University of Copenhagen
Computer Science Department

Parallel Parsing using Futhark
The Implementation of a Parallel LL Parser Generator

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

Abstract

An implementation of a deterministic parallel LL parser generator is presented. This LL parser generator only works for a subset of LL grammars which is known as the grammar class LLP. The implementation details of such a parser generator will be discussed. These details entail the creation of LL(k) parsers and parallel computing using Futhark. Furthermore, problems with definitions and algorithms that stems from the LLP grammar class will be examined.

Contents

1 Introduction

Parsing is usually a sequential task which can be done using LL, LR or LALR parsers. In the case of LL parsing you would need to keep track of a pushdown store through the parsing. Using this pushdown store you can keep track of the context for the given input string from start till the end.

Instead of doing this an idea for a parallel parser is to put some restrictions on the LL grammar class. These restrictions should allow for the parser to look at an arbitrary substring from a grammar and determine if it is valid or not. It should then also be able to compare this specific substring to its neighboring substrings and determine if the given context is valid.

One such grammar class which uses this strategy is the LLP grammar class. The LLP(q, k) grammar class uses q lookback and k lookahead for determining the substrings sizes.

In this paper a LLP(q, k) parser generator will be implemented. It is likely this is the first implementation of such a parser. This is due to the LLP papers [5] few citations and the fact that some mistakes have been found in the LLP papers algorithms and definitions through this project. It is known there exists a LLP$(1, 1)$ [7], but one of the problems found are unlikely to appear and the other appears when $q > 1$ and $k > 1$.

The technologies used to do the implementation is Haskell 2010 using GHC 9.2.7, Cabal 3.10.1.0, Python 3.10 and Futhark nightly. The code base can be found at the link below.

https://github.com/diku-dk/parallel-parser/releases/tag/BachelorProject

Acknowledgements

Thanks, Troels Henriksen for the immense amount of help throughout this project, and thanks to my sister for helping me proofread.

2 Theory

2.1 LL(k) Parser Generator

To construct a LLP(q, k) parser generator the construction of FIRST sets, FOLLOW sets [5, p. 5] and a LL(k) parser generator is needed.

2.1.1 FIRST$_k$ and FOLLOW$_k$

During the research of this project it was quite often explained how to construct FIRST$_k$ and FOLLOW$_k$ where $k = 1$ 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" [4, pp. 55–65] and the parser notes [3, 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] [8].

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

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

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 $\alpha, \beta \in (N \cup T)^+$[3]

[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 [5, p. 5].

[2] Let Z be an alphabet, Z^* is defined to be $\varepsilon \in Z^*$ and $Z^* \overset{\text{def}}{=} \{tv : t \in Z, v \in Z^*\}$

[3] $Z^+ \overset{\text{def}}{=} Z^* \backslash \{\varepsilon\}$

be nonempty symbol strings. The set of every nonempty way to split ω into two substrings is defined to be.

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

Using these definitions the FIRST_k and FOLLOW_k algorithms can now be described.

Algorithm 2.1 (Solving FIRST_k sets)**.** Let $G = (N, T, P, S)$ be a context-free grammar and $\text{FIRST}_k : (N \cup T)^* \to \mathbb{P}(T^*)$[4]. The first set for a given string can be solved as followed.

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

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

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

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

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.

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

3. If $\mathcal{M}_{i+1} = \mathcal{M}_i$ then terminate the algorithm else recompute step 2.

4. Let $\mathcal{M}_f$ be the dictionary last dictionary constructed during the fixed point iteration. It then holds that $\text{FIRST}_k(\omega) = \text{FIRST}'_k(\omega, \mathcal{M}_f)$ if k stays fixed.

Now that the FIRST_k sets can be computed, the FOLLOW_k set can also be computed.

Algorithm 2.2 (Solving FOLLOW_k sets)**.** Let $G = (N, T, P, S)$ be a context-free grammar and $\text{FOLLOW}_k : N \to \mathbb{P}(T^*)$. The follow set for a given nonterminal can be solved as followed.

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

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.

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

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

1. Extend the grammar $G = (N, T, P, S)$ using $G' = (N', T', P', S') = (N \cup \{S'\}, T \cup \{\Box\}, P \cup \{S' \to S\Box^k\}, S')$.

2. Initialize a dictionary $\mathcal{M}_0$ such that $\mathcal{M}_0(A) = \emptyset$ for all $A \in N' \backslash \{S\}$ and $\mathcal{M}_0(S) = \{\Box^k\}$.

3. A new dictionary $\mathcal{M}_{i+1} : N' \to \mathbb{P}(T^*)$ is constructed by $\mathcal{M}_{i+1}(A) = \bigcup_{B\,:\,B\to\alpha A\beta\in P} \text{FIRST}_k(\beta) \overset{k}{\bullet} \mathcal{M}_i(B)$ 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 dictionary last dictionary constructed during the fixed point iteration. Let $\mathcal{M}_u$ be a dictionary where $\mathcal{M}_u(A) = \{\alpha : \alpha\Box^* \in \mathcal{M}_f(A)\}$ for all $A \in N$.

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

Using FIRST$_k$ and FOLLOW$_k$ a LL(k) table which maps nonterminals and k or less terminals to the index of a production can be constructed.

Definition 2.3 (LL(k) table)**.** Let $G = (N, T, P, S)$ be a LL(k) context-free grammar and $\tau : N \times T^* \to \mathbb{P}(\mathbb{N})$ denote the LL(k) table. For a given production $A \to \delta = p_i \in P$ where $i \in \{0, ..., |P| - 1\}$ is a unique index.

$$i \in \tau(A, s) \text{ where } s \in \text{FIRST}_k(\delta) \overset{k}{\bullet} \text{FOLLOW}_k(A)$$

If for a given grammar it holds that $|\tau(A, s)| = 1$ for all $(A, s) \in N \times T^*$ then the grammar is LL(k).

2.1.2 LL Parsing

To describe how LL parsing is done, a definition for a given state during LL(k) parsing is needed. First a definition for production sequences is needed.

Definition 2.4 (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 indexes $\mathcal{S}$[5] is given by.

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

This definition can now be used to define a given state of a LL(k) parser.

Definition 2.5 (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 [5, p. 5] of a LL(k) parser is then given by.

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

Where w is the suffix of an input string, α is the current states pushdown store and π is the prefix of a production sequence used to derive the input string.

When using deterministic LL(k) parsing you want to create a parsing function. To define how such a function is created the LL parsing relation is defined. This relation holds if the left LL configuration can turn into the

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

right LL configuration after the left configuration pops the top element of the pushdown store.

$$(w, \alpha, \pi) \vdash (\bar{w}, \bar{\alpha}, \bar{\pi})$$
$$\iff$$
$$\underbrace{(a \in T \land w = a\bar{w} \land a\bar{\alpha} = \alpha \land \pi = \bar{\pi})}_{\text{Popping condition}}$$
$$\lor$$
$$\underbrace{(\alpha = A\omega \land \tau(A, \text{FIRST}_k(w)) = i \land p_i = A \to \delta \land \delta\omega = \bar{\alpha} \land \pi i = \bar{\pi} \land w = \bar{w})}_{\text{Deriving condition}}$$

The pop condition is if the left configuration has an input string and a pushdown store with the same first terminal. And the right configuration matches the left configuration, but the first terminal is popped from the stack and input.

The deriving condition is if the pushdown store α of the left LL configuration has a first element that is a nonterminal A. If the nonterminal together with the lookahead string results in a valid production index. And the right-hand side of the production is on top of the pushdown store on the right.

It is a possibility this definition does not match the definition in the paper fully [5, p. 6]. This is related to the notation used in Algorithm 13 [5, p. 15] which could be a mistake, this will be discussed later. Besides for this problem the relation will correspond to a single change in state during LL parsing.

This relation is expanded upon by introducing the relation $\vdash^*$ which is reflexive and transitive, therefore the relation may hold if.

$$(w, \alpha, \pi) \vdash^* (w, \alpha, \pi)$$

Or the relation may hold if.

$$(w_1, \alpha_1, \pi_1) \vdash^* (w_n, \alpha_n, \pi_n)$$
$$\iff$$
$$(w_1, \alpha_1, \pi_1) \vdash (w_2, \alpha_2, \pi_2) \vdash \cdots \vdash (w_n, \alpha_n, \pi_n)$$

The relation holds if and only if any of these conditions are fulfilled. This parsing relation can be used to define the following parser function $\phi : T^* \to \mathcal{S}$.

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

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

2.2 LLP(q, k) Parser Generator

2.2.1 LLP parsing

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 do this a definition for a LLP configuration will be needed.

Definition 2.6 (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 $\mathcal{L}(G)$[6], $x \in T^{*q}$[7], $y \in T^{*k}$. 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 pushdown store, α is the final pushdown store after parsing $(y, \omega, (\ )) = (vw, \omega, (\ )) \vdash^* (w, \alpha, \pi)$, $v \in T$, $w \in T^*$ and π are the resulting productions.

The idea now is, that you would start off by creating every pair $\mathcal{P}$ that can occur in the given input string. These pairs are defined to be the following.

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

Where $i \in \mathbb{N}$ denotes the index of where the start of the substring y_i starts, this i is used such that the ordering can be kept. Then we would want to create a table look up 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 2.5.

After all the x, y pairs have been constructed Φ is mapped over all the pairs. The idea is now to check if the configuration besides each other $((\omega, \alpha, \pi), i)$ and $((\bar{\omega}, \bar{\alpha}, \bar{\pi}), i + 1)$ matches on α and $\bar{\omega}$. This can be done using the associative **glue** binary operation, which is described in detail in the LLP paper [5, p. 7] or using Algorithm 18 [5, p. 18]. Using **glue** you can do a parallel reduce since it is associative.

This description will suffice for the theory needed to explain the important parts of the LLP paper [5] in relation to this paper.

[6]This is the set of all derivable strings $s \in T^* : S \Rightarrow^* s$ here $\Rightarrow^*$ is almost the same relation as $\Rightarrow$ but transitive and reflexive.

[7]This set is defined to be $\{t \in T^* : |t| \leq q\}$.

2.2.2 The PSLS definition

To construct Φ function Algorithm 8, 9 and 13 [5, pp. 13, 15] can be used where Algorithm 13 results in the final table which is Φ. There is just the problem that Algorithm 8 contains a mistake. This is due to Definition 6 [5, p. 12] which is the definition of the PSLS function.

The function PSLS : $T^* \times T^* \to \mathbb{P}((N \cup T)^*)$ is a function that finds the Prefix of a Suffix of a Leftmost Sentential form. This function is able to determine the initial pushdown store ω in an LLP configuration (ω, α, π) for the pair (x, y).

The trouble occurs when considering the following LL(2) grammar. Let $G = (S', \{\vdash, \dashv, a\}\{S', S, A\}, P)$ be an augmented grammar where P is a set of the following productions.

$$0)\ S' \to \vdash S \dashv \qquad 1)\ A \to \varepsilon \qquad 2)\ S \to aAa \qquad 3)\ A \to a$$

The augmentation comes from Algorithm 8 [5, p. 13] and introduces production 0. If you now were to use the definition from the paper as it is you would arrive at the following PSLS table.

	$\dashv$	$a \dashv$	aa
$\vdash$			$\{S\}$
$\vdash a$		$\{A\}$	$\{A\}$
aa	$\dashv$	$\{a\}$	

Table 1: The computed PSLS table using Algorithm 8 [5, p. 13].

To construct the table function that maps admissible pairs to LLP configurations, Algorithm 13 [5, p. 15] will be needed. This Algorithm needs a LL(k) parser so the following LL(2) table is constructed.

	$\vdash a$	aa	$a \dashv$
S'	$S' \to \vdash S \dashv$		
S		$S \to aAa$	
A		$A \to a$	$A \to \varepsilon$

Table 2: The LL(2) parsing table.

The trouble you run into is if you create the LLP configuration from the PSLS table entry $(\vdash a, a \dashv)$. Then Algorithm 13 states that you need to parse the first symbol of the lookahead string $a \dashv$ to obtain the final pushdown store and the production sequence. But when you try to parse $(a \dashv, A, ())$ then you get $(a \dashv, A, ()) \vdash (a \dashv, \varepsilon, 1)$ due to the parser table. Because of this you would never be able to parse the first symbol and Algorithm 13 would fail.

This can be fixed by changing the PSLS definition such that it is dependent on k, it could then be called PSLS_k instead. The dependency would result in FIRST_k being used instead of FIRST_1[8].

Definition 2.7 (PSLS_k)**.** Let $G = (N, T, P, S)$ be a context-free grammar. The function $\text{PSLS}_k(x, y)$ for a pair of strings $x, y \in T^*$ is defined as follows:[9]

$$\begin{aligned}\text{PSLS}_k(x, y) = \{\alpha : \exists S &\Rightarrow^*_{lm} wuA\beta \Rightarrow wxB\gamma \Rightarrow^* wxy\delta,\\ &w, u \in T^*, A, B \in N, \alpha, \beta, \gamma, \delta \in (N \cup T)^*, u \neq x,\\ &\alpha \text{ is the shortest prefix of } B\gamma \text{ such that } y \in \text{FIRST}_k(\alpha)\}\\ \cup \{y : \exists S &\Rightarrow^* wuA\beta \Rightarrow wxa\gamma \Rightarrow^* wxy\delta,\\ &a = \text{FIRST}_1(y), w, u \in T^*, \beta, \gamma, \delta \in (N \cup T)^*, u \neq x\}\end{aligned}$$

And Algorithm 8 would then be changed when solving for the shortest prefix.

Algorithm 2.3 (Construction of a collection of sets of LLP(q, k) items.)**.** Input: A context-free grammar $G = (N, T, P, S)$ that is LL(k). Ouput: A collection C of sets of LLP(q, k) items for G.

1. The grammar is augmented in the following way:

   $$G' = (N', T', P', S') = (N \cup \{S'\}, T \cup \{\vdash, \dashv\}, P \cup \{S' \to \vdash S \dashv\}, S')$$

   where S' is a new nonterminal symbol and $\vdash, \dashv$ are new terminal symbols.

2. The initial set of LLP(q, k) items is constructed as follows:

   (a) $D_0 := \{[S' \to \vdash S \dashv \bullet, u, \varepsilon, \varepsilon] : u \in \text{LAST}_q(\vdash S \dashv)\}$

   (b) $C := \{D_0\}$

3. If a set of LLP(q, k) items has been constructed, then a new set D_j is constructed for each symbol $X \in N' \cup T'$ standing just before the dot in D_i. The set D_j is constructed as follows:

   (a) $D_j := \{[Y \to \alpha \bullet X\beta, u_j, v_j, \gamma] : [Y \to aX \bullet \beta, u_i, v_i, \delta] \in D_i, u_j \in \bigcup_{\omega \in \text{BEFORE}_q(Y)\alpha} \text{LAST}_q(\omega), v_j \in \text{FIRST}_k(Xv_i), \gamma$ is the shortests prefix of $X\delta$ such that $v_j \in \text{FIRST}_k(\gamma)\}$.

[8]There are also some other changes made besides this which are mentioned in section 3.2.

[9]$\Rightarrow^*_{lm}$ is much the same as $\Rightarrow^*$ but only leftmost derivable strings are considered in the relation.

(b) If $[X \to \alpha Y \bullet \beta, u, v, \gamma] \in D_j, Y \in N'$ and $Y \to \delta \in P'$, then $D_j := D_j \cup \{[Y \to \delta \bullet, u', v, \gamma] : u' \in \bigcup_{\omega \in \text{BEFORE}_q(Y)\delta} \text{LAST}_q(\omega)\}$.

(c) Repeat step (3b) till no new item can be added into D_j.

(d) $C := C \cup \{D_j\}$

4. Repeat step (3) for all created sets till no new set can be added into C.

Using the new definition and algorithm would result in the following PSLS table.

	$\dashv$	$a \dashv$	aa
$\vdash$			$\{S\}$
$\vdash a$		$\{Aa \dashv\}$	$\{Aa\}$
aa	$\{\dashv\}$	$\{a \dashv\}$	

Table 3: The computed PSLS table using the new PSLS definition.

Using the new definition to create the LLP configuration from $Aa \dashv$ which is the table $(\vdash a, a \dashv)$ would now succeed. This is due to $(a \dashv, Aa \dashv, ()) \vdash (a \dashv, a \dashv, 1) \vdash (\dashv, \dashv, 1)$.

This changed definition would still work for LLP parsing. The difference is the prefix of a suffix of a leftmost sentential form found via PSLS is now guaranteed to be able to parse the first symbol. Since the pushdown store will always have enough symbols to derive the k first symbols of the input.

It is important to note that this argument needs some assumptions. Before these assumptions will be accounted for a typo in Algorithm 13 [5, p. 15] will be cleared up. This typo is the pushdown store and input string is switched in the 3-tuple because of the definition in the LLP paper [5, p. 5].

The assumption is that Algorithm 13 uses implicitly the lookahead in the LL configuration. This is assumed since only one symbol in the input string of the LL configuration appears. If the lookahead is not accounted for then LL(1) parsing would only be possible. This is a reasonable assumption since the paper says the following about their parallel LL parsing method.

"The method is not universal because only a subset of LL(k) grammars can be deterministically parsed in this way." [5, p. 2]

If Algorithm 13 only used LL(1) then it would be true the method is a subset of LL(k). It would then have been more precise to write their method only works for a subset of LL(1). Therefore, it is assumed that LL(k) parsing is meant to be used in Algorithm 13. Because of this assumption the example shown here with the original PSLS definition will fail.

2.2.3 Infinite loop

There is an infinite loop in Algorithm 8 [5, p. 13]. It will be shown in this section that the infinite loop can occur. Consider the following augmented grammar.

$$S' \to \vdash S \dashv \quad S \to aSA \quad A \to \varepsilon \quad S \to b$$

This grammar is LL(1), so you should be able to use Algorithm 8 to determine the PSLS table and therefore determine if it is LLP$(1,1)$. If you try to construct the LLP item collection for the given grammar with $q = 1$ and $k = 1$ what you find is.

$$\begin{aligned} D_0 &= \{[S' \to \vdash S \dashv \bullet, \dashv, \varepsilon, \varepsilon]\} \\ D_1 &= \{[A \to \bullet, b, \dashv, \dashv], [S \to aSA\bullet, b, \dashv, \dashv], \\ &\quad [S \to b\bullet, b, \dashv, \dashv], [S' \to \vdash S\bullet \dashv, b, \dashv, \dashv]\} \\ &\vdots \end{aligned}$$

The problem will occur due to the tuple $[S \to aSA\bullet, b, \dashv, \dashv]$. This item will become a singleton set when grouping the set D_1 in the start of step 3. In step 3 (a) you are able to create the tuple $[S \to aS\bullet A, b, \dashv, A \dashv]$. This is due to:

$$\begin{aligned} &b \in \bigcup_{\omega \in \text{BEFORE}_1(S)aS} \text{LAST}_1(\omega) = \bigcup_{\omega \in \{\vdash aS, aaS\}} \text{LAST}_1(\omega) = \text{LAST}_1(S) = \{b\} \\ &\dashv \in \text{FIRST}_1(A \dashv) \text{ since } A \Rightarrow^* \varepsilon \\ &A \dashv \text{ is the shortests prefix of } A \dashv \text{ such that } \dashv \in \text{FIRST}_1(A \dashv) \end{aligned}$$

Because $[S \to aS\bullet A, b, \dashv, A \dashv]$ occurs, then the production $S \to aSA\bullet$ also appears in some item in D_j due to step 3 (b). This item has the same prefix and same shortest prefix as $[S \to aS\bullet A, b, \dashv, A \dashv]$. The suffix b will also appear in the next tuple because.

$$\bigcup_{\omega \in \text{BEFORE}_1(S)aSA} \text{LAST}_1(\omega) = \bigcup_{\omega \in \{\vdash aSA, aaSA\}} \text{LAST}_1(\omega) = \text{LAST}_1(S) = \{b\}$$

Because of this $[S \to aSA\bullet, b, \dashv, A \dashv]$ is a member of the new set D_j which is:

$$\begin{aligned} \{&[A \to \bullet, b, \dashv, A \dashv], [S \to aS\bullet A, b, \dashv, A \dashv], \\ &[S \to aSA\bullet, b, \dashv, A \dashv], [S \to b\bullet, b, \dashv, A \dashv]\} \end{aligned}$$

The problem is since the production, suffix and prefix is the same as the tuple $[S \to aSA\bullet, b, \dashv, \dashv]$ the same item set will be created but with a different shortest prefix. The new shortest prefix will always be prepended by A such that $\dashv$ can be derived. This results in this computation can be repeated infinitely. If you compute the next set using $[S \to aSA\bullet, b, \dashv, A\dashv]$ you will arrive at:

$$\{[A \to \bullet, b, \dashv, AA\dashv], [S \to aS\bullet A, b, \dashv, AA\dashv],\\ [S \to aSA\bullet, b, \dashv, AA\dashv], [S \to b\bullet, b, \dashv, AA\dashv]\}$$

Therefore the item $[S \to aSA\bullet, b, \dashv, A^*\dashv]$ will appear infinitely many times in different sets. Algorithm 8 only terminates if a fixed point is reached, but this is impossible, so it will not halt.

This algorithm was not fixed in this paper, instead a grammar is checked for the following. Let $G = (N, T, P, S)$ be a context-free grammar, this grammar may cause an infinite loop if.

$$\exists \mathcal{N} \in N : \alpha \in (N \cup T)^* \wedge A \in N \wedge \mathcal{N} \Rightarrow^* \alpha AA \wedge A \Rightarrow^* \varepsilon$$

This predicate should not capture the essence of the problem. It is intended as a way to make sure not infinitely many nullable nonterminals are prepended to the shortest prefix.

2.2.4 Determining if a grammar is LLP

A common answer to whether or not a grammar is a LL(k) grammar is: if the LL(k) parser 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) parser can be constructed.

The first step in determining if a grammar is a LLP(q, k) grammar is if it is in the LL(k) 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 [5, p. 13] is used for. To determine if the grammar is LLP.

Definition 10 [5, p. 13] uses the PSLS(x, y) [5, p. 12] values to determine the initial pushdown stores which can be used to determine the LLP configuration. The trouble 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 $\mathrm{LLP}(1,1)$, when computing $\mathrm{PSLS}_1(b, b)$. It can be seen that there exists the following occurrences of bb which leads to two different initial pushdown stores.

$$A \Rightarrow^*_{lm} (abb)^*A \Rightarrow (abb)^*abb_xB_{B\gamma} \Rightarrow^* (abb)^*abb_xb_y$$

This derivation[10] corresponds to the first set in the $\mathrm{PSLS}_1(b, b)$ definition. The shortest prefix of B is B where $b \in \mathrm{FIRST}_1(B)$ so $B \in \mathrm{PSLS}_1(b, b)$ [6, p. 2] by definition. The other occurrence comes from the last set in the PSLS definition.

$$A \Rightarrow^*_{lm} (abb)^*A \Rightarrow (abb)^*ab_xb_yB \Rightarrow^* (abb)^*ab_xb_yB$$

Since $\mathrm{FIRST}_1(b) = \{b\}$ then $b \in \mathrm{PSLS}(b, b)$ by definition. The initial pushdown store for the admissible pair (b, b) is $\mathrm{PSLS}_1(b, b) = \{b, B\}$. Therefore, by Definition 10 [5, p. 13] this grammar is not $\mathrm{LLP}(1,1)$. If one were to check for all admissible pairs then they would find that $\mathrm{PSLS}_1(b, b)$ is the only problem pair. If one wishes to parse the grammar a $\mathrm{LLP}(2,1)$ parser can be used. It solves the ambiguities since $\mathrm{PSLS}_1(ab, b) = \{b\}$ and $\mathrm{PSLS}_1(bb, b) = \{B\}$.

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

$$S \to [S] \qquad S \to \varepsilon$$

This grammar seems like it is not $\mathrm{LLP}(q, k)$ for any $q, k \geq 1$. 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 $\mathrm{LLP}(1,1)$ grammar because the LLP parser only uses the shortest prefix of the initial pushdown store in the LLP configuration. This can be determined as not a problem by using the PSLS definition.

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

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

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

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

$$S \to aaS \qquad S \to \varepsilon$$

This grammar is mentioned in the LLP paper [5, 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), that could lead to the possible initial pushdown stores are $\text{PSLS}_k(a^q, a^k) = \{S, a(S|\epsilon)\}$. 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 $\text{PSLS}_k(a^q, a^k)$ become a singleton. If the productions for the grammar were.

$$S \to aS \qquad S \to \varepsilon$$

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

3 Implementation

3.1 Structure

The parser generator was chosen to be written in Haskell. The reason for doing so is the author likes the language. Another reason is also when dealing with parsing, Haskell seems to be commonly used, so this tool seems to belong in this toolbox.

The generated parser comes in the form of a Futhark source file. The reason for doing so is Futhark is designed for "parallel efficient computing" [2] that can be executed on a GPU. This is a good choice since the parser that will be generated is designed for parallel parsing on a GPU. Futhark is also in the ML family which helps with readability and makes it easy for the author to write efficient GPU code. These advantages come in the form of abstractions like non-recursive sum types and pattern matching. The structure of the parser generator is.

1. Parse a context-free grammar.
2. If the grammar has common left factors, left recursion, predicate from section 2.2.3 or the LL(k) table can not be constructed fail.
3. Construct PSLS table using Algorithm 9 and the new Algorithm 8 [5, p. 13].
4. Use Algorithm 13 [5, p. 13] to construct the LLP(q, k) table.

5. Add to the LLP(q, k) table, table entries (ε, w) which maps to the LLP configuration $(S', S \dashv, 0)$ where $w \in \text{FIRST}_k(S')$
6. Make a Futhark source file which contains a LLP(q, k) table and algorithm 18.

3.2 Assumptions

At times the notation of the LLP paper [5] was unknown to the author of this paper. This was cause of problems when trying to implement the algorithms. Therefore the following assumptions about the LLP paper [5] are mentioned here.

Algorithm 8 [5, p. 13] has the following notation.

step 2. (a): "$\{[S' \to \vdash S \dashv \bullet, u, \varepsilon, \varepsilon]\}, u = \text{LAST}_q(\vdash S \dashv)$"
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 [5, p. 14]. It is therefore assumed that for each element in the u and u' sets, an item is constructed from them i.e.

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

The second type of notation in Algorithm 8 is.

step 3. (a): "$u_j \in \text{LAST}_q(\text{BEFORE}_q(Y)\alpha)$"
step 3. (b): "$u' = \text{LAST}_q(\text{BEFORE}_q(Y)\delta)$"

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 in turn 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.

step 3. (a): $u_j \in \bigcup_{\omega \in \text{BEFORE}_q(Y)\alpha} \text{LAST}_q(\omega)$

step 3. (b): $u' \in \bigcup_{\omega \in \text{BEFORE}_q(Y)\delta} \text{LAST}_q(\omega)$

Another assumption made in algorithm 8 is when G' is defined.

$$\text{“}G' = (N \cup S', T \cup \{\vdash, \dashv\}, P \cup \{S' \to \vdash S \dashv\}, S')\text{”}$$

It is assumed that N, T or P now refers to the sets with which includes the elements they are unionized with. Since if not, then step 3. (a) would result in the empty set in the first iteration.

In Example 12 [5, p. 14] the tuple in the set E_2 should be $T \to [\bullet E]$.

3.3 Memoization of FIRST and LAST

The FIRST implementation corresponds to Algorithm 2.1. The LAST implementation corresponds to the one mentioned in the LLP paper [5, p. 12], which uses an existing FIRST implementation. A big problem with FIRST is it is extremely expensive because of the truncated product step.

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

This inefficiency can be solved by the use of memoization since FIRST is recursively defined and has overlapping subproblems. In Haskell this can be done using a `State` monad from the `mtl` library. As an example the difference in performance can be seen in the before[11] and after[12] tests. In the tests at the time this optimization was made, 25 LLP(q, k) parsers where $q, k \in \{1, 2, 3, 4, 5\}$, FIRST$_k$ where $k \in \{1, 2, \ldots, 20\}$ and FOLLOW$_k$ where $k \in \{1, 2, \ldots, 7\}$ are generated. All of these functions are also used on some generated input. The difference in performance of using memoization can be seen below.

	Time
Before memoization	4h 16m 27s
After memoization	21s

Table 4: The time it takes to run the tests before and after memoization was implemented at the time. These times are taken from Github actions results. No further comparisons was done since the speed of the parser generator was not the main objective with this project.

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

[11] `db564853c2fe4cd1c3d2839795738c56dbb209a0` is the SHA of the version without memoization.

[12] `daa9d6ff4640ca5683cec1d7956d2aa7ecd89c47` is the SHA of the version with memoization.

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

Figure 1: Grammar that generates every combination of its n terminals.

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. 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 [5, 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. In the original Algorithm 8 it was said that.

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.

The way this could be solved is by doing a breadth first search (BFS) on all the possible prefixes to see if the a can be derived. Then just choose the shortest of the prefixes that can derive $a\omega$. The problem here is you would need a condition for stopping the BFS such that it does not try to derive something that can not appear.

An idea is to check if the first symbol of a given derivation has been visited before, if so then skip the derivation. A problem that may occur here is if $X\delta = YYa$ and $Y \Rightarrow^* \varepsilon$ then since Y is visited on the second derivation then that path is not further explored. This was at some point the used implementation in this project and was a cause of trouble It was later realized step 3. had a mistake, and it should say something like.

step 3. (a): γ is the shortest prefix of $X\delta$ such that $\gamma \Rightarrow^* v_j\omega$.

But an easier idea was to create all prefixes of γ and compute the FIRST_k of these prefixes. γ is then the shortest prefix where $a \in \text{FIRST}_k(\gamma)$. This implementation should is easy to comprehend and fast because of the use of memoization in the FIRST. Due to this the Algorithm 8 was changed to have a step with.

step 3. (a): γ is the shortest prefix of $X\delta$ such that $v_j \in \text{FIRST}_k(\gamma)$.

3.5 Parser

When the parser generator has created a table, this table will need to be represented as code Futhark somehow. And a parallel bracket matching algorithm to check the LLP configurations match. These problems will be solved in this section.

3.5.1 String Packing

The first problem is how do we represent strings within Futhark. Futhark does not have dynamic arrays and is not able to represent arrays of different lengths in an array. This is because Futhark needs to be able to know how much memory will be allocated. The reason, this is a problem, is the function `key_to_config` in the parser acts as a table look up which returns the corresponding LLP configuration. These LLP configuration may result in differently sized pushdown stores or number of productions (α, ω, π).

This is actually not completely true since instead of using the LLP configuration (α, ω, π) as is, the list homomorphisms in algorithm 18 [5, p. 18] which results in the tuples $(RBR(\alpha)LBR(\omega^R), \pi)$ besides for the starting pairs $(\varepsilon, \vdash w)$ where $w \in T^*$ which becomes $(LBR(\omega^R), \pi)$. This is done such that step 2. from Algorithm 18 [5, p. 18] can be precomputed.

Since `key_to_config` is mapped over an array of (x, y) input pairs this will result in the Futhark compiler not knowing how much memory needs to be allocated. This is also a problem since in the parser generator terminals and nonterminals are strings, so they may have different lengths.

To solve this problem string padding can be used. The way this is done is every terminal string is assigned an index $i \in \{0, 1, \ldots, |T| - 1\}$ where $|T| - 1 < 2^{32} - 1$. This is because the largest 32-bit value of $2^{32} - 1$ is used as string padding. When dealing with a sequence of production then the indexes $i \in \{0, 1, \ldots, |P| - 1\}$ where $|P| - 1 < 2^{32} - 1$ for the same reason as for terminal strings. When a string is made of nonterminals and terminals then each nonterminal is are assigned an index from $i \in \{|T| - 1, |T|, |N| + |T| - 1\}$ where $2^{32} - 1 \leq i \leq 2^{64} - 1$. This is done since instead of assigning a specific integer as being padding the non-recursive sum types from Futhark can be used. This is a good choice since they are already used to label an element on the pushdown store is a `#left` or `#right` bracket.

These considerations result in the following table look up function which is created by the parser generator. This function does pattern matching on the given input pair.

```
def key_to_config (key : ((u32), (u32))) : maybe ([]bracket, []u32) =
    match key
    case ((0), (1)) → #just ([#right 5, #right 1], [3])
    case ((0), (2)) → #just ([#right 5, #left 5], [1])
    case ((0), (3)) → #just ([#right 5, #epsilon], [2])
    case ((2), (1)) → #just ([#right 5, #right 1], [3])
    case ((2), (2)) → #just ([#right 5, #left 5], [1])
    case ((2), (3)) → #just ([#right 5, #epsilon], [2])
    case ((3), (1)) → #just ([#right 1, #epsilon], [u32.highest])
    case ((4294967295), (0)) → #just ([#left 1, #left 5], [0])
    case _ → #nothing
```

The returned configurations are inside a `Maybe`-like type. So if at any point `#nothing` is returned then an invalid string pair was returned and therefore (x, y) was not an admissable pair. This table function above arises from the following grammar where each subscript correspondx to the integers.

$$0.\ S'_0 \to \vdash_0 A_1 \dashv_1 \qquad 1.\ A_1 \to a_2 A_1 \qquad 2.\ A_1 \to b_3 \qquad 3.\ A_1 \to \varepsilon$$

3.5.2 Bracket Matching

Besides this the Futhark implementation matches Algorithm 18. A missing piece is the parallel bracket matching which is not described in the LLP paper [5, text]. The implementation used takes a lot of inspiration from the implementation described on the Futhark [1]. The differences are a balancing check is made as can be seen below.

```
def depths [n] (input : [n]bracket) : maybe ([n]i64) =
    let left_brackets =
      input
      ▷ map (is_left)
    let bracket_scan =
      left_brackets
      ▷ map (\b → if b then 1 else -1)
      ▷ scan (+) 0
    let result =
      bracket_scan
      ▷ map2 (\a b → b - i64.bool a) left_brackets
    in if any (<0) bracket_scan || last bracket_scan ≠ 0
    then #nothing
    else #just result
```

This is done using `bracket_scan` which must only contain positive values and the last value must be zero. If these predicates are not fulfilled then it is known the brackets are not balanced. The returned result is the `bracket_scan` returned such that it is adjusted for the left brackets being one then the right. Here the non-recursive sum-type from Futhark are utilized once again where `#nothing` is returned if an error occurs.

Something to note is the glue [5, p. 7] reduce could have been used instead of algorithm 18 [5, p. 18]. This is a bad choice for two reasons, the first is it is slow [5, 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.

Assuming the number of processes used is the number of terminals then the LLP(q, k) parsing algorithm using bracket matching is $O(\log n)$ where n is the number of terminals in the input string [5, p. 19].

4 Testing

4.1 FIRST and FOLLOW

Unit testing with some property based testing are used to ensure the first and follow sets are computed correctly by Algorithm 2.1 and 2.2. The structure of the first and follow set tests is some small hard coded tests and tests which assert the first and follow set definition is fulfilled. The small tests are meant to be easy to comprehend and help guide the programmer with large errors. While the first and follow set tests are used to assert the correctness.

4.1.1 Unit tests

The unit tests used to assert the correctness of the first and follow sets are done by comparing the computed sets with precomputed sets of some grammars. The existing results were taken from [4, 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 [5, p. 5]. These grammars are complex enough to cover different possible properties of grammars which can influence the sets. Such as nonterminals followed by each other and nullable nonterminals.

4.1.2 Property based like testing

These test are like property based tests which tests for some property, but they tested on random input. The property tested for is if the functions can reconstruct the LLP paper definition of first and follow [5, p. 5].

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

This can be done by implementing the naive implementation described later. For the property based tests $k \in \{1, 2, 3, 4\}$ is used for two grammars [4, pp. 62, 63]. These tests could have been more extensive with more k values or random grammars. This was never done because the naive first and follow implementations becomes extremely slow for larger k.

FIRST$_k$ testing

The FIRST$_k$ definition can be naively implemented by doing a breadth first search on derivable strings for each nonterminal of a grammar. If the first sets of all nonterminals are reproducible by Algorithm 2.1 then the FIRST$_k$ function is computed correctly for the given grammar. The reason for this is you are able to compute any FIRST$_k$ of a string in a grammar if you know FIRST$_k$ of any nonterminal.

FOLLOW$_k$ testing

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 FOLLOW$_k(A)$ set. If all FOLLOW$_k(A)$ are reproducible by Algorithm 2.2 then the FOLLOW$_k$ function is computed correctly for the given grammar.

4.2 LL(k)

To test the LL(k) parser, integration tests with some property based like testing is used, since once again these tests are not random. The integration testing comes in the form of ensuring that a working parser is created from the first and follow sets.

One of the integration tests are creating LL(k) parsers with $k \in \{1, \dots, 5\}$ for a grammar [4, p. 45]. Then a parsable string is parsed with a known production sequence. This is done to check if the parser can parse strings correctly. This is the only test done which check for a correct sequence. A better test would have been to compute the sequence order of every derivable string and see if the parser produces the same sequence. This was never done because it was not believed this kind of testing was needed at the time.

The property based like test only checks for if every leftmost derivable string of a given length will result in a sequence. And every string that is not left most derivable will result in an error.

4.3 LLP(q, k)

To assert the parser generators works as intended, integration tests with property based testing is used. And some small hard coded tests are used to help debug for larger problems. Besides these tests there are also the property based tests which are meant as a way to guarantee the parser generator works as intended.

4.3.1 Unit tests

Some precomputed examples from the LLP paper [5] 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, that computing the LLP item collection by hand can be quite a daunting task. Therefore, the LLP collection in Example 11 [5, p. 14] and the PSLS table in Example 12 [5, p. 14].

Another test starts by constructing nine LLP(q, k) parsers where $q, k \in 1, 2, 3$ using the grammar in Example 11 [5, p. 14]. 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. These parsers are also tested against a single string to check if a hard coded production sequence can be reproduced.

4.3.2 Property based testing

For the property based testing, random LLP(q, k) grammars will be needed. These are found by creating random grammars and checking if the parser generator will create a parser.

Stuck testing

The first kind of property to test for, is if the parser generator is able to get stuck in an infinite loop. This will need to be tested for since the parser generator uses fixed point iterations. To get an idea if the parser generator gets stuck one thousand LLP$(1, 1)$ grammars are generated and their parsers are created. These grammars are randomly generated and have three terminals, three nonterminals and six productions. If all of these parsers are not created within 5 hours, then it is assumed the parser generator can get stuck.

Parsing testing

The property tested for is if an arbitrary LLP(q, k) parser is able to parse leftmost derivable strings of a given length and fail on any other string. Besides this the production sequence computed also needs to match the sequence used to derive the string that is being parsed.

This property would translate to let $G = (N, T, P, S)$ be a LLP(q, k) grammar, $\varphi : T^* \to \mathcal{S}$ be a LL(k) parser and $\Pi : T^* \to \mathcal{S}$ be a LLP(q, k) parser. These parser functions generate the production sequence if the input can be parsed else the empty sequence is produced. The property tested for is then[13].

$$\forall s \in T^* : \Pi(s) = \varphi(s)$$

These tests are done for 50 random LLP$(1,1)$ grammars, 50 LLP$(2,2)$ and 50 LLP$(3,3)$ grammars. Where the leftmost derivable strings are of length 20 or less and not leftmost derivable string are of length 6 or less.

Problems with this method

The LLP grammars are generated by rejecting grammars if they are not LL(k), or they just have a single common left factor. Or if the infinite loop property from Section 2.2.3 is fulfilled. The concern is if almost every grammar just gets rejected.

When generating random grammars with three terminals, three nonterminals and six productions with a right-hand side with a length of at max three. And the three of the productions must use the nonterminals as their left-hand side. Because of this the grammars are not uniformly distributed, but they are more likely to be useful. Using these specifications we see the following percentage of random grammars are accepted as LLP.

Lookback	Lookahead	Acceptance Percentage
1	1	7.46%
2	2	9.43%
3	3	10.40%

Table 5: The percent of grammars accepted when trying to generate 1000 LLP(q,k) grammars.

As one can see when increasing the lookback and lookahead more grammars are accepted which is expected due to the grammar class becoming larger. It is assumed that the tests are working as expected because it does not seem like that many grammars gets rejected.

[13] $S \Rightarrow_{lm}^{*} s$ means all leftmost derivable strings from the start of the grammar. $\Rightarrow_{lm}^{*}$ allows only for left derivations and is reflexive and transitive.

5 Conclusion

The LLP parser generator works due to it being thoroughly tested using multiple random grammars. But the parser generator does not work as intended since it does reject grammars which may be LLP(q, k) because of the infinite loop check. Nevertheless, the implementation is still acceptable.

The parser generator is missing a way of constructing the concrete syntax trees. The time for this was never found because a multitude of problems with the theory and implementation of the parser generator. This is most likely due to the author of this paper being inexperience with parsing.

During this project it was also realized that there are two mistakes in the LLP paper [5]. It is believed that the two counter examples shown in this paper should convince a reader that there are mistakes in the algorithms.

The usability of the grammar class LLP(q, k) is at the moment unknown for $q > 1$ and $k > 1$. It is known a LLP$(1, 1)$ parser generator can be used for JSON and a small C-like language [7]. These implementations come with limitations, an example is JSON is not LLP$(1, 1)$ [7, p. 60], so modifications are made such that JSON can be parsed. The hope is that by creating a LLP(q, k) parser this can help mitigate these limitations.

References

[1] Futhark Website Contributors. *Matching parentheses*. https://github.com/diku-dk/futhark-website. [Online; accessed 31-May-2023]. 2023. URL: https://futhark-lang.org/examples/parens.html.

[2] Futhark Website Contributors. *The Futhark Programming Lnaguage*. https://github.com/diku-dk/futhark-website. [Online; accessed 31-May-2023]. 2023. URL: https://futhark-lang.org.

[3] 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.

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

[5] 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.

[6] 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.

[7] Robin Voetter. “Parallel Lexing, Parsing and Semantic Analysis on the GPU”. MA thesis. Leiden University, 2021.

[8] 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.

A PSLS Glueable proof

It will be shown that given two neighbouring admissible pairs (x_1, y_1) and (x_2, y_2) will always result in LLP configurations that are glueable. Consider the following substring $cdefg$ where $c, e, g \in T$ and $d, f \in T^*$. The admissible pairs in this case are (cd, ef) and (de, fg). First the initial pushdown store is computed using $\text{PSLS}_k(cd, ef)$ and $\text{PSLS}_k(de, fg)$. The pair (cd, ef) can result in an initial pushdown store due to the first or second set in the PSLS definition. This would mean the following condition may hold.

s_1) $\alpha : \exists S \Rightarrow^*_{lm} wuD\kappa \Rightarrow wcdA\sigma \Rightarrow^* wcdefg\delta$
where α is the shortest prefix of $B\gamma$ such that $ef \in \text{FIRST}_k(\alpha)$

s_2) $ef : \exists S \Rightarrow^* wuD\kappa \Rightarrow wcda\sigma \Rightarrow^* wcdefg\delta$
where $a = \text{FIRST}_1(ef)$

And for the pair (cd, ef) it may hold that.

e_1) $\beta : \exists S \Rightarrow^*_{lm} wvC\mu \Rightarrow wcdeB\gamma \Rightarrow^* wcdefg\delta$
where β is the shortest prefix of $B\gamma$ such that $fg \in \text{FIRST}_k(\beta)$

e_2) $fg : \exists S \Rightarrow^* wuD\kappa \Rightarrow wcdeb\gamma \Rightarrow^* wcdefg\delta$
where $b = \text{FIRST}_1(fg)$

If you consider the case that (cd, ef) results in s_1 being true then e_1 and/or e_2 may hold. If e_1 holds then it must be true that.

$$A\sigma \Rightarrow^*_{lm} eB\gamma$$

Then in the context of LL parsing it must mean $(ef, A\sigma, ()) \vdash^* (f, B\gamma, \pi)$. It then also must be true that $(ef, \alpha, ()) \vdash^* (f, \chi, ())$ where $\chi\nu = \beta$ where

$\chi, \nu \in (T \cup N)^*$. Then since the final pushdown store χ is a prefix of/or equal to the initial pushdown store β they are glueable.

If s_1 and e_2 holds.

$$A\sigma \Rightarrow^*_{lm} b\gamma$$

Then in the context of LL parsing it must mean $(ef, A\sigma, ()) \vdash^* (f, b\gamma, \pi)$. It then also must be true that $(ef, \alpha, ()) \vdash^* (f, n, ())$ where $nm = fg$ and $n, m \in T^*$. Then since the final pushdown store n is a prefix of/or equal to the initial pushdown store β they are glueable.

The same kind of argument can be used for the case that s_2 and e_1 and/or s_2 and e_2 holds.