar, for any $k \geq 0$, has a conflict-free parse table, and thus there exists a deterministic PDA accepting its language. Moreover, the standard translation of a PDA into context-free language yields an LR(1) grammar when the PDA is deterministic. Therefore, the class of *languages* generated by LR(k) grammars is the same for all $k \geq 1$. It is partially for this reason that parsers with a lookahead of more than one are rarely ever used in practice.

Bibliography

References

[1] Alfred V. Aho, Ravi Sethi et Jeffrey D. Ullman. Compilers: principles, techniques and tools. Addison-Wesley, 1986.

[2] Alfred V. Aho et Jeffrey D. Ullman. The theory of parsing, translation, and compiling. Volume I: Parsing. Prentice-Hall, 1972.

[3] John E. Hopcroft et Jeffrey D. Ullman. Introduction to automata theory, languages and computation. Addison-Wesley, 1979.