University of Copenhagen
Computer Science Department

Parallel Parsing using Futhark
Subtitle

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

Contents

1 Introduction

2 Theory

2.1 LL(k) Parser Generator

For the construction of a LLP(q, k) parser generator the construction of first and follow-set [3, p. 5] and a LL(k) parser generator is needed. A short explanation of the construction of a LL(k) parser generator will be given since in the research of this project $k = 1$ was quite often explained but never $k > 1$ in a manner the author found understandable.

The first and follow-set algorithms described takes heavy inspiration from Mogensens book Introduction to Compiler Design [2, pp. 55–65] and the parser notes [1, pp. 10–15] by Sestoft and Larsen. The modifications are

mainly using the LL(k) extension described in the Wikipedia article in the section "Constructing an LL(k) parsing table"[1] [4].

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

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

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

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

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

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

This may result in an infinite loop if implemented as is so fixed point iteration is used. Let $\mathcal{M} : N \to \mathbb{P}(T^*)$ be a surjective function which is used as a dictionary which maps nonterminals to sets of terminal strings. 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 a fixed k with fixed point iteration the following way.

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

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 $\mathcal{M}_{i+1}(A) = \bigcup_{\delta\,:\,A\to\delta\in P} \mathrm{FIRST}'_k(\delta, \mathcal{M}_i)$ 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 $\mathrm{FIRST}_k(\omega) = \mathrm{FIRST}'_k(\omega, \mathcal{M}_f)$ if k stays fixed.

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

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

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

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

This FOLLOW_k function for a fixed k can then be computed using the following algorithm.

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

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

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

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

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

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

2.2 LLP(q, k) Parser Generator

2.2.1 The idea

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

Definition 2.3 (LL parser configuration)**.** Let $G = (N, T, P, S)$ be a context-free grammar that is an LL(k) grammar for some $k \in \mathbb{Z}_+$. Let each production $p_i \in P$ be assigned a unique integer $i \in \{0, ..., |P| - 1\} = \mathcal{I}$. Then the set of every valid and invalid sequence of productions $\mathcal{S}$[2] is given by $\mathcal{S} = \{(a_k)_{k=0}^n : n \in \mathbb{N}, a_k \in \mathcal{I}\}$. A given configuration [3, p. 5] of a LL(k) parser is then given by.

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

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

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

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

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

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

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

Where $i \in \mathbb{N}$ denotes the index of where the start of the substring y_i such the ordering can be kept. Then we would want to create table lookup function $\Phi : T^* \times T^* \to (T \cup N)^* \times (T \cup N)^* \times \mathcal{S}$. This function maps the pairs (x, y) to a triplet (ω, α, π) which is much the same as the configuration described in definition 2.3. The difference is ω is the push down store before parsing ys first terminal and α is after parsing the first terminal of y. The idea is then

[2] It is chosen to use a squence for the "prefix of a left parse" [3, p. 5] because it did not seem obvious to which set the element is a member of.

you can apply Φ to all the pairs constructed from w. Afterwards these pairs are glued together to determined if the resulting triplet is (S, ϵ, π) meaning the input was parsable. This is described in detail in the LLP paper [3, p. 7], but this description will be helpful for the rest of the paper.

2.2.2 Determining if a grammar is LLP

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

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

Definition 10 [3, p. 13] uses the PSLS(x, y) [3, p. 12] values to determine the initial push down stores which can be used to determine final push down store in the triplet (ω, α, π). The trouble is when working with LLP grammars the PSLS(x, y) definition makes it hard to realize if a grammar is LLP.

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

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

The initial push down store for the admissible pair (b, b) is PSLS$(b, b) = \{b, B\}$. This is because the LL parser configuration can either be (bw, B, π) or (bw, bB, π) where $w \in T^+$. This configuration is right after the first b in the pair is parsed i.e. a string bbw or bb is parsed. The configuration before the first b is parse could be (bbw, bB, π) or (bb, bb, π). Therefore, this grammar is not LLP$(1, 1)$, but it is LLP$(2, 1)$.

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 for any pairs $([^q, [^k)$ can lead to a LL configuration $(]^n, S]^n, \pi)$ where $q + k \leq n$. This grammar is actually a LLP$(1, 1)$ grammar because the LLP parser uses the shortest prefix of the push down store when performing the gluing the LLP

triplets (ω, α, π). The triplet for $([^q, [^k)$ would then be $(S, S], \pi)$ because S is the initial push down store and $S]$ is the final push down store after parsing expanding S and popping [.

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

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

This grammar is mentioned in the LLP paper [3, p. 16] as a grammar that is not LLP(q, k) for any $q, k \in \mathbb{Z}_+$. This is because pair (a^q, a^k) could lead to the possible initial push down stores (a^+, S, π) or (a^+, aS, π). If the productions for the grammar were.

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

Then the grammar is LLP$(1, 1)$ because now only (a^+, S, π) only can occur for the pair (a^q, a^k).

3 Implementation

3.1 Assumptions

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

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

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

In the context u and u' are used they are supposed to be terminal strings. Both of these sets do not result in singletons, so the interpretation cannot be unwrapping them. An example of this is if you compute u with $q = 2$ for the Example 11 grammar [3, p. 14]. It is therefore assumed that for each element in the u and u' sets an item is constructed from them i.e.

step 2. (a): $\{[S' \to \vdash S \dashv \bullet, u, \epsilon, \epsilon], u \in \text{LAST}_q(\vdash S \dashv)\}$
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 has 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.

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

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

3.2 Algorithm 8

3.3 Parser

The parser generator creates a Futhark source file which can be used for parsing. The reason for doing so is Futhark is designed for "parallel efficient computing"[3] that can be executed on a GPU. Therefore the general purpose language, Haskell is used to generate the source files which contains the table parser.

The code the table generator written in Haskell does is mainly actually creating the table. This table is the function **`key_to_config`** that patterns matches on a tuple of two tuples which maps to their respective configuration. These configuration do not actually correspond to an LLP configuration. Instead of using the the LLP configuration (α, ω, π) as is the list homomorphism in algorithm 18 [3, p. 18] which results in the tuples $(RBR(\alpha)LBR(\omega^R), \pi)$ besides for the starting pairs $(\epsilon, \vdash w)$ where $w \in T*$ which becomes $(LBR(\omega^R), \pi)$.

Besides this the Futhark implementation matches algorithm 18. A missing piece is the parallel bracket matching which is not described in the paper. The implementation used takes a lot of inspiration from the implementation described on the Futhark website. The differences are the balancing check is made before the grading.

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

[3] https://futhark-lang.org/

It is also important to note that creating an array of strings in Futhark is also problematic task. Therefore the terminals, nonterminals andp roductions are assigned a index which is used instead.

4 Testing

4.1 First and Follow sets

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

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

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

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

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

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

4.2 LLP(q, k)

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

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

5 Conclusion

References

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

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

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

[4] 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.