Technical Appendix: Accelerating LLM Code Generation Through Mask Store Streamlining

Vivien Tran-Thien

January 2025

Abstract

This document formalizes algorithms mentioned in the blog post titled *Accelerating LLM Code Generation Through Mask Store Streamlining* [3] and provides proofs of their correctness.

Contents

1 Notation

With the following notation:

- G is a context-free grammar;
- Σ is the set of terminals of G;
- L is the context-free language associated with G

- L_p is the set of prefixes of L
  (i.e. $L_p \stackrel{\text{def}}{=} \{P \in \Sigma^*, \exists S \in \Sigma^*, PS \in L\}$)

... we define three functions:

$$\begin{aligned}
\texttt{is_always_legal}&: \Sigma \times \Sigma^* && \to \{\texttt{True}, \texttt{False}\} \\
&\quad (X, S) && \mapsto (\forall P \in \Sigma^*, PX \in L_p \implies PXS \in L_p)
\end{aligned}$$

$$\begin{aligned}
\texttt{is_never_legal}&: \Sigma \times \Sigma^* && \to \{\texttt{True}, \texttt{False}\} \\
&\quad (X, S) && \mapsto (\forall P \in \Sigma^*, PXS \notin L_p)
\end{aligned}$$

$$\begin{aligned}
\texttt{are_jointly_legal}&: \Sigma \times \Sigma^* \times \Sigma^* && \to \{\texttt{True}, \texttt{False}\} \\
&\quad (X, S_1, S_2) && \mapsto (\forall P \in \Sigma^*, PXS_1 \in L_p \iff PXS_2 \in L_p)
\end{aligned}$$

2 An algorithm for `is_never_legal`

Given its definition, computing `is_never_legal` is equivalent to determining whether the intersection of L and $\Sigma^* X S \Sigma^*$ is empty. Since $\Sigma^* X S \Sigma^*$ is a regular language, this is straightforward, thanks to standard CFG algorithms to compute the intersection of a context-free language and a regular language [1] and to check whether the resulting context-free language is empty [2].

Algorithm 1: `is_never_legal`

Input: $X \in \Sigma, S \in \Sigma^*, L$ context-free language
Output: `True` or `False`

```
1 function is_never_legal(X, S, L)
2     return is_empty(L ∩ (Σ*XSΣ*))
```

3 `is_always_legal` is not computable

We first demonstrate that `is_always_legal` is in general not computable before showing how to obtain `is_always_legal`(X, Y) in some cases.

Proposition 1. `is_always_legal` is not computable for an arbitrary context-free grammar.

Proof. We denote L_1 and L_2 as context-free languages, as well as $L = L_1 X \cup L_2 X Y$ with $X, Y \in \Sigma$, $X \neq Y$ and X not part of the alphabets of L_1 and L_2. We prove that:

$$\texttt{is\_always\_legal}(X, (Y,)) = \texttt{True} \iff L_1 \subset L_2$$

Assume that `is_always_legal`$(X, (Y,)) =$ `True` and let $P \in L_1$. Since $L_1X \subset L$, $PX \in L$ and, by definition of `is_always_legal`, $PXY \in L$ and then $P \in L_2$. Otherwise said, $L_1 \subset L_2$.
Now, assume that $L_1 \subset L_2$ and let $P \in \Sigma^*$ such that $PX \in L_p$. Since X is not part of the alphabets of L_1 and L_2, this means that $P \in L_1$ or $P \in L_2$. Given that $L_1 \subset L_2$, $P \in L_2$ in all cases, and therefore $PXY \in L$ and $PXY \in L_p$. We can then conclude that `is_always_legal`$(X, (Y,)) =$ `True`.
Now that the equivalence is proven, the result follows from the undecidability of determining whether a context-free language is included in another context-free language [2]. □

4 Obtaining some values of `is_always_legal`

4.1 The DFT pushdown automaton

Since systematically computing `is_always_legal` is beyond our reach, we now aim to compute its values in specific cases. In this context, it is relevant to understand which symbols can follow another symbol and to represent these relationships with a directed graph, as illustrated in Figure 1.

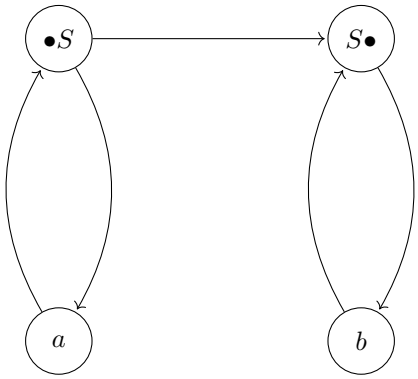


Figure 1: A directed graph showing the direct successor relationship for the grammar corresponding to the $S ::= aSb|\epsilon$ rule. $\bullet S$ and $S\bullet$ respectively represent the start and the end of non-terminal S.

However, such a directed graph is generally not rich enough to draw conclusions about the values of `is_always_legal`. To go a step further, we extend this representation to not only track the successor relationships among symbols but also the rules from which these relationships arise. More precisely, we introduce the following pushdown automaton.

Definition 1. For a grammar G, we define its *DFT pushdown automaton* (where *DFT* stands for depth-first traversal) as follows:

- The set of states Q is $\Sigma \cup \{\bullet X_1, X_1\bullet, ..., \bullet X_n, X_n\bullet, \texttt{\$END}\}$ where:

  - Σ is the set of terminals of G;
  - $\bullet X_1$, $X_1\bullet$, ..., $\bullet X_n$, $X_n\bullet$ are new symbols derived from the non-terminals X_1, ..., X_n of G ($\bullet X$ and $X\bullet$ respectively represent the "start" and the "end" of non-terminal X);
  - `$END` is a new symbol;

  The states represent terminal or nonterminal symbols being generated.

- The initial state is $\bullet S$ where S is the start symbol of G;
- `$END` is the only accepting state;
- The input alphabet is Σ;
- The stack alphabet Γ includes a new symbol $\{Z_0\}$ and, for all rules $X ::= \alpha_1...\alpha_n$ of G, n new symbols denoted $X ::= \alpha_1$, $X ::= \alpha_1\alpha_2$ and $X ::= \alpha_1...\alpha_n$. The stack symbols represent specific locations in the rules of G and serve as "return addresses" to follow once a symbol has been generated;
- The transitions are derived from the rules of G. For each rule $X ::= \alpha_1...\alpha_n$ of G, we add the following transitions:
  - $\bullet X, \tau(\alpha_1), \epsilon, X ::= \alpha_1, \texttt{start}(\alpha_1)$
  - $\texttt{end}(\alpha_1), \tau(\alpha_2), X ::= \alpha_1, X ::= \alpha_1\alpha_2, \texttt{start}(\alpha_2)$
  - ...
  - $\texttt{end}(\alpha_{n-1}), \tau(\alpha_n), X ::= \alpha_1...\alpha_{n-1}, X ::= \alpha_1...\alpha_n, \texttt{start}(\alpha_n)$
  - $\texttt{end}(\alpha_n), \epsilon, X ::= \alpha_1...\alpha_n, \epsilon, X\bullet$

  ... where:

  - The transitions are represented as a 5-tuple with, in this order, the current state, the character to read, the character to pop from the stack, the character to push on the stack and the new state;
  - $\texttt{start}(\alpha) = \texttt{end}(\alpha) = \tau(\alpha) = \alpha$ if α is a terminal;
  - $\texttt{start}(\alpha) = \bullet\alpha$ and $\texttt{end}(\alpha) = \alpha\bullet$ and $\tau(\alpha) = \epsilon$ if α is a non-terminal;

  ... and we also include a final transition: $(S\bullet, \epsilon, Z_0, \epsilon, \texttt{\$END})$;

- The initial stack symbol is Z_0.

Figure 2 shows the DFT pushdown automaton for the grammar with $S ::= aSb|\epsilon$ as a single rule. As illustrated in Figure 3, the pushdown automaton accepts *aabb* and the corresponding path mimics a depth-first traversal from left to right of the syntax tree of *aabb*.

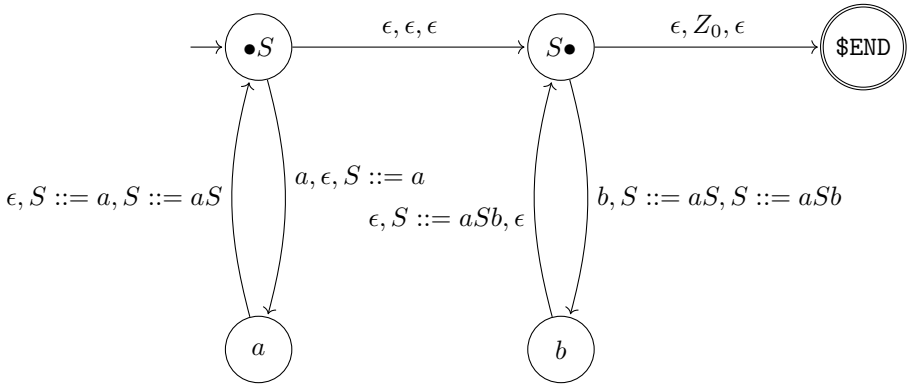


Figure 2: DFT pushdown automaton for $S ::= aSb|\epsilon$.

Instantaneous descriptions

	State	String	Stack
0	$\bullet S$	$aabb$	Z_0
1	a	abb	$(S ::= a)Z_0$
2	$\bullet S$	abb	$(S ::= aS)Z_0$
3	a	bb	$(S ::= a)(S ::= aS)Z_0$
4	$\bullet S$	bb	$(S ::= aS)(S ::= aS)Z_0$
5	$S\bullet$	bb	$(S ::= aS)(S ::= aS)Z_0$
6	b	b	$(S ::= aSb)(S ::= aS)Z_0$
7	$S\bullet$	b	$(S ::= aS)Z_0$
8	b	ϵ	$(S ::= aSb)Z_0$
9	$S\bullet$	ϵ	Z_0
10	$	ϵ	ϵ

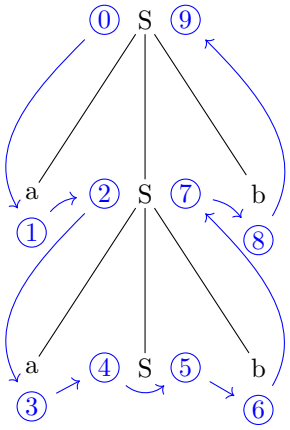


Figure 3: (Left) The pushdown automaton in Figure 2 accepts the word *aabb* through the path displayed in the table. (Right) This path corresponds to a left-to-right depth-first traversal of the syntax tree of *aabb*.

4.2 Two lemmas on the DFT pushdown automaton

We now prove two lemmas before showing how the DFT pushdown automaton can be leveraged to compute some values of `is_always_legal`.

Lemma 1. M, the set of strings accepted by the DFT pushdown automaton of G, is equal to L, the context-free language corresponding to G.

Proof. Since the only way to pop Z_0, the initial symbol of the stack, is to go from $S\bullet$ to $, a necessary and sufficient condition for $w \in M$ is that there is a path from $\bullet S$ to $S\bullet$ that reads w and does not alter the stack. We will see that this observation is useful to prove both $M \subset L$ and $L \subset M$.

To show that $M \subset L$, we will prove by recurrence for $n \geq 1$ that:

$P(n)$: for an arbitrary context-free grammar G, M_n, the set of strings accepted by the DFT pushdown automaton of G *with a maximum stack height of* n, is included in L.

$n = 1$: Let $w \in M_1$. A maximum stack height equal to 1 is only possible if $w = \epsilon$ and $S ::= \epsilon$ is a rule of G. Therefore, $w \in L$ and $P(1)$ is true.
$n > 1$: Let $w \in M_n$. Since w is accepted by the DFT pushdown automaton, there is a sequence of steps $((p_i, w_i, \beta_i))_{1 \leq i \leq N}$ such that:

- For all $1 \leq i \leq N$, $p_i \in Q, w_i \in \Sigma^*, \beta_i \in \Gamma^*$;
- For all $1 \leq i < N$, $(p_i, w_i, \beta_i) \vdash (p_{i+1}, w_{i+1}, \beta_{i+1})$ where $\vdash$ is the step relation of the DFT pushdown automaton;
- $(p_1, w_1, \beta_1) = (\bullet S, w, Z_0)$;
- $(p_N, w_N, \beta_N) = (\texttt{\$END}, \epsilon, \epsilon)$.

Given the transition relation of the DFT pushdown automaton and the fact that $(p_N, w_N, \beta_N) = (\texttt{\$END}, \epsilon, \epsilon)$, we necessarily have:

$$(p_{N-1}, w_{N-1}, \beta_{N-1}) = (S\bullet, \epsilon, Z_0)$$

The only possible way to end up in $S\bullet$ with $n > 1$ is, for a certain rule $S ::= \alpha_1 ... \alpha_K$ of G, to go successively through:

- $\texttt{start}(\alpha_1)$ with a $(S ::= \alpha_1)Z_0$ stack;
- $\texttt{end}(\alpha_1)$ with a $(S ::= \alpha_1)Z_0$ stack;
- ...
- $\texttt{start}(\alpha_K)$ with a $(S ::= \alpha_1 \dots \alpha_K)Z_0$ stack;
- $\texttt{end}(\alpha_K)$ with a $(S ::= \alpha_1 \dots \alpha_K)Z_0$ stack.

Consider $1 \leq i \leq K$. If α_i is a terminal, then $\texttt{start}(\alpha_i) = \texttt{end}(\alpha_i)$ and we necessarily have read α_i while arriving at $\texttt{start}(\alpha_i)$.
If α_i is a nonterminal, there is a path between $\bullet\alpha_i$ and $\alpha_i\bullet$ that does not affect the stack. From this path, we can directly build an accepted path for the DFT pushdown automaton of the grammar with the same rules as G but α_i as the start symbol. The maximum stack height of this new accepted path is strictly less than n and, as a consequence of the induction hypothesis, a word corresponding to the nonterminal α_i is read along the path between $\bullet\alpha_i$ and $\alpha_i\bullet$.
Otherwise said, whether α_i is a terminal or a nonterminal, going through the segment between $\texttt{start}(\alpha_i)$ and $\texttt{end}(\alpha_i)$ corresponds to reading a word matching α_i. If we apply this to all the symbols of the rule, we conclude that the whole sequence of steps reads $\alpha_1 ... \alpha_K$ which is a word of L. This proves $P(n)$.

Let us now show that $L \subset M$. For this, we will prove by recurrence for $n \geq 1$ that:

$P(n)$: for an arbitrary context-free grammar G, L_n, the set of strings generated by G *with a syntax tree height of* n, is included in M.

$n = 1$: Let $w \in L_1$. Since the height of the syntax tree of w is 1, w has been generated by a single rule, whose right-hand side is ϵ or a sequence of terminals. In both cases, we can check that w is accepted by the DFT pushdown automaton and, hence, that $P(1)$ is true.
$n > 1$: Let $w \in L_n$ and let $S ::= \alpha_1 ... \alpha_K$ be the first production rule applied to generate w. We write $w = w_1 ... w_K$ where w_i is generated by α_i for $1 \leq i \leq K$. From this, we can start building a path from $\bullet S$ to $\texttt{\$END}$:

- $(\bullet S, Z_0) \vdash (\texttt{start}(\alpha_1), (S ::= \alpha_1) Z_0)$
- $(\texttt{end}(\alpha_1), (S ::= \alpha_1) Z_0) \vdash (\texttt{start}(\alpha_2), (S ::= \alpha_1 \alpha_2) Z_0)$
- ...
- $(\texttt{end}(\alpha_1), (S ::= \alpha_1 ... \alpha_K) Z_0) \vdash (S\bullet, Z_0)$
- $(S\bullet, Z_0) \vdash (\texttt{\$END}, \epsilon)$

This path is incomplete because, if α_i is a nonterminal, $\texttt{start}(\alpha_i) = \bullet\alpha_i$ is not connected to $\texttt{end}(\alpha_i) = \alpha_i\bullet$. However, w_i necessarily has a syntax tree whose height is strictly less than s. Thanks to the induction hypothesis, we can then build a path from $\bullet\alpha_i$ to $\alpha_i\bullet$ that reads w_i and leaves the stack unaffected. This means that we can actually complete the partial path described in such a way that the resulting path reads w. As a consequence, $w \in M$ and $P(n)$ is true. □

We now prove an interesting property of the DFT pushdown automaton: if there is a path between the initial state with the initial stack and a certain state with a certain stack, there is an accepted path going through that specific state with this specific stack. In other words, *there are no dead ends* when traversing the DFT pushdown automaton.

Lemma 2. For all $w \in \Sigma^*$ such that there exist $q \in Q$ and $(Z_1, ..., Z_n) \in \Gamma^*$ with $(\bullet S, w, Z_0) \vdash^* (q, \epsilon, Z_n ... Z_0)$ and $q \neq \bullet S$, there exists $w_2 \in \Sigma^*$ such that $(\bullet S, w w_2, Z_0) \vdash^* (q, w_2, Z_n ... Z_0) \vdash^* (\texttt{\$END}, \epsilon, \epsilon)$.

Proof. We prove the lemma by recurrence over $n \geq 0$, the number of additional symbols in the stack in state q. Let $w \in \Sigma^*$, $q \in Q$ and $(Z_1, ..., Z_n) \in \Gamma^*$ with $(\bullet S, w, Z_0) \vdash^* (q, \epsilon, Z_n ... Z_0)$.
$n = 0$: The only possibility to move to a state other than the initial state with a stack equal to Z_0 is if $q = S\bullet$. In that case, $w_2 = \epsilon$ is adequate and $P(0)$ is true.
$n > 0$: Assume that $X ::= \alpha_1 ... \alpha_N$ is a rule of G and that Z_n can be written as $X ::= \alpha_1 ... \alpha_k$ with $k \leq N$. In this case, $q = \texttt{start}(\alpha_k)$ or $q = \texttt{end}(\alpha_k)$. Let $w' \in \Sigma^*$ such that:

$$w' = \begin{cases} s_k ... s_N & \text{if } \alpha_k \text{ is a nonterminal and } q = \texttt{start}(\alpha_k) \\ s_{k+1} ... s_N & \text{if } \alpha_k \text{ is a terminal or } q = \texttt{end}(\alpha_k) \end{cases}$$

... where $s_k, ..., s_N$ are strings matching the α_k, ... α_N symbols. With such a w': $(\bullet S, ww', Z_0) \vdash^* (q, w', Z_n...Z_0) \vdash^* (\texttt{end}(\alpha_N), \epsilon, Z_n...Z_0) \vdash (X\bullet, \epsilon, Z_{n-1}...Z_0)$. Using the induction hypothesis with $w = ww'$ is enough to conclude. □

4.3 `unlimited_credit_exploration`

With the two lemmas above, we can present `unlimited_credit_exploration`, an algorithm to compute some values of `is_always_legal`. For clarity, this exposition focuses on the case of `is_always_legal`(X, Y) with Y of length one, but the algorithm is extensible to sequences of arbitrary lengths. Using `unlimited_credit_exploration` to compute `is_always_legal`(X, Y) with $X, Y \in \Sigma$ involves the following steps:

1. Starting from the instantaneous description (X, Y, ϵ), we perform a breadth-first search through the DFT pushdown automaton to reach any $(Y, \epsilon, Z_n...Z_1)$ with $Z_n...Z_1$ being any stack. For this, we follow the step relation of the DFT pushdown automaton, except that, if the stack is empty and we need to pop a stack symbol, we still proceed with the transition and take note of the missing stack symbol. It is as if we had an unlimited line of credit to borrow stack symbols when the stack is empty. We also define a maximum stack size during the breadth-first search so that it is guaranteed to terminate.

2. After the breadth-first search, we collect all the trajectories that successfully led to Y. Each of these trajectories can be described as a sequence of transitions between nodes. The nodes correspond to a state and a stack and the transitions can be labeled with the *stack symbol debt*, i.e. α if we needed to pop α while the stack is empty or ϵ otherwise.

3. We represent all the trajectories as a non-deterministic finite automaton (NFA) with (X, ϵ) as the start node, any node with Y as the pushdown automaton state as a final node. This NFA is called below the *debt* NFA.

4. A node of the debt NFA is (recursively) defined as ϵ-coaccessible if:

   - It is a final node but not an initial node;
   - There is an ϵ from it to an ϵ-coaccessible node;
   - For each stack symbol that can be popped from the PDA state of the node, there is a transition with this stack symbol from this node to an ϵ-coaccessible node.

   `is_always_legal`(X, Y) = `True` if (X, ϵ) is ϵ-coaccessible.

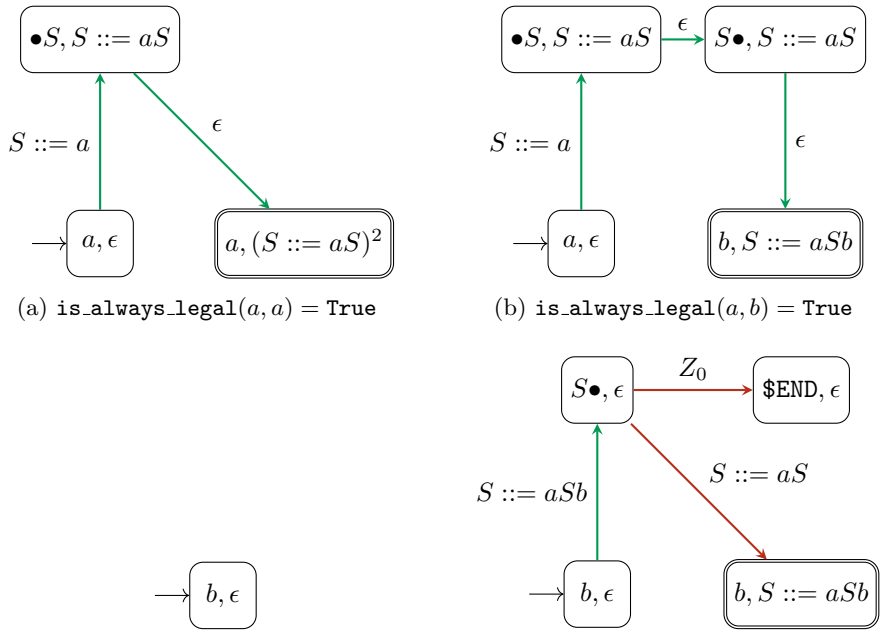
Figure 4 shows the debt NFA obtained with `unlimited credit exploration` applied to the $S = aSb|\epsilon$ grammar:

(a) $X = a, Y = a$: $S ::= a$ is the only stack symbol that can be popped from a in the DFT pushdown automaton displayed in Figure 2, hence (a, ϵ) is ϵ-coaccessible and `is_always_legal`(a, a) = `True`;

(b) $X = a, Y = b$: for the same reason as in the previous case, (a, ϵ) is ϵ-coaccessible and `is_always_legal`$(a, b) =$ `True`;

(c) $X = b, Y = a$: Starting from the instantaneous description (a, b, ϵ), it is not possible to reach (b, ϵ, s) (s being any stack). As a result, `is_always_legal`$(a, a) =$ `False` and `is_never_legal`$(a, a) =$ `True`;

(d) $X = b, Y = b$: (`$END`$, \epsilon)$ and therefore $(S\bullet, \epsilon)$ are not ϵ-coaccessible. We can then not conclude that `is_always_legal`$(b, b) =$ `True` (and in fact, we can see with a simple counter-example that `is_always_legal`$(b, b) =$ `False`).



(a) `is_always_legal`$(a, a) =$ `True`

(b) `is_always_legal`$(a, b) =$ `True`

(c) `is_always_legal`$(b, a) =$ `False`

(d) `is_always_legal`$(b, b) =$ `False`

Figure 4: Debt NFAs obtained by applying `unlimited_credit_exploration` to the $S = aSb|\epsilon$ grammar.

Below, we summarize and prove `unlimited_credit_exploration`.

Algorithm 2: `unlimited_credit_exploration` (outline; see above for a more comprehensive description)

Input: $X \in \Sigma, Y \in \Sigma, G$ context-free grammar
Output: `True` or `Unknown`

```
function unlimited_credit_exploration(X, Y, G)
    pda = build_dft_pushdown_automaton(G)
    T = explore_with_unlimited_credit(pda, X, Y)
    nfa = build_debt_nfa(T)
    if (X, ε) ∈ list_epsilon_coaccessible_nodes(nfa, pda) then
        return True
    else
        return Unknown
```

Proof. Let $X, Y \in \Sigma$ so that `unlimited_credit_exploration`$(X, Y, G) =$ `True` let `pda` and `nfa` be the DFT pushdown automaton and the debt NFA computed when evaluating `unlimited_credit_exploration`(X, Y, G).
We need to prove that `is_always_legal`$(X, Y) =$ `True`. For this, we assume that there exists $P \in \Sigma^*$ so that $PX \in L_p$ which means that there also exists $Q \in \Sigma^*$ so that $PXQ \in L$.
Lemma1 then implies that PXQ is accepted by `pda`. Therefore, there is a path from $(\bullet S, PX, Z_0)$ to $(X, \epsilon, Z_n...Z_0)$ with $Z_1, ..., Z_n \in \Gamma$.
Given that *there are no dead ends* in `pda` (i.e. Lemma 2), we know that it is possible to leave the current state. Therefore, there is either an outgoing transition that pops Z_n or an outgoing ϵ transitions. In both cases, since (X, ϵ) is ϵ-coaccessible in `nfa`, it is possible to choose the outgoing transition so that it brings us one step closer to Y. By repeating this, we can create a path between $(X, Y, Z_n...Z_0)$ and $(Y, \epsilon, z_k...z_1Z_0)$ with $z_1, ..., z_k \in \Gamma$.
Then there is a path from $(\bullet S, PXY, Z_0)$ to $(Y, \epsilon, z_k...z_1Z_0)$. As a consequence, according to Lemmas 1 and 2, $PXY \in L_p$ and `is_always_legal`$(X, Y) =$ `True`.
□

5 `are_jointly_legal` is not computable

We now turn our attention to `are_jointly_legal`, which, like `is_always_legal`, is not computable.

Proposition 2. `are_jointly_legal` is not computable for an arbitrary context-free grammar.

Proof. This is a direct consequence of the fact that `is_always_legal` is not computable and can be derived from `are_jointly_legal`. Indeed, for $X \in \Sigma$ and $S \in \Sigma^*$:

$$\texttt{is\_always\_legal}(X, S) = \texttt{are\_jointly\_legal}(X, S, \epsilon)$$

□

6 Obtaining some values of `are_jointly_legal`

6.1 Leveraging some known values `is_always_legal`

Although systematically computing `are_jointly_legal` is out of reach, we can obtain some of its values.
For this, we can take advantage of the known values of `is_always_legal`. If `is_always_legal`(Y_1, Y_2) = `True` for $Y_1 \in \Sigma, Y_2 \in \Sigma^*$, we can conclude that `are_jointly_legal`$(X_1, X_2...X_nY_1Y_2, X_2...X_nY_1)$ = `True` for all $X_1, ..., X_n \in \Sigma$.

6.2 Exploiting the interchangeability of some terminals

We can also exploit the fact that some terminals are interchangeable in the rules of the grammar. For $A, B \in \Sigma$, let $\phi_{A,B}$ be:

$$\begin{aligned} \phi_{A,B} \colon \Sigma &\to \Sigma \\ X &\mapsto \begin{cases} X & \text{if } X \neq A \\ B & \text{if } X = A \end{cases} \end{aligned}$$

We define two terminals $Y_1, Y_2 \in \Sigma$ as interchangeable if, for each rule $S ::= \alpha_1...\alpha_n$ of G:

- $S ::= \phi_{Y_1,Y_2}(\alpha_1)...\alpha_n$
- ...
- $S ::= \alpha_1...\phi_{Y_1,Y_2}(\alpha_n)$
- $S ::= \phi_{Y_2,Y_1}(\alpha_1)...\alpha_n$
- ...
- $S ::= \alpha_1...\phi_{Y_2,Y_1}(\alpha_n)$

...are also rules of G.
In this case, `are_jointly_legal`$(X, X_1...X_n, \phi_{Y_1,Y_2}(X_1)...\phi_{Y_1,Y_2}(X_n))$ = `True` for all $X_1, ..., X_n \in \Sigma$.

References

[1] Yehoshua Bar-Hillel, M. Perles, and E. Shamir. "On Formal Properties of Simple Phrase Structure Grammars". In: *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung* 14 (1961). Reprinted in Y. Bar-Hillel. (1964). *Language and Information: Selected Essays on their Theory and Application*, Addison-Wesley 1964, 116–150, pp. 143–172.

[2] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321455363.

[3] Vivien Tran-Thien. *Accelerating LLM Code Generation Through Mask Store Streamlining*. 2025. URL: `https://vivien000.github.io/blog/journal/grammar-llm-decoding.html`.