University of Copenhagen
Computer Science Department

Parallel Parsing using Futhark
Subtitle

Author: William Henrich Due
Advisor: Troels Henriksen
Submitted: June 12, 2023

Contents

1 Introduction

2 Theory

2.1 LL(k) Parser Generator

For the construction of a LLP(q, k) parser generator the construction of first and follow-set [3, p. 5] and a LL(k) parser generator is needed. In this section a short explanation of the construction of constructing these sets will be given. This is because during the research of this project $k = 1$ was quite often explained but never $k > 1$ since this was often seen as trivial.

The first and follow-set algorithms described takes heavy inspiration from Mogensens book "Introduction to Compiler Design" [2, pp. 55–65] and the parser notes [1, pp. 10–15] by Sestoft and Larsen. The modifications are mainly using the LL(k) extension described in the Wikipedia article in the section "Constructing an LL(k) parsing table"[1] [5].

Definition 2.1 (Truncated product)**.** Let $G = (N, T, P, S)$ be a context-free grammar, $A, B \in \mathbb{P}((N \cup T)^*)$[2] be sets of symbol strings and $\omega, \delta \in (T \cup N)^*$. The truncated product is defined in the following way.

$$A \odot_k B \stackrel{\text{def}}{=} \left\{ \underset{\gamma \in \{\omega \,:\, \omega\delta = \alpha\beta, |\omega| \leq k\}}{\arg\max} |\gamma| : \alpha \in A, \beta \in B \right\}$$

The truncated product can be computed by hand by concatenating each element $\alpha \in A$ in front of every element $\beta \in B$. This results in a new set, then the k first symbols of each element in this set is kept while the rest is discarded.

Definition 2.2 (Nonempty substring pairs)**.** Let $G = (N, T, P, S)$ be a context-free grammar, $\omega \in (N \cup T)^*$ be a symbol string and $\alpha, \beta \in (N \cup T)^+$ be nonempty symbol strings. The set of every nonempty way to split ω into two substrings is defined to be.

$$\varphi(\omega) \stackrel{\text{def}}{=} \{(\alpha, \beta) : \alpha\beta = \omega\}$$

Algorithm 2.1 (Solving FIRST_k set)**.** Let $G = (N, T, P, S)$ be a context-free grammar, the first-sets can be solved as followed.

$$\begin{aligned}
\text{FIRST}_k(\epsilon) &= \{\epsilon\} \\
\text{FIRST}_k(t) &= \{t\} \\
\text{FIRST}_k(A) &= \bigcup_{\delta \,:\, A \to \delta \in P} \text{FIRST}_k(\delta) \\
\text{FIRST}_k(\omega) &= \bigcup_{(\alpha,\beta) \,\in\, \varphi(\omega)} \text{FIRST}_k(\alpha) \odot_k \text{FIRST}_k(\beta)
\end{aligned}$$

This may result in an infinite loop if implemented as is so fixed point iteration is used to solve this system of set equations. Let $\mathcal{M} : N \to \mathbb{P}(T^*)$ be a surjective function which maps nonterminals to sets of terminal strings, this

[1] At the time of writing the Wikipedia article does have a description of constructing first and follow-sets for $k > 1$. The problem is the algorithm described does not fullfill the definition of first and follow-sets that is being used in the LLP paper [3, p. 5].

[2] $\mathbb{P}$ is the powerset.

function can be thought of as a dictionary. FIRST'_k is then the following modified version of FIRST_k.

$$\begin{aligned}
\text{FIRST}'_k(\epsilon, \mathcal{M}) &= \{\epsilon\} \\
\text{FIRST}'_k(t, \mathcal{M}) &= \{t\} \\
\text{FIRST}'_k(A, \mathcal{M}) &= \mathcal{M}(A) \\
\text{FIRST}'_k(\omega, \mathcal{M}) &= \bigcup_{(\alpha,\beta)\,\in\,\varphi(\omega)} \text{FIRST}'_k(\alpha, \mathcal{M}) \odot_k \text{FIRST}'_k(\beta, \mathcal{M})
\end{aligned}$$

This function is then used to solve for a FIRST_k function for some fixed k with fixed point iteration the following way.

1. Initialize a dictionary $\mathcal{M}_0$ such that $\mathcal{M}_0(A) = \emptyset$ for all $A \in N$.
2. A new dictionary $\mathcal{M}_{i+1} : N \to \mathbb{P}(T^*)$ is constructed by $\mathcal{M}_{i+1}(A) = \bigcup_{\delta\,:\,A\to\delta\in P} \text{FIRST}'_k(\delta, \mathcal{M}_i)$ for all $A \in N$ where $\mathcal{M}_i$ is the last dictionary that was constructed.
3. If $\mathcal{M}_{i+1} = \mathcal{M}_i$ then terminate the algorithm terminates else recompute step 2.

Let $\mathcal{M}_f$ be the final dictionary after the algorithm terminates then it holds that $\text{FIRST}_k(\omega) = \text{FIRST}'_k(\omega, \mathcal{M}_f)$ if k stays fixed.

Algorithm 2.2 (Solving FOLLOW_k set)**.** Let $G = (N, T, P, S)$ be a context-free grammar, the follow-sets can be solved as followed.

$$\text{FOLLOW}_k(A) = \bigcup_{B\,:\,B\to\alpha A\beta\in P} \text{FIRST}_k(\beta) \odot_k \text{FOLLOW}_k(B)$$

Once again this may not terminate so fixed point iteration can be used to solve the equation with the following altered FOLLOW_k where $\mathcal{M} : N \to \mathbb{P}(T^*)$ is a surjective function.

$$\text{FOLLOW}'_k(A, \mathcal{M}) = \bigcup_{B\,:\,B\to\alpha A\beta\in P} \text{FIRST}_k(\beta) \odot_k \mathcal{M}(B)$$

This FOLLOW'_k function for sine fixed k can then be computed using the following algorithm.

1. Extend the grammar $G = (N, T, P, S)$ using $G' = (N', T', P', S') = (N \cup \{S'\}, T \cup \{\square\}, P \cup \{S' \to S\square^k\}, S')$.
2. Initialize a dictionary $\mathcal{M}_0$ such that $\mathcal{M}_0(A) = \emptyset$ for all $A \in N\backslash\{S\}$ and $\mathcal{M}_0(S) = \{\square^k\}$.

3. A new dictionary $\mathcal{M}_{i+1} : N \to \mathbb{P}(T^*)$ is constructed by $\mathcal{M}_{i+1}(A) = \bigcup_{B\,:\,B\to\alpha A\beta\in P} \text{FIRST}_k(\beta) \odot_k \mathcal{M}_i(B)$ for all $A \in N$ where $\mathcal{M}_i$ is the last dictionary that was constructed.

4. If $\mathcal{M}_{i+1} \neq \mathcal{M}_i$ then recompute step 3.

5. Let $\mathcal{M}_f$ be the final dictionary after step 4. is completed. Let $\mathcal{M}_u$ be another dictionary where $\mathcal{M}_u(A) = \{\alpha : \alpha\square^* \in \mathcal{M}_f(A)\}$ for all $A \in N\backslash\{S'\}$

It then holds that $\text{FOLLOW}_k(A) = \mathcal{M}_u(A)$ if k stays fixed for grammar G.

2.2 LLP(q, k) Parser Generator

2.2.1 The idea

The idea of the LLP(q, k) grammar class comes from wanting to create a LL(k) like grammar class which can be parsed in parallel. To describe how this is done a definition for a given state during LL(k) parsing is needed.

Definition 2.3 (Production sequence)**.** Let $G = (N, T, P, S)$ be a context-free grammar where each production $p_i \in P$ is assigned a unique integer $i \in \{0, ..., |P| - 1\} = \mathcal{I}$. Then the set of every valid and invalid sequence of productions $\mathcal{S}$[3] is given by.

$$\mathcal{S} = \{(a_k)_{k=0}^n : n \in \mathbb{N}, a_k \in \mathcal{I}\}$$

Definition 2.4 (LL parser configuration)**.** Let $G = (N, T, P, S)$ be a context-free grammar that is an LL(k) grammar for some $k \in \mathbb{Z}_+$. A given configuration [3, p. 5] of a LL(k) parser is then given by.

$$(w, \alpha, \pi) \in T^* \times (T \cup N)^* \times \mathcal{S}$$

For a LL(k) parser configuration (w, α, π) would w denote the input string, α denote the push down store and π denote the sequence of rules used to derive the consumed input string.

When using deterministic LL(k) parsing you want to create a parsing function $\phi : T^* \to \mathcal{S}$ for a grammar $G = (N, T, P, S)$. This parser function is a function which is able to create the production sequence as defined by the relation $\vdash^*$ [3, p. 6].

$$\phi(w) = \pi \text{ where } (w, S, (\ )) \vdash^* (\epsilon, \epsilon, \pi)$$

[3]It is chosen to use a squence for the "prefix of a left parse" [3, p. 5] because it seemed like a better choice then the grammar notation.

If the $\vdash^*$ relation does not hold then w can not be parsed.

The concept of deterministic LLP(q,k) parsing is if a string $w \in T^*$ is going to be parsed then construct every pair such that.

$$\begin{aligned} M = &\{((x,y),i) : w = \delta x y_i \beta, |x| = q, |y_i| = k\} \\ &\cup \{((x,y),i) : w = x y_i \beta, |x| \leq q, |y| = k\} \\ &\cup \{((x,y),i) : w = \delta x y_i, |x| = q, |y| \leq k\} \end{aligned}$$

Where $i \in \mathbb{N}$ denotes the index of where the start of the substring y_i such the ordering can be kept. Then we would want to create table lookup function $\Phi : T^* \times T^* \to (T \cup N)^* \times (T \cup N)^* \times \mathcal{S}$. This function maps the pairs (x,y) to a triplet (ω, α, π) which is much the same as the configuration described in definition 2.4.

Definition 2.5 (LLP parser configuration)**.** Let $G = (N, T, P, S)$ be a context-free grammar that is a LLP(q,k) grammar for some $q, k \in \mathbb{Z}_+$. Let (x,y) be a pair such that xy occurs as a substring in $\mathcal{L}(G)$, $x \in T^{*q}$, $y \in T^{*k}$. If PSLS$(x,y) = \{\omega\}$ the pair (x,y) has the following LLP configuration.

$$(\omega, \alpha, \pi) \in (T \cup N)^* \times (T \cup N)^* \times \mathcal{S}$$

Where ω is the initial push down store, α is the final push down store after parsing $(y, \omega, (\ )) = (vw, \omega, (\ )) \vdash^* (w, \alpha, \pi)$, $v \in T$, $w \in T^*$ and π are the resulting productions.

After all the x, y pairs have been constructed they are glued together to determined if the resulting triplet is (S, ϵ, π) meaning the input was parsable. This is described in detail in the LLP paper [3, p. 7], but this description will be helpful for the rest of the paper.

2.2.2 Determining if a grammar is LLP

When dealing with a LL(k) parser a common answer to if the grammar is a LL(k) grammar is: if the $LL(k)$ parser table can be constructed then it is a LL(k) grammar. The same goes for LLP(q,k) grammars, that is a grammar is a LLP(q,k) if the LLP(q,k) table can be constructed.

The first step in determining if a grammar is a LLP(q,k) grammar is if it is in the LL grammar class. This is because the LLP parser uses the LL parser to construct the table, therefore the class suffers from the same limitations. The next step is to determine if the (x,y) pair leads to multiple (ω, α, π) LLP configurations. This is what definition 10 [3, p. 13] is used for, to determine if the grammar is LLP.

Definition 10 [3, p. 13] uses the PSLS(x, y) [3, p. 12] values to determine the initial push down stores which can be used to determine the LLP configuration. The trouble is when working with LLP grammars the PSLS(x, y) definition can be troublesome when trying to understand if a grammar is LLP(q, k). Therefore, some examples of using the definition are given below.

Example 2.1. Let $(\{A, B\}, \{a, b\}, P, A)$ be a context free grammar where P is.

$$A \to abbB \qquad B \to b \qquad B \to A$$

It will be checked if the grammar is LLP$(1, 1)$, when computing PSLS(b, b) it can be seen there exists the following occurrences of bb which leads to two different initial push down stores.

$$A \Rightarrow_{lm}^{*} (abb)^{*}A \Rightarrow (abb)^{*}abb_{x}B_{B\gamma} \Rightarrow^{*} (abb)^{*}abb_{x}b_{y}$$

This derivation[4] corresponds to the first set in the PSLS(b, b) definition. The shortest prefix of B is B where $\text{FIRST}_1(b) \subseteq \text{FIRST}_1(B)$ so $B \in \text{PSLS}(b, b)$ [4, p. 2] by definition. The other occurrence comes from the last set in the PSLS definition.

$$A \Rightarrow_{lm}^{*} (abb)^{*}A \Rightarrow (abb)^{*}ab_{x}b_{y}B \Rightarrow^{*} (abb)^{*}ab_{x}b_{y}B$$

Since $\text{FIRST}_1(b) = \{b\}$ then $b \in \text{PSLS}(b, b)$ by definition. The initial push down store for the admissible pair (b, b) is PSLS$(b, b) = \{b, B\}$. Therefore, by Definition 10 [3, p. 13] this grammar is not LLP$(1, 1)$. If one were to check for all admissible pairs then they would find that PSLS(b, b) is the only problem. If one wishes to parse the grammar a $LLP(2, 1)$ parser can be used. It solves the ambiguities since PSLS$(ab, b) = \{b\}$ and PSLS$(bb, b) = \{B\}$.

Example 2.2. Let $(\{S\}, \{[,]\}, P, S)$ be a context free grammar where P is.

$$S \to [S] \qquad S \to \epsilon$$

This grammar seems like it is not LLP(q, k) for any $q, k \geq 1$ because when LL parsing the pairs $([^{q},]^{k})$ can lead multiple LL configuration $(]^{n}, S]^{n}, \pi)$ where $q + k \leq n$. This grammar is actually a LLP$(1, 1)$ grammar because the LLP parser only uses the shortest prefix of the initial push down store in the LLP configuration. This can be determined as not a problem by using the PSLS definition.

$$S \Rightarrow_{lm}^{*} [^{n}S]^{n} \Rightarrow [^{n+1}S]^{n+1} \Rightarrow^{*} [^{n+1}]^{n+1}$$

[4]The subscripts denote what the symbols correspond to in the PSLS definition and does change what the symbols mean in the grammar.

Here the admissible pair (x, y) are $([^q,]^k)$ where $q, k \leq n+1$ for a LLP(q, k) grammar. Here $B\gamma$ from the definition corresponds to $S]^{n+1}$ and the shortest prefix of $S]^{n+1}$ is $S]$ since $\text{FIRST}_1(S) = \{\epsilon\}$, but $\text{FIRST}_1(]^k) \subseteq \text{FIRST}_1(S])$ holds. Therefore, this is not a reason for the grammar not being LLP(q, k).

Example 2.3. Let $(\{S\}, \{a\}, P, S)$ be a context free grammar where P is.

$$S \to aaS \qquad S \to \epsilon$$

This grammar is mentioned in the LLP paper [3, p. 16] as a grammar that is not LLP(q, k) for any $q, k \in \mathbb{Z}_+$. This is because the pairs (a^q, a^k) could lead to the possible initial push down stores are $\text{PSLS}(a^q, a^k) = \{a, S\}$. This is much the same reason as to why Example 2.1 is not LLP$(1, 1)$. The trouble with this grammar is even when increasing q or k there is not a symbol that can make $\text{PSLS}(a^q, a^k)$ become a singleton. If the productions for the grammar were.

$$S \to aS \qquad S \to \epsilon$$

Then the grammar is LLP$(1, 1)$ because now only $\text{PSLS}(a^q, a^k) = \{S\}$ can occur.

3 Implementation

3.1 Structure

3.2 Assumptions

At times the notation of the LLP paper [3] was unknown to the author of this paper. This was cause for problems when trying to implement the algorithms, therefore the following assumptions about the LLP paper [3] are mentioned here.

Algorithm 8 [3, p. 13] has the following notation.

step 2. (a): "$\{[S' \to \vdash S \dashv \bullet, u, \epsilon, \epsilon]\}, u = \text{LAST}_q(\vdash S \dashv)$"
step 3. (b): "$\{[Y \to \delta\bullet, u', v, \gamma]\}, u' = \text{LAST}_q(\text{BEFORE}_q(Y)\delta)$"

In the context u and u' are used, they are supposed to be terminal strings. Both of these sets do not result in singletons, so the interpretation cannot be unwrapping them. An example of this is if you compute u with $q = 2$

for the Example 11 grammar [3, p. 14]. It is therefore assumed that for each element in the u and u' sets an item is constructed from them i.e.

$$\begin{aligned}&\text{step 2. (a):} \quad \{[S' \to \vdash\ S\ \dashv \bullet, u, \epsilon, \epsilon], u \in \text{LAST}_q(\vdash\ S\ \dashv)\}\\ &\text{step 3. (b):} \quad \{[Y \to \delta \bullet, u', v, \gamma], u' \in \text{LAST}_q(\text{BEFORE}_q(Y)\delta)\}\end{aligned}$$

The second type of notation in Algorithm 8 is.

$$\begin{aligned}&\text{step 3. (a):} \quad \text{“}u_j \in \text{LAST}_q(\text{BEFORE}_q(Y)\alpha)\text{”}\\ &\text{step 3. (b):} \quad \text{“}u' = \text{LAST}_q(\text{BEFORE}_q(Y)\delta)\text{”}\end{aligned}$$

For step 3. (a) $\text{BEFORE}_q(Y)\alpha$ is interpreted as element-wise concatenation of α on the back of each string in $\text{BEFORE}_q(Y)$. Since this results in a set and $\text{LAST}_q : (N \cup T)^* \to \mathbb{P}(T^*)$ then it is assumed that LAST_q is used element-wise on the set $\text{BEFORE}_q(Y)\alpha$. This would intern mean u_j is a set, therefore it is also assumed that union is implicitly used. The same idea goes for step 3. (b) since from before it was assumed that it should be interpreted as $\{[Y \to \delta \bullet, u', v, \gamma], u' \in \text{LAST}_q(\text{BEFORE}_q(Y)\delta)\}$. Therefore, these two steps should be interpreted as.

$$\begin{aligned}&\text{step 3. (a):} \quad u_j \in \bigcup\nolimits_{\omega \in \text{BEFORE}_q(Y)\alpha} \text{LAST}_q(\omega)\\ &\text{step 3. (b):} \quad u' \in \bigcup\nolimits_{\omega \in \text{BEFORE}_q(Y)\delta} \text{LAST}_q(\omega)\end{aligned}$$

3.3 Memoization of FIRST and LAST

The FIRST implementation corresponds to algorithm 2.1 and the LAST implementation corresponds to the one mentioned in the LLP paper [3, p. 12] which uses an existing FIRST implementation. A big problem with FIRST is it is extremely expensive because of the truncated product step.

$$\text{FIRST}'_k(\omega, \mathcal{M}) = \bigcup_{(\alpha,\beta)\,\in\,\varphi(\omega)} \text{FIRST}'_k(\alpha, \mathcal{M}) \odot_k \text{FIRST}'_k(\beta, \mathcal{M})$$

This inefficiency can be solved by the use of memoization since FIRST is recursively defined. In Haskell this can be done using a `State` monad from the `mtl` library. A problem with this is it will end up taking a lot of memory because every possible input string might end up being stored with its corresponding FIRST set. Since the FIRST and LAST implementation is only used on derivable strings then the memory must be bounded by the grammar.

The worst case grammar one could use is a grammar which generates every combination of terminals.

$$S \to \epsilon | a_1 S | a_2 S | a_3 S | \cdots | a_n S$$

This could occur but does not seem useful so such a case seem unlikely to happen.

3.4 LLP collection of item sets

When constructing the LLP collection of items sets Algorithm 8 [3, p. 13] is used. The actual Haskell implementation of algorithm 8 should match the implementation, but some implementation details would be needed to be explained. The first implementation detail to consider is.

step 3. (a): "γ is the shortest prefix of $X\delta$ such that $\gamma \Rightarrow^* a\omega$, a is the first symbol of v_j" [4]

The way this can be solved is by doing a breadth first search on all the possible prefixes to see if the a can be derived. Then just choose the shortest of the prefixes that can derive $a\omega$. This is the first implementation that was used when algorithm 8 was implemented.

The current implementation creates all prefixes of γ and computes FIRST_1 of these prefixes. γ is then the shortest prefix where $a \in \text{FIRST}_1(\gamma)$. This should be a faster implementation because the FIRST implementation uses memoization. This results in FIRST can become a dictionary look up instead of a breadth first search.

3.5 Parser

The parser generator creates a Futhark source file which can be used for parsing. The reason for doing so is Futhark is designed for "parallel efficient computing"[5] that can be executed on a GPU. Therefore the general purpose language, Haskell is used to generate the source files which contains the table parser.

The code the table generator written in Haskell does is mainly creating the table. This table is the function `key_to_config` that patterns matches on a pair (x, y) which maps to their respective configuration. These configurations do not actually correspond to an LLP configuration. Instead

[5] link

of using the LLP configuration (α, ω, π) as is, the list homomorphism in algorithm 18 [3, p. 18] which results in the tuples $(RBR(\alpha)LBR(\omega^R), \pi)$ besides for the starting pairs $(\epsilon, \vdash w)$ where $w \in T^*$ which becomes $(LBR(\omega^R), \pi)$.

Besides this the Futhark implementation matches algorithm 18. A missing piece is the parallel bracket matching which is not described in the paper. The implementation used takes a lot of inspiration from the implementation described on the Futhark website. The differences are the balancing check is made before the grading and the tabulation is never done.

Something to note is the glue [3, p. 7] reduce could have been used instead of algorithm 18 [3, p. 18]. This is a bad choice for two reasons, the first is it is slow [3, p. 17]. The second reason is Futhark does not have dynamic arrays and does not allow for using concatenation when using the built-in `reduce` function.

It is also important to note that creating an array of strings in Futhark is also problematic task. Therefore, the terminals, nonterminals and productions are assigned an index which is used instead. Besides this a specific integer is used to assign an empty terminal or nonterminal since a function in Futhark can not return arrays of different lengths. These empty symbols are filtered away later on.

3.6 Admissible Pairs

A method for computing admissible pairs is needed for algorithm 15 [3, p. 15]. These admissible pairs are such that $xy \in \mathcal{L}(G)$ where G is the grammar. The obvious way to do this is simply creating all leftmost derived strings from the start i.e.

$$\{w : S \Rightarrow_{lm}^{*} w, w \in T^*, |w| \leq 2(q + k + 1)\}$$

Then if $w = axyb$ where $a, b \in T^*$ then xy occurs as a string within the grammar and therefore (x, y) is an admissible pair. This matches the current implementation and is the fasts' implementation the author was able to create.

An alternative idea was for every production $P \to a_1 a_2 a_3 \dots a_n$ create productions for all tails of the right-hand side i.e. the productions created from P are.

$$A_1 \to a_1 a_2 a_3 \dots a_n \qquad A_2 \to a_2 a_3 \dots a_n \qquad A_3 \to a_3 \dots a_n \qquad A_n \to a_n$$

Then for all new productions created from each existing production create a new starting nonterminal S' which has all productions that leads to any

nonterminal i.e.

$$S' \to \mathcal{N} \text{ where } \mathcal{N} \in N$$

Then using $\text{FIRST}_{q+k+1}(S')$ and $\text{FOLLOW}_{q+k+1}(S')$ to construct a LL parser table. Then by creating a set of the parser tables terminal keys one is able to check if xy is infix of any elements in the set.

The problem is this is slower than constructing the leftmost derivations described. This is probably because the grammar ends up getting really large so the fix point iteration of FIRST and FOLLOW becomes expensive. Also, the leftmost derivations are only constructed once so the first method are likely to be a good fit.

4 Testing

4.1 First and Follow sets

To test that the first and follow-sets are computed correctly using Algorithm 2.1 and 2.2, two kinds of testing methods are employed. The first testing method is simply by using some small cases of examples of existing precomputed first and follow sets. Existing results were taken from [2, pp. 58, 62, 63, 65] with some small modifications to the results where ϵ is included in the first and follow sets due to the LLP paper definition [3, p. 5]. The last kind of test is done using property based testing but only for the two grammars [2, pp. 62, 63]. The property tested for is if the functions can reconstruct the LLP paper definition of first and follow [3, p. 5].

$$\begin{aligned}\text{FIRST}_k(\alpha) &= \{x : x \in T^* : \alpha \Rightarrow^* x\beta \wedge |x| = k\} \\ &\cup \{x : x \in T^* : \alpha \Rightarrow^* x \wedge |x| \leq k\} \\ \text{FOLLOW}_k(A) &= \{x : x \in T^* : S \Rightarrow^* \alpha A \beta \wedge x \in \text{FIRST}_k(\beta)\}\end{aligned}$$

The FIRST_k definition can be naively implemented by doing a breadth first search on derivable strings for each nonterminal of a grammar. If all nonterminals first sets are reproducible by algorithm 2.1 then the FIRST_k function is computed correctly for the given grammar.

The FOLLOW_k definition can also be naively implemented by doing a breadth first search on derivable strings from the starting nonterminal. Every time a nonterminal A occurs in the derivable string the FIRST_k function is computed of the trailing symbols β. These sets are then used to construct the $\text{FOLLOW}_k(A)$ set. If all $\text{FOLLOW}_k(A)$ are reproducible by algorithm 2.2 then the FOLLOW_k function is computed correctly for the given grammar.

For the property based tests $k \in \{1, 2, 3, 4\}$ is used for the two grammars because the naive first and follow implementations becomes extremely slow for larger k.

All of these tests was passed. It is therefore assumed that the naive implementations are correct due to their simplicity, therefore Algorithm 2.1 and 2.2 works as intended.

4.2 LLP(q, k)

The algorithms related to parsing and construction of LLP(q, k) are tested. Some precomputed examples from the LLP paper [3] are used to test that each algorithm arrives at the correct result. This is because there is not really any easy way of implementing the PSLS definition. The second problem is computing the LLP Item collection by hand would be quite a daunting task. Therefore, the LLP collection in example 11. [3, p. 14] and the PSLS table in example 12. [3, p. 14] is used to assert correctness. These two tests helps in directing the correctness of the LLP collection and PSLS table generation but does not help with asserting the correctness of the parser.

To assert the validity of the parser, the grammar in example 11. [3, p. 14] is once again used. It has epsilon productions and nonterminals in a row, so it still seems like a good choice for testing the parser on a smaller scale. To do this the LLP(q, k) parser with $q, k \in 1, 2, 3$ are constructed resulting in nine parsers. These parsers are tested by constructing all leftmost derivable strings of a given length and testing if the parsers can parse all these strings. The parsers are all tested on all strings of a given length which are not leftmost derivable strings and should fail on all these strings.

All of these tests are parsed, and therefore it is assumed that the parser works as intended.

5 Conclusion

References

[1] Sestoft Peter and Larsen Ken Friis. *Grammars and parsing with Haskell Using Parser Combinators*. Version 3. At the time of writing these notes are used in the Advanced Programming course at the University of Copenhagen. Sept. 2015.

[2] Mogensen Torben Ægidius. *Introduction to Compiler Design*. 2nd ed. London: Springer Cham. DOI: https://doi.org/10.1007/978-3-319-66966-3.

[3] Ladislav Vagner and Bořivoj Melichar. “Parallel LL parsing”. In: *Acta Informatica* 44.1 (Apr. 2007), pp. 1–21. ISSN: 1432-0525. DOI: 10.1007/s00236-006-0031-y. URL: https://doi.org/10.1007/s00236-006-0031-y.

[4] Ladislav Vagner and Bořivoj Melichar. “Parallel LL parsing”. In: *Acta Informatica* 44.1 (Apr. 2007), pp. 73–73. ISSN: 1432-0525. DOI: 10.1007/s00236-006-0032-x. URL: https://doi.org/10.1007/s00236-006-0032-x.

[5] Wikipedia. *LL parser — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=LL%20parser&oldid=1145098081. [Online; accessed 03-May-2023]. 2023.