The LL($finite$) strategy for optimal LL(k) parsing

Peter Belcak

Mathematical Institute

University of Oxford, UK

peter.belcak@st-hildas.ox.ac.uk

Abstract

The LL($finite$) parsing strategy for parsing of LL(k) grammars where k needs not to be known is presented. The strategy parses input in linear time, uses arbitrary but always minimal lookahead necessary to disambiguate between alternatives of nonterminals, and it is optimal in the number of lookahead terminal scans performed. Modifications to the algorithm are shown that allow for resolution of grammar ambiguities by precedence – effectively interpreting the input as a parsing expression grammar – as well as for the use of predicates, and a proof of concept, the open-source parser generator Astir, employs the LL($finite$) strategy in the output it generates.

1 Related work

Despite their apparent simplicity and lesser power when compared to their LR counterparts, LL parsers of various types enjoy notable popularity in practice, as can be seen on the widespread use of LL-parser generators in the present day.

Perhaps the best understood class of grammars parsable from left to right while tracing a chain of left-most derivations are the LL(k) grammars. Since their introduction in [1], the case $k = 1$ of LL(k) grammars has been put into spotlight, mainly due to the reduced complexity and the ease of human-friendly implementation of LL(1) parsing in the form of a recursive-descent parser, as demonstrated early in [2] and elaborated on in [3]. Indeed, pioneer papers extending the theory of LL(k) parsing considered LL(1) "quite a bit simpler" [3] and "particularly attractive from a practical standpoint" [4], the latter posing the question of whether every LL(k) language is LL(1), hoping to simplify the construction of general LL(k) parsers. This was later proven false in general, unlike the corresponding LR(k) problem, although restricting to LL(k)-linear grammars the result has recently been shown true [5].

After their foundation in 1968, the work on LL(k) grammars briefly continued: [6] elaborated on their properties and [7] proved that LL($k + 1$) grammars are more powerful than LL(k). But shortly after, [8] gave proof of $LL(k) \subsetneq LR(k)$, which made the problem of parsing LL(k) grammars an LR(k)-problem and led to development of an LL(k) parsing process closely resembling the classical LR(k) parsing based on LR-items [9, 10].

On the parsing side, the LR-based [11] and the canonical fixed-lookahead-driven [12, 13] parsing remain the dominant methods for handling LL(k) grammars. More recently, variations of LL-inspired top-down parsers such as in [14] and [15] supporting syntactic and semantic predicates were developed, with the latter permitting backtracking if lookahead by a predicated deterministic finite automaton (DFA) fails.

Various other LL parsing strategies and parser construction algorithms have been proposed since [16, 17, 18], often disregarding the parse time in favour of extended front-end capabilities.

2 Motivation

We focus on the class of LL(k) algorithms for any k and aim to give a parser construction method that consistently yields efficient LL(k) linear-time parsers.

Following the notation from [1], let $G = (N, T, P, S)$ be a context-free grammar, where N is the nonterminal alphabet, T is the terminal alphabet, S is the starting symbol, and P is a set of productions of the form $A \to \boldsymbol{w}$ where A is in N and $\boldsymbol{w} = \gamma_1 \gamma_2 \dots \gamma_n$ in $X := (N \cup T)^*$. Let $\Rightarrow$ represent the transitive completion of $\to$, i.e., $\boldsymbol{w}_1 \Rightarrow \boldsymbol{w}_2$ if and only if $\boldsymbol{w}_2$ can be generated from $\boldsymbol{w}_1$ using productions in P. We denote empty strings of grammar symbols and empty sequences by ϵ. We say that a production $p \in P$ is an alternative of a nonterminal A if A is on the left-hand side of p.

Of the two dominant LL(k) parsing strategies, the LR-based parsing [11] is not commonly used in practice, whereas the tabular construction is standard [8]. The latter accepts strong-LL(k) grammars (to minimize the size of the resulting table) and makes extensive use of two functions: $FIRST_k$ and $FOLLOW_k$. $FIRST_k$ takes a string of grammar symbols α on input and returns the set of possible k-prefixes (initial strings of length at most k) of strings derivable from

α. $FOLLOW_k$ takes a nonterminal A on input and and returns the set of possible k-prefixes of strings that may follow the string derived from A, which combined with the prefixes returned by $FIRST_k$ guide the filling of the LL(k) parsing table.

Informally, the $FIRST_k$ function captures the naive LL(k) alternative disambiguation attempt: for $A \in N$ and $A \to \boldsymbol{\alpha} \in P$, $FIRST_k(\alpha)$ tries to retrieve possible input prefixes of length at most k that, when seen by the parser on encountering A and looking ahead in the parsing input, guarantee $A \to \boldsymbol{\alpha}$ is to be chosen to continue parsing. It is possible that $A \to \boldsymbol{\alpha} \Rightarrow \epsilon$ or that some of the prefixes in $FIRST_k(\alpha)$ are shorter than k, which would mean that the information returned by $FIRST_k$ might be insufficient for disambiguating between the productions considered. The strings in $FOLLOW_k(A)$ are thus concatenated in a cross-product fashion to the prefixes in $FIRST_k(k)$, and the result is trimmed to length k to give a list of k-prefixes to be used for production option disambiguation. If there are multiple alternatives of A that may lead to the same prefix being derived, the grammar suffers from an LL(k) ambiguity and an LL(k) parser cannot be built.

This approach has its drawbacks:

(a) The parameter for the amount of lookahead k needed must be known in advance.

(b) The construction of the "canonical" parsing table requires strong-LL(k) grammar, with the resulting table having $|T|^k$ columns and a total of $|N||T|^k$ entries. This space requirement quickly becomes unfeasible as k grows. Furthermore, in situations where parsable structures of variable length arrive in a continuing stream (over a network socket, say), the requirement that exactly k units of lookahead are always used can cause unnecessary delay in parsing due to having to wait for a continuation of the stream.

(c) For $k > 1$, the parser using the LL(k) parsing table has to perform overlapping lookahead look-ups under all circumstances. These look-ups are likely to be $O(1)$ for the simple use cases if the table is static in program memory, but more likely $\omega(1)$ in k for modern parsers with support for syntactic and semantic predicates (possibly recursive-descent) for which it is desirable to be programmer-friendly. This is especially costly if the type of the nonterminal is derived using a computationally expensive function, say by tracing the type in an OOP inheritance hierarchy, retrieving the lookahead from a slower memory device without caching, or due to an otherwise complex structure of the terminal object.

As an example, Grammar 1 is LL(5), and so the corresponding table would have at least $|N||T|^5$ entries,

$$
\begin{aligned}
A &\to abBc \\
A &\to abBd \\
B &\to bC \\
B &\to Cb \\
C &\to bc \\
C &\to bd
\end{aligned}
$$

Grammar 1: An LL(5) grammar

yet for the productions with B, C on the right-hand side it would be sufficient for the parser to look ahead by just 3. If separated into a sub-grammar containing the nonterminals B, C and their productions, the parsing table would have only $|N||T|^3$ entries. This demonstrates (point (b)) how easy it is for the canonical approach to lead to tables larger than necessary by orders of magnitude. Furthermore, whether the production $C \to bc$ or $C \to bd$ is to be parsed once C has been encountered can be decided simply by looking ahead at the second input terminal. The first input terminal does not have to be checked to be b as every reference to C in Grammar 1 already verifies whether b is in place when choosing between the alternatives of B. Similarly, looking ahead further by considering suffixes coming from $FOLLOW_3(C)$ is unnecessary – if any occurrence of a string recognized by C is followed by an invalid non-terminal (in this case any non-terminal different from b, c, d) it will be picked up on by the ensuing parsing process in the productions of parent nonterminals A, B. These demonstrate the drawbacks in points (c) and (b) respectively.

The LL($finite$) parsing strategy removes these drawbacks. The deterministic finite automata used to choose between alternatives of a nonterminal are minimal in the amount of lookahead used, and the generation mechanism also removes the drawback (c) by keeping stock of the already-verified input for lookahead when DFAs are being generated. As a result of the use of finite automata, the process of looking ahead to identify the production to be used is linear in the length of the transition path traced by the automaton (and thus $O(k)$), which is a disadvantage against the constant table look-up time of the canonical stack-and-table approach but no difference for the modern recursive-descent implementations. The removal of the drawback (c) comes at no additional cost.

3 The LL($finite$) parsing strategy

Whenever a nonterminal is to be parsed, the LL($finite$) parsing strategy decides which alternative to use by the output of a deterministic finite automaton. Multiple DFAs may be needed for the same nonterminal if the grammar parsed is not strong-LL(k) due to the additional context information determining the alternative, and thus we give two generative algorithms and two parsing behaviours, one for each case.

3.1 Definitions

For a string $w \in T^*$, denote the first k terminals of w (or the string itself if $|w| < k$) by w/k.

Consistently with [1] we say that a grammar $G = (N, T, P, S)$ is **LL(k)** if, for all $\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_2', \boldsymbol{w}_3, \boldsymbol{w}_3' \in T^*, A \in N, p, p' \in P$ such that $p = A \to \boldsymbol{\alpha}$, $p' = A \to \boldsymbol{\alpha}'$,

$$S \Rightarrow \boldsymbol{w}_1 A \boldsymbol{w}_3, S \Rightarrow \boldsymbol{w}_1 A \boldsymbol{w}_3', \boldsymbol{\alpha} \Rightarrow \boldsymbol{w}_2, \boldsymbol{\alpha}' \Rightarrow \boldsymbol{w}_2',$$
$$(\boldsymbol{w}_2 \boldsymbol{w}_3)/k = (\boldsymbol{w}_2' \boldsymbol{w}_3')/k \text{ implies } \boldsymbol{\alpha} = \boldsymbol{\alpha}'$$

We say that G is **strong-LL(k)** or **LLS(k)** if, for all $\boldsymbol{w}_1, \boldsymbol{w}_1', \boldsymbol{w}_2, \boldsymbol{w}_2', \boldsymbol{w}_3, \boldsymbol{w}_3' \in T^*, A \in N, p, p' \in P$ such that $p = A \to \boldsymbol{\alpha}$, $p' = A \to \boldsymbol{\alpha}'$,

$$S \Rightarrow \boldsymbol{w}_1 A \boldsymbol{w}_3, S \Rightarrow \boldsymbol{w}_1' A \boldsymbol{w}_3', \boldsymbol{\alpha} \Rightarrow \boldsymbol{w}_2, \boldsymbol{\alpha}' \Rightarrow \boldsymbol{w}_2',$$
$$(\boldsymbol{w}_2 \boldsymbol{w}_3)/k = (\boldsymbol{w}_2' \boldsymbol{w}_3')/k \text{ implies } \boldsymbol{\alpha} = \boldsymbol{\alpha}'$$

Define **LL(finite)-context** or just **context** to be any sequence of production-position pairs $\boldsymbol{\pi} = (p_1, i_1) \ldots (p_{l_{\boldsymbol{\pi}}}, i_{l_{\boldsymbol{\pi}}})$ where $p_1 \ldots p_{l_{\boldsymbol{\pi}}} \in P$ and $i_1 \ldots i_{l_v} \geq 1$. We will refer to $p_{l_{\boldsymbol{\pi}}}$ by $top(\boldsymbol{\pi})$, to $p_1 \ldots p_{l_{\boldsymbol{\pi}}-1}$ by $prev(\boldsymbol{\pi})$, and ϵ will stand for the empty context.

3.2 LLS(k) grammars

The algorithm building LL($finite$) parsers for strong-LL(k) grammars resembles the familiar $FIRST, FOLLOW$ approach more closely than the algorithm for general LL(k) grammars presented in Section 3.3 and requires less contextual bookkeeping. On the outset, it consists from three components: the $SUCC$, LA_A functions, and the CREATEDFA procedure. The $SUCC$ function (cf. 3.2.1) identifies all the nonterminal contexts in which the given nonterminal appears. The LA_A (cf. 3.2.2) retrieves the set of next lookahead characters for the specified sentential form and prefix. Finally, for a nonterminal A, CREATEDFA (cf. 3.2.3) constructs a deterministic finite automaton $\mathcal{M}_A$ that, when given input terminal sequence, determines which of the A's alternatives is to be used for parsing.

Equipped with $\{\mathcal{M}_A : A \in N\}$, Algorithm 2 simulates an LL($finite$) parser.

Successor terminal sequence identification

Define $SUCC : N \to \mathcal{P}(N \times X)$ by

$$A \mapsto \Big\{ (B, \gamma_{k+1} \ldots \gamma_n)$$
$$: P \ni B \to \gamma_1 \gamma_2 \ldots \gamma_n \text{ and } \gamma_k = A \Big\}.$$

In plain words, $SUCC$ takes a nonterminal symbol and yields a list of pairs $(B, \gamma_{k+1} \ldots \gamma_n)$, each characterizing an appearance of A by the "parent nonterminal" B (i.e. terminal that was on the right-hand side of the

$$\begin{aligned} A &\to aB \\ B &\to bCbc \\ C &\to c \\ D &\to ABC \end{aligned}$$

Grammar 2: An example grammar

production where A appeared) and the grammar symbols $\gamma_{k+1} \ldots \gamma_n$ that followed the occurrence of A in the production at position k.

Informally, $SUCC(E)$ gives the list of nonterminals in whose alternatives the parameter nonterminal E appears, combined with remainder of the production after its appearance. In a parser generator this function would be likely to be implemented as a simple retrieval from a map after such a map has been constructed while loading the individual productions into program's memory.

Consider Grammar 2. Then $SUCC(A) = \{(D, BC)\}$ due to the appearance of A in the production for D, $SUCC(B) = \{(D, C), (A, \epsilon)\}$ as B appears both in the productions for both A, D, $SUCC(C) = \{(B, bc)\}$ since C is referenced in the production for B and at the end of the production for D, and $SUCC(D) = \emptyset$ as there are no productions in the grammar that mention D on the right-hand side.

Sequential lookahead with LA_A

For $A \in N, \boldsymbol{\alpha} \in X^*, \boldsymbol{t} \in T^*$ define $LA_A(\boldsymbol{\alpha}, \boldsymbol{t})$ by the following recursive construction:

- If $\boldsymbol{\alpha} = \epsilon$, set $LA_A(\boldsymbol{\alpha}, \boldsymbol{t})$ to

$$\bigcup_{\substack{(B, \gamma_1 \ldots \gamma_n) \\ \in \\ SUCC(A)}} LA_B(\gamma_1 \ldots \gamma_n, \boldsymbol{t}).$$

- If $\boldsymbol{\alpha} = a \in T$ then
    - if $\boldsymbol{t} = \epsilon$ let $LA_A(\boldsymbol{\alpha}, \boldsymbol{t}) = \{a\}$,
    - if $\boldsymbol{t} = a$ let $LA_A(\boldsymbol{\alpha}, \boldsymbol{t}) = \{\epsilon\}$,
    - otherwise let $LA_A(\boldsymbol{\alpha}, \boldsymbol{t}) = \emptyset$.

- If $\boldsymbol{\alpha} = \alpha_1 \ldots \alpha_l$ is a sequence of grammar symbols, start with $LA_A(\boldsymbol{\alpha}, \boldsymbol{t}) = \emptyset$ and do the following:

    I. Add $LA_A(\alpha_1, \boldsymbol{t}) - \epsilon$ to $LA_A(\boldsymbol{\alpha}, \boldsymbol{t})$.

    II. If $\boldsymbol{t} \neq \epsilon$, add

    $$LA_A(\alpha_2 \ldots \alpha_l, t_2 \ldots t_m) \text{ to } LA_A(\boldsymbol{\alpha}, \boldsymbol{t}).$$

    III. If $LA_A(\alpha_1, \boldsymbol{t}) \ni \epsilon$, add

    $$LA_A(\alpha_2 \ldots \alpha_l, \boldsymbol{t}) \text{ to } LA_A(\boldsymbol{\alpha}, \boldsymbol{t}).$$

- If $\boldsymbol{\alpha} = A \in N$, then for every $A \to \boldsymbol{\gamma} \in P$, add $LA_A(\boldsymbol{\gamma}, \boldsymbol{t})$ to $LA_A(\boldsymbol{\alpha}, t)$. If A is also the start nonterminal of G and $LA_A(\boldsymbol{\gamma}, \boldsymbol{t}) = \emptyset$, add $\dashv$ to $LA_A(\boldsymbol{\alpha}, \boldsymbol{t})$, denoting an "end of file" terminal that is expected in such a situation.

Thus, for Grammar 2, $LA_A(A, \epsilon) = \{a\}$, $LA_B(bCbc, b) = \{c\}$, $LA_D(ABC, abcbcb) = \{c\}$.

Lookahead DFA construction

With the LA_A defined we now give the algorithm that gives automata choosing productions to be parsed. Algorithm 1 simply looks at every possible unordered pair of alternatives of a nonterminal and steps through lookaheads permissible by the productions depth-first, building an acyclic DFA in the process. A recursive branch of the algorithm terminates whenever it can be said with certainty which alternative is to be parsed, given a lookahead sequence.

Algorithm 1 DFA construction for $G \in LLS(k)$

1: **procedure** CREATEDFA(A)
2:     create a new empty DFA $\mathcal{M}_A$
3:     add a new state ρ to $\mathcal{M}_A$
4:     BUILDDFA(A, ρ, ϵ)
5:     **return** $\mathcal{M}_A$
6: **end procedure**

7: **procedure** BUILDDFA(A, s_0, $\boldsymbol{t}$)
8:     let $p_1 \ldots p_n$ be the alternatives of A
9:     **for** $1 \leq i \leq n$ **do**
10:         **for** $1 \leq j \leq i$ **do**
11:             suppose $p_i = A \to \alpha_i$, $p_j = A \to \alpha_j$
12:             let $l_i := LA_A(\alpha_i, \boldsymbol{t})$, $l_j := LA_A(\alpha_j, \boldsymbol{t})$
13:             let $intersection := l_i \cap l_j$
14:             **for all** terminals $a \in l_i \cup l_j$ **do**
15:                 add state s to $\mathcal{M}_A$
16:                 add to $\mathcal{M}_A$ transition $e : s_0 \to s$
17:                 condition e on a
18:                 **if** $a \in intersection$ **then**
19:                     BUILDDFA($A, s, \boldsymbol{t}a$)
20:                 **else if** $a \in l_i$ **then**
21:                     label s by p_i
22:                     mark s as final
23:                 **else if** $a \in l_j$ **then**
24:                     label s by p_j
25:                     mark s as final
26:                 **end if**
27:             **end for**
28:         **end for**
29:     **end for**
30: **end procedure**

Algorithm 1 consists of the two procedures CREATEDFA and BUILDDFA. CREATEDFA creates a new finite automaton $\mathcal{M}_A$ for the nonterminal A and calls BUILDDFA to build on it. BUILDDFA takes as parameters a nonterminal A, a state s_0 of $\mathcal{M}_A$, and a terminal

$p_1 = S \to A$
$p_2 = S \to B$
$A \to aA$
$A \to x$
$B \to aB$
$B \to y$
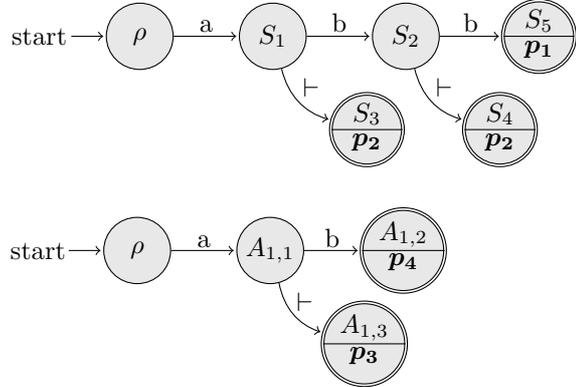
Grammar 3: An LLR [19] grammar.

Figure 1: The DFAs produced for nonterminals S and A_1 of Grammar 5.

string $\boldsymbol{t}$ representing the lookahead prefix for which a sub-DFA is to be built. It extends $\mathcal{M}_A$ by building the sub-DFA for the prefix $\boldsymbol{t}$ and attaching it to the state s_0.

It is easy to see that for a nonterminal A the machine M_A produced by CREATEDFA is a DFA, that M_A is *acyclic*, and that every final state of the automaton is marked by some alternative of A.

Figure 1 gives the DFA produced by CREATEDFA for the nonterminals S and A_1 of Grammar 5. The DFA produced for the nonterminal A_2 would be functionally identical to the DFA for A_1.

Clearly, Algorithm 1 is at the risk of not terminating when applied to nonterminals of a grammar that is not $LLS(k)$. A pathological example is Grammar 3. In this grammar it is impossible to determine whether p_1 or p_2 is to be used for all prefixes $a^k, k \geq 0$ – the last character must be seen first. If the procedure goes on unstopped it will continue generating the DFA indefinitely. In Section 4.1 we discuss how the DFA construction algorithm can be terminated in a way that still leads to a reasonable choice of alternative to be used.

LL($finite$) parser simulation

A parser for grammar G making use of the DFAs $\{\mathcal{M}_A : A \in N(G)\}$ would proceed as follows: upon being instructed to parse nonterminal A (either because A is the start symbol or due to A appearing in some production of G that is being parsed), the parser will execute the DFA $\mathcal{M}_A$ scanning ahead the remainder

Algorithm 2 LL($finite$) parsing for $G \in LLS(k)$

Require: global input terminal string $\mathcal{I}$
1: **procedure** PARSE(G)
2:     PARSENONTERMINAL(S)
3: **end procedure**

4: **procedure** PARSENONTERMINAL(A)
5:     execute $\mathcal{M}_A$ against $\mathcal{I}$ without consumption
6:     $p \leftarrow$ the label of the final state $\mathcal{M}_A$ is in
7:     suppose $p = A \rightarrow \alpha_1 \ldots \alpha_{l_\alpha}$
8:     **for** $1 \leq i \leq l_\alpha$ **do**
9:         **if** $\alpha_i \in T$ **then**
10:           **if** $\alpha_i = \mathcal{I}_1$ **then**
11:             CONSUME
12:           **else**
13:             ERROR
14:           **end if**
15:         **else**   ▷ $\alpha_i \in N$
16:           PARSENONTERMINAL(α_i)
17:         **end if**
18:     **end for**
19:     ANNOUNCE(A)
20: **end procedure**

of the parsing input without consuming it. The label of the final state reached gives the alternative of A to be used for further parsing. Algorithm 2 shows pseudocode for behaviour of a program that simulates LL($finite$) parsing of an LLS(k) grammar. It illustrates how an LL($finite$) parser makes use of the DFAs generated by Algorithm 1. The procedure PARSE accepts an LLS(k) grammar G. When called, it proceeds to parse the input $\mathcal{I}_1 \ldots \mathcal{I}_n = \mathcal{I} \in T^*$ against the rules of G, beginning with the start symbol S. Procedure ERROR is placeholder for an error recovery, and CONSUME simply moves the input $\mathcal{I}$ forward by discarding $\mathcal{I}_1$ and re-labeling $\mathcal{I}_2$ to $\mathcal{I}_1$, $\mathcal{I}_3$ to $\mathcal{I}_2$, etc.

One possible parser implementation is to store a map with nonterminals representing the keys and the corresponding DFAs being the values. Another, likely a more efficient option also applicable in generation of recursive-descent parsers, is to translate each DFA into a conditional expression or a sequence of conditional expressions to be used by nested `if-else` statements. This is always possible as the DFAs are acyclic, and such are the output parsers produced by the generator Astir [20].

3.3 General LL(k) grammars

The construction given in Section 3.2 can also be applied to modifications of general LL(k) grammars. Consider Grammar 4 with the alternatives of S appended by the new terminal $\dashv$ as in 3.2.2. This grammar is LL(3) but not LLS(k) for any $k \geq 1$. In partic-

$$p_1 = S \rightarrow Abb$$
$$p_2 = S \rightarrow A$$
$$p_3 = A \rightarrow a$$
$$p_4 = A \rightarrow ab$$

Grammar 4: An LL(3) grammar that is not LLS(k) for any $k \geq 1$.

$$p_1 = S \rightarrow A_1 bb$$
$$p_2 = S \rightarrow A_2$$
$$p_3 = A_1 \rightarrow a$$
$$p_4 = A_1 \rightarrow ab$$
$$p_5 = A_2 \rightarrow a$$
$$p_6 = A_2 \rightarrow ab$$

Grammar 5: The Grammar 4 modified to be LLS(3).

ular,

$$FIRST_3(a \ FOLLOW_3(A)) \cap$$
$$FIRST_3(ab \ FOLLOW_3(A)) = \{ab \dashv\} \neq \emptyset,$$

and similarly for $k > 3$. By introducing new nonterminals A_1, A_2 and copies of their alternatives per alternative of S, Grammar 4 is converted to Grammar 5 that is LLS(3). This process can be generalized (care must be taken when a nonterminal recursively refers to itself), and it is equivalent to the procedure in this section in the number of DFAs that need to be generated for parsing of the starting LL(k) grammar. An advantage of the direct approach shown in Algorithm 3 is that it permits optimisation of lookahead sequence overlaps as explained in point (c) of Section 2.

Our direct approach to general LL(k) grammar parsing consists of three principal components: the modified sequential lookahead function LA_π, optimized lookahead DFA construction algorithms CREATECDFA and BUILDCDFA, and a new PARSENONTERMINAL procedure. DFAs are constructed per unique nonterminal appearance rather than per nonterminal alone, and the optimised construction process is given in Algorithms 3, 5, 4.

Modified sequential lookahead with LA_π

To track the production context within which a nonterminal appeared we use the LL($finite$) context $\boldsymbol{\pi}$ (cf. Section 3.1). The tails of production strings starting after occurrences of nonterminals will be of particular interest. For $A \rightarrow \alpha_1 \ldots \alpha_{l_\alpha} = p \in P$ and $1 \leq i \leq l_\alpha$, define $tail(p, i) = \alpha_{i+1} \ldots \alpha_{l_\alpha}$ (i.e. all the grammar symbols of string $\boldsymbol{\alpha}$ following the position i). For $i = l_\alpha$, $tail(p, i) = \epsilon$.

For $\boldsymbol{\pi}$ a context, $\boldsymbol{\alpha} \in X^*$, p the production currently being considered, i the position of α_1 in p, and $\boldsymbol{t} \in T^*$, let $\boldsymbol{\pi}^+$ be $\boldsymbol{\pi}, (p, i)$ if $\alpha_1 \in N$ and $\boldsymbol{\pi}$ otherwise. Then define $LA_{\boldsymbol{\pi}}(\boldsymbol{\alpha}, \boldsymbol{t})$ by the following recursive construction:

- If $\boldsymbol{\alpha} = \epsilon$, then if $prev(\pi) \neq \epsilon$ set, $LA_\pi(\boldsymbol{\alpha}, \boldsymbol{t})$ to $LA_{prev(\boldsymbol{\pi})}(tail(top(\boldsymbol{\pi})), \boldsymbol{t})$, otherwise set it to $\emptyset$.

- If $\boldsymbol{\alpha} = a \in T$ then

  - if $\boldsymbol{t} = \epsilon$ let $LA_\pi(\boldsymbol{\alpha}, t) = \{a\}$,

  - if $\boldsymbol{t} = a$ let $LA_\pi(\boldsymbol{\alpha}, t) = \{\epsilon\}$,

  - otherwise let $LA_\pi(\boldsymbol{\alpha}, \boldsymbol{t}) = \emptyset$.

- If $\boldsymbol{\alpha} = \alpha_1 \ldots \alpha_{l_\alpha}$ is a sequence of grammar symbols, start with $LA_\pi(\boldsymbol{\alpha}, t) = \emptyset$ and do the following:

  I. Add $LA_{\boldsymbol{\pi}^+}(\alpha_1, \boldsymbol{t}) - \epsilon$ to $LA_\pi(\boldsymbol{\alpha}, \boldsymbol{t})$.

  II. If $\boldsymbol{t} \neq \epsilon$, add

  $$LA_\pi(\alpha_2 \ldots \alpha_{l_\alpha}, t_2 \ldots t_m) \text{ to } LA_\pi(\boldsymbol{\alpha}, \boldsymbol{t}).$$

  III. If $LA_{\boldsymbol{\pi}^+}(\alpha_1, \boldsymbol{t}) \ni \epsilon$, add

  $$LA_\pi(\alpha_2 \ldots \alpha_l, t) \text{ to } LA_\pi(\boldsymbol{\alpha}, \boldsymbol{t}).$$

- If $\boldsymbol{\alpha} = A \in N$, then for every $A \to \boldsymbol{\gamma} = p \in P$, add $LA_\pi(\boldsymbol{\gamma}, \boldsymbol{t})$ (remembering p as the production being considered) to $LA_\pi(\boldsymbol{\alpha}, \boldsymbol{t})$. If A is also the start nonterminal of G and $LA_\pi(\boldsymbol{\gamma}, \boldsymbol{t}) = \emptyset$, add $\dashv$ to $LA_\pi(\boldsymbol{\alpha}, \boldsymbol{t})$ as in Section 3.2.2.

For Grammar 4, $LA_\epsilon(S, \epsilon) = \{a\}$, $LA_\epsilon(S, a) = \{b\}$, $LA_{(p_4,1)}(A, ab) = \{b\}$, but $LA_\epsilon(A, ab) = \{\dashv\}$. This is in contrast with $LA_A(A, ab) = \{b, \dashv\}$ used for $LLS(k)$ grammars.

Optimized lookahead DFA construction

The CREATEDFA and BUILDDFA procedures in Algorithm 1 can be easily modified to produce a DFA for use with a $LL(k)$ grammar instead. The procedure BUILDDFA adjusted for the general case takes an $LL(finite)$ context $\boldsymbol{\pi}$ on input instead of a nonterminal A, works on and produces the DFA $\mathcal{M}_\pi$ rather than $\mathcal{M}_A$, and uses the modified lookahead function LA_π wherever 1 uses LA_A. We refer to the new procedures by CREATECDFA (i.e. "create contextual DFA") and BUILDCDFA respectively.

The DFAs in Figure 1 for Grammar 5 produced by Algorithm 1 coincide with the DFAs produced by CREATECDFA for Grammar 4 in everything but labels. In particular, the labels A_1, A_2 are replaced by $\boldsymbol{\pi}_1 = (p_1, 1), \boldsymbol{\pi}_2 = (p_2, 1)$ respectively.

When a DFA $\mathcal{M}_\pi$ matches an input prefix $\mathcal{I}_1 \ldots \mathcal{I}_m$ for some $m \leq k$ to choose an alternative p of a nonterminal A, we say that $\mathcal{M}_\pi$ *verifies* $\mathcal{I}_1 \ldots \mathcal{I}_m$ for p. Turning this around, given DFA $\mathcal{M}_\pi$ constructed by CREATECDFA and alternative p of A, we can find the **verified lookahead** of $\mathcal{M}_\pi$ for p – that is, the maximal prefix $\mathcal{I}_1 \ldots \mathcal{I}_m$ that can certainly be found on $\mathcal{I}$ after $\mathcal{M}_A$ chose p. Formally, the verified lookahead of $\mathcal{M}_A$ is the string of transition condition terminals on

the longest path connecting the root of $\mathcal{M}_\pi$ and a state s such that there exist no two disjoint directed paths from s to leaves of $\mathcal{M}_\pi$ labeled by p.

It was observed in Section 3.2.3 that the resulting machine uses only the minimum amount of lookahead necessary to choose an alternative for parsing at all times. To avoid unnecessary lookahead verification overlaps one needs to consider the wider production circumstances, remembering how much of the alternative p within which the DFA for given nonterminal has already been scanned and verified when choosing p over other productions of the same parent nonterminal.

Algorithm 5 shows how a set of DFAs indexed by $LL(finite)$ contexts is built for parsing of $LL(k)$ grammar G. In practice, say if a recursive-descent parser were to be generated, such a set (map) would not be explicitly constructed by the parser generator, and the indices characterizing when the DFA is to be used would be implicit in the stack information of the parser.

Algorithm 3 DFA construction core for $G \in LL(k)$

Require: the set $contextMachines$, the set $variants$
1: **procedure** BUILDSUBCONTEXTS($\boldsymbol{\pi}$, $\boldsymbol{t}$)
2:     let $(p, i) = top(\boldsymbol{\pi})$
3:     let A be the nonterminal at position i in p
4:     $\mathcal{M}_\pi \leftarrow$ CREATEODFA($\boldsymbol{\pi}$)
5:     $\mathcal{M}'_\pi \leftarrow$ REMOVEREDUNDANT($\mathcal{M}_A, \boldsymbol{t}$)
6:     **if** $variants$ does not contain (A, M_A) **then**
7:         add $(A, \mathcal{M}'_\pi)$ to $variants$
8:         add $\mathcal{M}'_\pi$ to $contextMachines$
9:         **for all** alternatives p of A **do**
10:             $\ell \leftarrow$ the verified lookahead of $\mathcal{M}_\pi$ for p_i
11:             suppose $p = A \to \alpha_1 \ldots \alpha_{l_\alpha}$
12:             **for** $1 \leq i \leq l_\alpha$ **do**
13:                 **if** $\ell \neq \epsilon$ **then**
14:                     remove the leading symbol from ℓ
15:                 **end if**
16:                 **if** α_i is a nonterminal **then**
17:                     BUILDSUBCONTEXTS($\boldsymbol{\pi}(p, i), \ell$)
18:                 **end if**
19:             **end for**
20:         **end for**
21:     **end if**
22: **end procedure**

Algorithm 5 will terminate as there is only a finite number of DFAs per nonterminal that can be produced by BUILDCDFA.

4 Extensions

4.1 Resolution of LL(k) ambiguities

When the execution reaches line 1.18 of Algorithm 1 more than some threshold κ times, the parser generator

Algorithm 4 DFA optimisation for $G \in LL(k)$

1: **procedure** REMOVEREDUNDANT($\mathcal{M}, \boldsymbol{t}$)
2:     $current \leftarrow$ the root state of $\mathcal{M}$
3:     **loop**
4:         **if** $\boldsymbol{t} = \epsilon$ **then**
5:             **break**
6:         **end if**

7:         let E be the out-transitions of $current$
8:         **if** $|E| \neq 1$ **then**
9:             **break**
10:        **else**
11:            suppose $E = \{current \rightarrow next\}$
12:            $current \leftarrow next$
13:            remove the leading symbol from $\boldsymbol{t}$
14:        **end if**
15:     **end loop**
16:     **return** the sub-DFA of $\mathcal{M}$ starting at $current$
17: **end procedure**

Algorithm 5 DFA map construction

1: **procedure** BUILDCONTEXTMACHINES(G)
2:     initialize $contextMachines$ to empty map
3:     initialize $variants$ to empty set
4:     add S' to N
5:     add $p_{S'} = S' \rightarrow S$ to p
6:     BUILDSUBCONTEXTS($(p_{S'}, 1), \epsilon$)
7: **end procedure**

---

might want to consider stopping the DFA construction and attempt to resolve the $LL(k)$ ambiguity of the two productions by precedence.

In particular, if the DFA construction depth κ has been exceeded, the algorithm may want to choose to label s by $p_{\min\{i,j\}}$ and mark it as final instead of calling the procedure BUILDDFA again.

This behaviour is equivalent to the one of a parser built to accept a parsing expression grammar [21]. Another, more sophisticated option that permits parsing of the usual nested arithmetic expressions, is to give an explicit order in which productions are to be considered for some terminals a that may be encountered, attempt to scout for such symbols after a soft lookup depth constraint κ_1 has been exceeded, and eventually resort to precedence resolution if a hard depth constraint $\kappa_2 > \kappa_1$ has been exceeded without encountering any of the symbols that would lead to a resolution.

4.2 Semantic and syntactic predicates

The procedure BUILDDFA can be readily extended to support the use of negative semantic predicates by considering every predicate a terminal. The transitions with terminals representing the negative predicates would not consume lookahead and would be omitted in procedure BUILDSUBCONTEXTS in the steps rel-

---

Algorithm 6 PARSENONTERMINAL for $G \in LL(k)$

Require: pre-built set of machines $\mathcal{M}_{\boldsymbol{\pi}}, \mathcal{M}'_{\boldsymbol{\pi}}$ for every relevant context $\boldsymbol{\pi}$
1: **procedure** PARSENONTERMINAL($\boldsymbol{\pi}$)
2:     $\ell \leftarrow$ verified lookahead of $\mathcal{M}_{\boldsymbol{\pi}}$
3:     execute $\mathcal{M}'_{\boldsymbol{\pi}}$ against $\mathcal{I}$ without consumption
4:     $p \leftarrow$ the label of the final state $\mathcal{M}'_{\boldsymbol{\pi}}$ is in

5:     suppose $p = A \rightarrow \alpha_1 \ldots \alpha_{l_{\boldsymbol{\alpha}}}$
6:     $sawNonterminal \leftarrow false$
7:     **for** $1 \leq i \leq l_{\boldsymbol{\alpha}}$ **do**
8:         **if** $\alpha_i \in T$ **then**
9:             **if** $\ell \neq \epsilon$ and not $sawNonterminal$ **then**
10:             remove the leading symbol of ℓ
11:           **else if** $\alpha_i = \mathcal{I}_1$ **then**
12:             CONSUME
13:           **else**
14:             ERROR
15:           **end if**
16:         **else** $\triangleright \alpha_i \in N$
17:            $sawNonterminal \leftarrow true$
18:            PARSENONTERMINAL(α_i)
19:         **end if**
20:     **end for**
21:     ANNOUNCE(A)
22: **end procedure**

---

evant to root optimisation. A positive semantic predicate can then be represented by double negation as in [21] for PEGs.

Syntactic predicates then be eliminated by conversion to semantic predicates as in [15].

4.3 Right-linear grammar rules

LL-regular grammars [19] are a strict superset of $LL(k)$ grammars and can be parsed in linear time [22] but face the obstacle of the undecidability of the problem of identifying regular partitions [23]. The $LL(*)$ parsing strategy [15] can attempt to parse a regular grammar but faces the very real risk of exponential parsing time due to extensive use of backtracking.

One is able to extend BUILDDFA to add back-transitions (transitions towards the root) to permit the parsing right-linear grammar rules in addition to the $LL(k)$ constraint on the remainder of the grammar rules, all in linear time.

5 Conclusion

The $LL(finite)$ parsing strategy gives an optimal way of parsing $LL(k)$ grammars in both the amount of lookahead needed at any point and avoidance of superfluous lookahead verification. This comes with the practical convenience of not having to know k in advance, acceptance of general $LL(k)$ grammars, and

minimality of the parsing table (if parsing table is to be generated). Furthermore, LL($finite$) parsers are readily extensible for support of syntactic and semantic predicates, allow for use of traditional LL error recovery procedures, and can be represented in an easily interpretable programmer-friendly recursive-descent form, as is demonstrated by the generation mechanism and output of the Astir parser generator [20].

References

[1] P. M. Lewis and R. E. Stearns, "Syntax-directed transduction," *Journal of the ACM (JACM)*, vol. 15, no. 3, pp. 465–488, 1968.

[2] D. Schorre, "Meta ii – a syntax-oriented compiler writing language," in *Proceedings of the 1964 19th ACM national conference*, 1964, pp. 41–301.

[3] D. E. Knuth, "Top-down syntax analysis," *Acta Informatica*, vol. 1, no. 2, pp. 79–110, 1971.

[4] D. B. Johnson and R. Sethi, "A characterization of ll(1) grammars," *BIT Numerical Mathematics*, vol. 16, no. 3, pp. 275–280, 1976.

[5] A. Okhotin and I. Olkhovsky, "On the transformation of ll(k)-linear grammars to ll(1)-linear," in *International Computer Science Symposium in Russia*. Springer, 2020, pp. 328–340.

[6] R. Kurki-Suonio, "Notes on top-down languages," *BIT Numerical Mathematics*, vol. 9, no. 3, pp. 225–238, 1969.

[7] D. J. Rosenkrantz and R. E. Stearns, "Properties of deterministic top-down grammars," *Information and Control*, vol. 17, no. 3, pp. 226–256, 1970.

[8] A. Aho, "Ullman. jd: The theory of parsing, translation and compiling, vol. i and vol. ii," 1973.

[9] S. Sippu and E. Soisalon-Soininen, "On constructing ll(k) parsers," in *International Colloquium on Automata, Languages, and Programming*. Springer, 1979, pp. 585–595.

[10] ——, "On ll(k) parsing," *Information and Control*, vol. 53, no. 3, pp. 141–164, 1982.

[11] ——, *Parsing Theory: Volume II LR(k) and LL(k) Parsing*. Springer Science & Business Media, 1990, vol. 20.

[12] T. H. Sharafy, "Jetpag - home," 2007. [Online]. Available: http://jetpag.sourceforge.net/index.php

[13] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[14] B. Ford, "Packrat parsing: simple, powerful, lazy, linear time, functional pearl," *ACM SIGPLAN Notices*, vol. 37, no. 9, pp. 36–47, 2002.

[15] T. Parr and K. Fisher, "Ll(*) the foundation of the antlr parser generator," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 425–436, 2011.

[16] T. Parr, S. Harwell, and K. Fisher, "Adaptive ll(*) parsing: the power of dynamic analysis," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 579–598, 2014.

[17] A. Afroozeh and A. Izmaylova, "Faster, practical gll parsing," in *Compiler Construction*, B. Franke, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 89–108.

[18] B. Slivnik, "LLLR Parsing: a Combination of LL and LR Parsing," in *5th Symposium on Languages, Applications and Technologies (SLATE'16)*, ser. OpenAccess Series in Informatics (OASIcs), M. Mernik, J. P. Leal, and H. G. Oliveira, Eds., vol. 51. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 5:1–5:13. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6010

[19] S. JARZABEK and T. Krawczyk, "Ll-regular grammars." *Instytutu Maszyn Matematycznych*, p. 107, 1974.

[20] P. Belcak, 2020. [Online]. Available: https://astir.dev

[21] B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2004, pp. 111–122.

[22] A. Nijholt, "On the parsing of ll-regular grammars," in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 1976, pp. 446–452.

[23] J. E. Hopcroft and J. D. Ullman, *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.