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**

2 Theory **2**
2.1 LL(k) Parser Generator . . . 2
2.1.1 FIRST$_k$ and FOLLOW$_k$ . . . 2
2.1.2 LL Parsing . . . 5
2.2 LLP(q, k) Parser Generator . . . 6
2.2.1 LLP parsing . . . 6
2.2.2 The PSLS definition . . . 7
2.2.3 Determining if a grammar is LLP . . . 9

3 Implementation **11**
3.1 Structure . . . 11
3.2 Assumptions . . . 11
3.3 Memoization of FIRST and LAST . . . 13
3.4 LLP collection of item sets . . . 14
3.5 Parser . . . 14
3.5.1 String Packing . . . 14
3.5.2 Bracket Matching . . . 16
3.5.3 Complexity . . . 16

4 Testing **16**
4.1 First and Follow sets . . . 16
4.1.1 Unit tests . . . 16
4.1.2 Property based testing . . . 17
4.2 LL(k) . . . 17
4.3 LLP(q, k) . . . 18
4.3.1 Unit tests . . . 18
4.3.2 Property based testing . . . 18

5 Conclusion **19**

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 [5, p. 5] and a LL(k) parser generator is needed.

2.1.1 FIRST$_k$ and FOLLOW$_k$

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" [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] [7].

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 $\omega, \delta \in (T \cup N)^*$[3]. 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 $\alpha, \beta \in (N \cup T)^+$[4]

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

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

[4] $Z^+ \stackrel{\text{def}}{=} Z^* \backslash \{\epsilon\}$

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 and $\text{FIRST}_k : (N \cup T)^* \to \mathbb{P}(T^*)$. The first-sets for a given string 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 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 and $\text{FOLLOW}_k : N \to \mathbb{P}(T^*)$. The follow-set for a given nonterminal can be solved as followed.

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

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 $A \in N$ 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 $A \in N$.

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

Using FIRST_k and FOLLOW_k a LL(k) table which maps nonterminals on the stack and the lookahead of k terminals to a production index.

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

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

If for a given grammar it holds that $|\tau(A, s)| = 1$ for all $(A, s) \in N \times T^*$ then the grammar has no table conflicts.

2.1.2 LL Parsing

To describe how this 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 $\mathcal{S}$[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.

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

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. To define how such a function is created the parsing relation is defined. This relation holds if the left LL configuration can turn into the right LL configuration after parsing the left configuration.

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

The pop condition holds if the left configuration has an input string and a push down store with the same first terminal which is popped in the right configuration.

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

The deriving condition holds if the left LL configuration push down store α has a first element that is a nonterminal A. If the nonterminal together with the lookahead string results in a valid production index. Where the right-hand side of the production is on top of the push down 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 or a different notation, this will be discussed later. Besides for this problem the relation will correspond to a single 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 condition are fufilled. This parsing relation can be used to define the following parsing function $\phi : T^* \to \mathcal{S}$.

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

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 $\mathcal{L}(G)$, $x \in T^{*q}$, $y \in T^{*k}$. 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.

The idea now is 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 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 2.5.

After all the x, y pairs have been constructed Φ is mapped over alle the pairs. The idea is now to check if the configuration besides each other $((\omega, \alpha, \pi), i)$ and $((\bar{\omega}, \bar{\alpha}, \bar{\pi}), i + 1)$ describes matches on α and $\bar{\omega}$. This is done using the associative **glue** binary operation described in detail in the LLP paper [5, p. 7] or using Algorithm 18 [5, p. 18]. This description will suffice for the theory needed to explain the important parts of LLP paper [5] in relation to this paper.

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 have 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 Suffix of Leftmost Sentential form for a pair of string (x, y). This function is able to determine the initial push down 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.

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

The augmentation comes from Algorithm 8 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 the original PSLS definition.

To construct the table function that maps admissible pairs to LLP configuration Φ Algorithm 13 will be needed. This Algorithm needs a LL(k) parser so the following LL(2) table is constructed.

	$\vdash a$	aa	$a \dashv$
S'	$S' \rightarrow \vdash S \dashv$		
S		$S \rightarrow aAa$	
A		$A \rightarrow a$	$A \rightarrow \epsilon$

Table 2: The $LL(2)$ parsing table.

The trouble you run into is if you want to create the LLP configuration from A which has the table keys $(\vdash a, a \dashv)$. Then Algorithm 13 states that you need to parse the first symbol of the lookahead string $a \dashv$ to obtain the final push down store and the production sequence. But when you try to parse $(a \dashv, A, ())$ then you get $(a \dashv, A, ()) \vdash (a \dashv, \epsilon, 1)$ 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 that it is dependent on k meaning it could be called PSLS_k instead. The dependency would result in FIRST_k being used instead of FIRST_1. And Algorithm 8 would be changed when solving for the shortest prefix such that.

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

Using the new definition 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 has the table keys $(\vdash a, a \dashv)$ would now succeed. This is due to when parsing

$(a \dashv, Aa \dashv, ())$ then you get $(a \dashv, Aa \dashv, ()) \vdash (a \dashv, a \dashv, 1) \vdash (\dashv, \dashv, 1)$ due to the parser table.

This changed definition would still work for LLP parsing. The difference is the initial push down store found via PSLS is now guaranteed to be able to parse the first symbol of the lookahead.

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

The assumption is in Algorithm 13 uses implicitly the lookahead in the LL configuration. This is assumed since only one symbol is in the input string of the LL configuration. 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 had 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 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 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 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[6] corresponds to the first set in the PSLS(b, b) definition. The shortest prefix of B is B where FIRST$_1(b) \subseteq$ FIRST$_1(B)$ so $B \in$ PSLS(b, b) [6, p. 2] by definition. The other occurrence comes from the last set in the PSLS definition.

$$A \Rightarrow^*_{lm} (abb)^* A \Rightarrow (abb)^* ab_x b_y B \Rightarrow^* (abb)^* ab_x b_y B$$

Since FIRST$_1(b) = \{b\}$ then $b \in$ 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 [5, 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.

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

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 FIRST$_1(S) = \{\epsilon\}$, but FIRST$_1(]^k) \subseteq$ FIRST$_1(S])$ holds. Therefore, this is not a reason for the grammar not being LLP(q, k).

[6] 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.

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

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) could lead to the possible initial push down stores are 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 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 PSLS$(a^q, a^k) = \{S\}$ 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 and the advisor to this project has written a lot of Haskell through the Futhark project. Another reason is also when dealing with parsing Haskell seems to be a 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. The 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.

3.2 Assumptions

At times the notation of the LLP paper [5] 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 [5] are mentioned here.

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

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

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.

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

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.

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

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.

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 [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.

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

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[7] and after[8] tests. In the tests at the time 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 adding 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 at the time memoization was implemented. 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 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.

[7] `db564853c2fe4cd1c3d2839795738c56dbb209a0` is the SHA of the version with memoization.

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

$$S \to \epsilon | a_1 S | a_2 S | a_3 S | \cdots | a_n S \tag{1}$$

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

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. The first implementation detail to consider is.

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

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 $Y \Rightarrow^* \epsilon$ then since Y is visited on the second derivation then that path is not further explored. This was at some point the used implementation and was a cause of trouble.

An easier idea was to create all prefixes of γ and compute the FIRST_1 of these prefixes. γ is then the shortest prefix where $a \in \text{FIRST}_1(\gamma)$. This implementation should is easy to comprehend and fast because of the use of memoization in the FIRST.

3.5 Parser

When the parser generator has created a table, this table will be needed to be represented as code Futhark somehow. And a way of Glueing the configurations in Futhark will be needed. These problems will be answered 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 lookup 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 homomorphism in algorithm 18 [5, 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 $(LBR(\omega^R), \pi)$. 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 string, so they may have different lengths.

To solve this problem string padding can be used, the way this is done when dealing with terminal string every terminal 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 terminal strings. When a string is made of nonterminals and terminal then each nonterminals 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.

```
1  def key_to_config (key : ((u32), (u32))) : maybe ([]bracket, []u32) =
2      match key
3      case ((0), (1)) → #just ([#right 5, #right 1], [3])
4      case ((0), (2)) → #just ([#right 5, #left 5], [1])
5      case ((0), (3)) → #just ([#right 5, #epsilon], [2])
6      case ((2), (1)) → #just ([#right 5, #right 1], [3])
7      case ((2), (2)) → #just ([#right 5, #left 5], [1])
8      case ((2), (3)) → #just ([#right 5, #epsilon], [2])
9      case ((3), (1)) → #just ([#right 1, #epsilon], [u32.highest])
10     case ((4294967295), (0)) → #just ([#left 1, #left 5], [0])
11     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 correspond to the integers.

$$0.\ S_0' \rightarrow \vdash_0 A_1 \dashv_1 \qquad 1.\ A_1 \rightarrow a_2 A_1 \qquad 2.\ A_1 \rightarrow b_3 \qquad 3.\ A_1 \rightarrow \epsilon$$

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 paper. The implementation used takes a lot of inspiration from the implementation described on the Futhark [1]. The differences are a balancing check is made before the grading and the tabulation is never done. Instead, simply checking if the index pairs $(2i, 2i + 1)$ are the same types of brackets.

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.

3.5.3 Complexity

4 Testing

4.1 First and Follow sets

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.

4.1.2 Property based testing

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 $\text{FOLLOW}_k(A)$ 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.

4.2 LL(k)

To test the LL(k) parser, integration tests with some property based testing is used. The integration testing comes in the form of ensuring that when using the first and follow-set functions result in a parser that can give the correct sequence of productions.

One of the integration tests are creating LL(k) parsers with $k \in \{1, \ldots, 5\}$ for a grammar [4, p. 45] found in "Introduction to Compiler Design". 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 correct sequences. A better test would have been to compute the sequence order of every derivable string and see if the parser produces the same sequence.

The property based 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 integrations tests with property based testing. Some small hard coded tests which are used to help debugger 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 computing the LLP Item collection by hand would 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]. To do this the 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. These parsers are also tests 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 be needed to be tested for since the parser generator uses fixed point iterations. To get an idea if the parser generator gets stuck 1000 LLP$(1,1)$ grammars and generated and their parsers are created. These grammars are randomly generated and have three terminals, three nonterminals and six productions. If not all of these parsers are 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 productions sequence computed also needs to match the sequence used to derive the string that is being parsed.

These tests are done for 50 LLP$(1,1)$, 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.

5 Conclusion

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] 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.