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 construnction 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 understable.

The first and follow-set algorithms described takes heavy inspiration from Mogensens book Introduction to Compiler Design [2, p. 55-65] and the parser notes [1, p. 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].

[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].

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.

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

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 $FIRST_k$.

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

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

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

$$FOLLOW_k(A) = \bigcup_{B\,:\,B\to\alpha A\beta\in P} FIRST_k(\beta) \odot_k FOLLOW_k(B)$$

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.

$$FOLLOW_k(A, \mathcal{M}) = \bigcup_{B\,:\,B\to\alpha A\beta\in P} FIRST_k(\beta) \odot_k \mathcal{M}(B)$$

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

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 $\mathcal{M}_{i+1}(A) = \bigcup_{B\,:\,B\to\alpha A\beta\in P} FIRST_k(\beta) \odot_k \mathcal{M}_i(B)$ 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 $\mathcal{M}_u(A) = \{\alpha : \alpha\Box^* \in \mathcal{M}_f(A)\}$ for all $A \in N \backslash \{S'\}$

It then holds that $FOLLOW_k(A) = \mathcal{M}_u(A)$ 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 configuarion [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 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].

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

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

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

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

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 a parsing function $\Phi : T^* \times T^* \to T^* \times (T \cup N)^* \times \mathcal{S}$

2.2.2 Determing if a grammar is LLP

Let $G = (N, T, P, S)$ be a context-free grammar from the $LL(k)$ grammar class. Then for a derivation $S \Rightarrow_{lm}^* \omega x y \beta$ it implies there exists some valid parser configuarion $(y\delta, \alpha, \pi)$

$\{((x, y), (y\delta, \alpha, \pi)) : S \Rightarrow_{lm}^* \omega x y \beta\}$

[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.

3 Implementation

4 Testing

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.