University of Copenhagen
Computer Science Department

Data-Parallel Compilation

Lexical Analysis & Syntax Tree Construction

Author: William Henrich Due (mcj284)
Advisor: Troels Henriksen
Submitted: 5th of April 2024

Abstract

The theory of a lexer generator which creates data-parallel lexers is presented. Such a generator is implemented and used to create a Lisp lexer which performance is compared with other Lisp lexers. Futhermore a description of how to extend an LLP parser generator such that it can construct concrete syntax trees is given. This is also implemented together with the lexer generator as part of Alpacc[1].

Contents

[1] https://github.com/diku-dk/alpacc

1 Introduction

Alpacc is a data-parallel parser generator which uses the LLP grammar class [6]. For this parser generator to actually be useful it would have to incorporate a lexer generator and construct a concrete syntax tree. If lexing or the tree construction was done sequentially then these algorithms would be a bottle-neck. This paper will describe in detail the theory of data-parallel lexing. This theory will be used to implement a data-parallel Lisp lexer which is benchmarked and compared with sequential Lisp lexer implementations.

Also the description of a not work efficient algorithm for constructing a concrete syntax tree is explained in depth. The fact that this algorithm is not work efficient does not make it a reason to disregard it. This is because it only performs more work by a factor of $O(\log n)$.

A problem that will also be solved is the LLP parser only produces a sequence of productions when parsing. We would want the parser to create an actual concrete syntax tree. This is done by extending a given grammar and it will be shown that this does not affect the grammars that can be parsed.

2 Theory

Hills paper "Parallel lexical analysis and parsing on the AMT distributed array processor" [2] describes a method to obtain the path in a deterministic finite automaton (DFA) given a input string in $O(n)$ work and $O(\log n)$ span. It does not describe how to create such a DFA for tokenization but it describes how to simulate it in a work-efficient manner. This section will describe the theory of this method and describe how it can be used for tokenization.

2.1 Data-parallel Lexical Analysis

2.1.1 Data-parallel DFA Traversal

To explain the theory of parallel lexical analysis we first remind the reader of a deterministic finite automaton.

Definition 2.1 (DFA)**.** A deterministic finite automaton [3] [8] is given by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where.

1. Q is the set of states where $|Q| < \infty$.

2. Σ is the set of symbols where $|\Sigma| < \infty$.

3. $\delta : \Sigma \times Q \to Q$ is the transition function.

4. $q_0 \in Q$ is the initial state.

5. $F \subseteq Q$ is the set of accepting states.

This definition is fine as is but we will need to reformualte it to develop data-parallel lexical analysis. We would want the definition to use a curried transition function. But for this to hold then the DFA would also have to be total.

Definition 2.2 (Total DFA)**.** A DFA $(Q, \Sigma, \delta, q_0, F)$ is said to be total if and only if

$$\forall (a, q) \in \Sigma \times Q : \delta(a, q) \in Q$$

If a DFA is total we may use a curried transition function $\delta : \Sigma \to Q \to Q$.

This is needed since otherwise the function would not be defined in the domains Σ and Q.

The reason for allowing δ to be curried is if we have any two functions $g = \delta(a)$ and $f = \delta(a')$ then it follows from composition that.

$$g(f(q)) = (g \circ f)(q)$$

This allows for an alternative way of determining if a string can be produced by an DFA. Instead of first evaluating $f(q)$ and then $g(f(q))$ and then checking if this state is a member of F. We could instead partially apply δ to the symbols and then compose them to a single function which could be used to determine if a string is valid. This sets the stage for data-parallel lexing, we want to find a way to make the problem into a `map-reduce`. We want to do this because it can be computed using a data-parallel reduce unlike the normal way of traversing a DFA.

For the ability to use a data-parallel `map-reduce` we must have a monoidal structure. Here Δ is the set of all the composed partially applied delta functions needs to be closed under function composition.

Remark. A partially applied delta function is shortened to *PADF*. If multiple PADFs are composed then they are a *composed PADF*.

Proposition 2.1 (DFA Composition Closure)**.** Given a total DFA then the set of composed PADFs $\Delta : Q^Q$ will be closed under composition. The

set Δ is the set Δ_i in the recurrence relation with the smallest i such that $\Delta_i = \Delta_{i+1}$.

$$\Delta_1 = \{\delta(a) : a \in \Sigma\}$$
$$\Delta_{i+1} = \Delta_i \cup \{f \circ g : f, g \in \Delta_i\}$$

Proof. We will start by showing that a solution Δ exists. First note that the cardinality is monotonically increasing i.e. $\Delta_i \subseteq \Delta_{i+1}$ since Δ_{i+1} is a union of Δ_i and another set. Secondly note that since $|Q| < \infty$ then a finite amount of functions of the form $Q \to Q$ can exists. Since the set is bounded and increasing then at some point $\Delta_i = \Delta_{i+1}$ and the smallest i where it holds is the solution Δ.

For Δ to be closed under composition, then for arbitray $f, g \in \Delta$ it must hold that $f \circ g \in \Delta$. Since Δ_1 is the set of PADFs that contructs Δ and composition is associative then all elements of Δ can be expressed of the form.

$$\delta(a_1) \circ \cdots \circ \delta(a_n) \in \Delta$$

Given an expression of the aforementioned form we could construct a sequence of PADFs.

$$\delta(a_1), \ldots, \delta(a_n)$$

So we would have a surjective map from sequences to Δ, since some seuqences would map to the same composed PADF. So if all permutations with replacement of Δ_1 of any sequence length are members of Δ then Δ would be closed under composition. Futhermore, since it was shown that Δ is finite then at some point $\Delta_i = \Delta_{i+1}$ then no new composed PADFs be added. Therefore it suffices to show that if all sequences of length k where $1 \leq k \leq i$ is a subset of Δ_i then Δ is closed under composition. This can be shown using a proof by induction.

Base: Δ_1 trivially holds since it only contains sequences of length one and they are the initial PADFs.

Step: Given Δ_i contains every sequence of length i or less then we need to show this implies that Δ_{i+1} will contain every sequence of length $i+1$ or less.

By the induction hypothesis Δ_i must contain every sequence of length i or less and so must Δ_{i+1} due to $\Delta_i \subseteq \Delta_{i+1}$. It remains to show that every sequence of length $i+1$ is a member of Δ_{i+1}. It is known that a direct product of Δ_i is used in the definition of Δ_{i+1} so $\{f \circ g : f, g \in \Delta_i\} \subseteq \Delta_{i+1}$. A direct product between sequences of length 1 and i will create every sequence of length $i+1$ and therefore every sequence of length $i+1$ is a member of Δ_{i+1}. Thereby Δ is closed under composition. □

Now since Δ is closed under composition then it follows that Δ and function composition induces a monoidal structure.

Corollary 2.1 (DFA Composition Monoid)**.** DFA composition closure induces a semigroup which in turn induces the monoid $(\Delta \cup \{id\}, \circ)$ where $id : Q \to Q$ and $id(q) = q$.

Knowing this we can establish the following algorithm

Algorithm 2.1 (Data-parallel String Match)**.** It can be determined in $O(n)$ work and $O(\log n)$ span if a string can be produced by a DFA. First construct the total DFA $(Q, \Sigma, \delta, q_0, F)$ from the DFA.

1. Partially apply δ to every symbol in the input string such that it becomes an array of PADFs.

$$\textbf{map}\ \delta\ [a_1, a_2, \ldots, a_{n-1}, a_n] = [\delta(a_1), \delta(a_2), \ldots, \delta(a_{n-1}), \delta(a_n)]$$

2. Reduce the PADFs into a single composed PADF $\delta' : Q \to Q$.

$$\textbf{reduce}\ (\circ)\ id\ [\delta(a_1), \delta(a_2), \ldots, \delta(a_{n-1}), \delta(a_n)] = \delta'$$

3. Evaluate $\delta'(q_0)$ and determine if $\delta'(q_0) \in F$.

2.1.2 Data-parallel Tokenization

For data-parallel tokenization we need to extent algorithm 2.1. The idea will be to use a data-parallel `map-scan` instead since it will gives all the states. This is also the methods described in Hills [2] paper. The problem is we need to be able to recongnize the longest strech of symbols that results in a token. And we also need to restart the traversal of DFA if a final state is hit with no options to traverse further. To do so we first need to define a function to recongnize tokens.

Definition 2.3 (Token Function)**.** Given a DFA and a set of tokens T. The token function $\mathcal{T} : F \to T$ is a function that maps accepting states to some token.

We will also need a single state to point to which is the dead state. This will become useful when the DFA needs to be traversed multiple times. Since we will need to be able to recongnize when the end of a traversal is reached and we have to restart.

Definition 2.4 (Total DFA with a Dead State)**.** Given a DFA, it is made total with a dead state by defining a new set of states $Q' = Q \cup \{q_\perp\}$ where $q_\perp$ is the dead state. Additionally a new transition function δ' is defined.

$$\delta'(a, q) = \begin{cases} q_\perp & (a, q) \notin \text{dom}(\delta) \\ \delta(a, q) & \text{otherwise} \end{cases}$$

Now that we have a definition of a DFA where the dead state is known another problem is needed to be solved. The problem is as mentioned before that the traversal of the DFA has to be restarted if a dead state is reached after an accepting state. This is done using the following binary operation.

Definition 2.5 (Make PADF Safe Under Composition)**.** Given a DFA with two PADFs $f = \delta(a)$ and $g = \delta(b)$. The operation $\oplus$ makes f safe to compose such that it will continue traversing the DFA.

$$(f \oplus g)(x) = \begin{cases} q_0 & f(g(x)) = q_\perp \wedge g(x) \in F \\ g(x) & \text{otherwise} \end{cases}$$

This definition will make every possible final state become the initial state. This forgets the final state but it allows for the traversal to continue. The forgetfulness is not a problem this can be solved by looking at the previous PADF. This puts a limit on the lexer which is it only allows for going back to the previous state if a dead state is hit. Using this and the previous definitions we get the following algorithm.

Algorithm 2.2 (Data-parallel Tokenization)**.** Given a total DFA with a dead state where $q_0 \notin F$ and a token function $\mathcal{T} : F \to T$. A string can be tokenized in $O(n)$ work and $O(\log n)$ span.

1. Let $s = [a_1, a_2, \ldots, a_{n-1}, a_n]$ be a string that will be tokenized then partially apply δ to every symbol.

$$\textbf{map}\ (\delta)\ s = x$$

2. Make every PADF safe for composition beside for the last PADF.

$$\textbf{map}\ (\oplus)\ (\textbf{init}\ x)\ (\textbf{tail}\ x) +\!\!+\ [\delta_n] = [\delta_1 \oplus \delta_2, \ldots, \delta_{n-1} \oplus \delta_n, \delta_n] = y$$

3. Do a scan to get every composition.

$$\textbf{scan}\ (\circ)\ id\ y = z$$

4. Compute the actual state and determine weather it is an end state[2].

$$\begin{aligned}&\textbf{let } f = \lambda i \to \\ &\quad \textbf{let } s = \textbf{if } i = 0 \textbf{ then } x[i](q_0) \textbf{ else } (z[i-1] \circ x[i])(q_0) \\ &\quad \textbf{in } (i = n-1 \vee (z[i](q_0) = q_0 \wedge s \in F), s) \\ &\textbf{in map } f \ (\textbf{iota } n)\end{aligned}$$

5. Remove every state that is not an ending state from the result in Step 4.

$$\textbf{filter } (\lambda(b, s) \to b)$$

6. Assert that the last state is an accepting state of the result in step 5.

$$(\lambda(b, s) \to s \in F) \circ \textbf{last}$$

7. Produce the token sequence by applying the following function to the result from step 5.

$$\textbf{map } (\lambda(b, s) \to \mathcal{T}(s))$$

We will give an explanation of why the algorithm produces an array of paths. Here every path has the initial state q_0 and q_0 is not included in the resulting path.

We know in step 1-2. every PADF is created and is made safe so we know at this point either it maps to some orignal states or some maps to q_0 because the compositions are safe. At step 3. a prefix sum using function composition is computed of these composed safe PADFs. Which by Algorithm 2.1 can give us all states traversed in the DFA. But due to $\oplus$ is used then an accepting state followed by an dead states will become the initial state q_0. This almost gives all the paths but all final states becomes initial states.

Then at step 4. the paths taken in the DFA from state q_0 is found. We know for a given state s in a path it can be one of these cases.

- If $i = 0$ then $x[i](q_0)$ will be the first state visited after q_0 by definition of the transition function.
- If $i \neq 0$ then s will be $(z[i-1] \circ x[i])(q_0)$. $z[i-1](q_0)$ could possibly not map to its correct state and instead map to q_0.

[2]If you were to also keep track of the indices then the span of each token could also be found.

– If $z[i-1](q_0)$ maps to q_0 then the traversal of the DFA may have been reset and $(z[i-1] \circ x[i])(q_0)$ will map to the correct state of a possibly new path.
– If $z[i-1](q_0)$ maps to the correct state then $(z[i-1] \circ x[i])(q_0)$ will also map to the correct state.

It can be determined where each path ends by the following predicate.

$$i = n - 1 \vee (z[i](q_0) = q_0 \wedge s \in F)$$

We know if $i = n - 1$ then it must be the last state of the last path. The second case is if $z[i](q_0) = q_0$ and $s \in F$ holds. We know if $z[i](q_0) = q_0$ holds then it is a possible final state that was mapped to q_0. And if $s \in F$ then since $q_0 \notin F$ we know $z[i](q_0) \neq s$ meaning the reason $z[i](q_0) = q_0$ must be because of a reset by composition being safe. Therefore it must be a final state in a path that is not the last path.

Step 5-7. will first keep every last state by this predicate. Afterwards it is asserted that the last state is $s \in F$. This only done for the last state of the last path since all previous paths would had been valid else they would had not been reset. Finally just map every state to the token it has.

Example 2.1 (Problems with this method)**.** If we had the following three patterns.

$$\tau_1 = a(a \qquad \tau_2 = a+ \qquad \tau_3 = a)a$$

Then the algorithm would not be ble to determine the tokens for the following input.

$$a(aaa)a$$

The token sequence of this input could be determined to be τ_1, τ_2, τ_3 with spans $(0, 3), (3, 4), (4, 7)$ if you reset at the last acceptings state. But since the algorithm resets at the last state from an accepting state then it fails. Meaning it would be able to recongnize τ_1. The problem is at the token τ_2 since it would get the span $(3, 5)$ and then the algorithm would not recongnize any token with the pattern $)a$.

There exists an alternative to this method given in Robin Voetters master thesis [7, pp. 13–16]. This is done by constructing a NFA for each token. Then create a new NFA where there is an epsilon transition from the initial state to each tokens NFAs inital state. Then you would convert this NFA into a DFA where you would have a function $\mathcal{T}$ which maps final states to

a token. Then for every outgoing transition $\delta(q_0, t) = q$ of the DFA then we add a transition from a final state $q_s \in F$ such that if $(q_s, t) \notin \text{dom}(\delta)$ then $\delta(q_s, t) = q$. If any of these new transitions are visited during the traversal of the DFA then a token is produced. We also have to account for the first transition visited which also will be a producing a token.

2.2 Concrete Syntax Trees

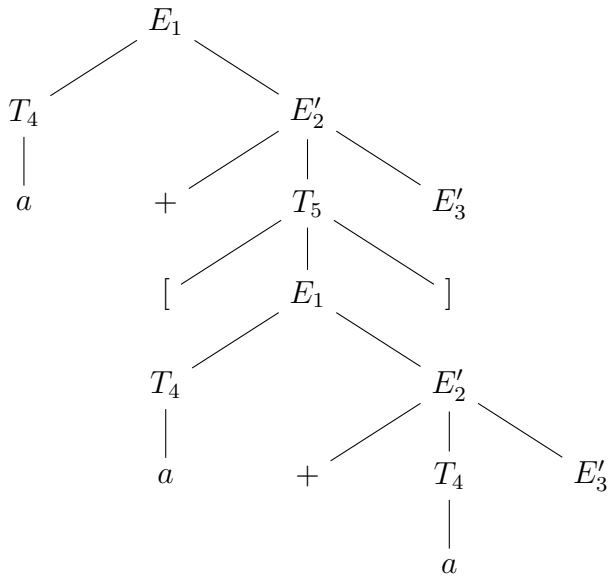
The Alpacc parser generator is a parser generator which creates parser for the LLP grammer class. The LLP parsers described in the LLP paper [6] only touches upon the verification and production sequence construction of parsing, not the creation of concrete syntax trees. This will also had to be done in a data-parallel manner else we would run into a bottle-neck. A method for this is described in Voetters master thesis [7, pp. 32–33] about compilation on the GPU. Here it is described how to create a concrete syntax tree using the representation from Hsu PhD dissertation [4, pp. 77–81] where you have a parent vector that can be used for multiple transformations of the syntax tree. The representation of the tree will be an array of nodes where each node has the index to its parent node. The nodes will also be given in a preorder traversal of the tree.

2.2.1 Concrete Syntax Tree Construction

So as an example we wish to construct syntax tree for the grammar.

$$1)\ E \to TE' \quad 2)\ E' \to +TE' \quad 3)\ E' \to \varepsilon \quad 4)\ T \to a \quad 5)\ T \to [E]$$

Then for the input $a + [a + a]$ we would want to construct the following tree.



The first problem to tackle is terminals are not included in the left parse returned by the parser, currently it only returns the productions.

$$[E_1, T_4, E'_2, T_5, E_1, T_4, E'_2, T_4, E'_3, E'_3]$$

To create the concrete syntax trees the first problem is how to weave together the production sequence with the sequence of terminals. This is done by extending the grammar such that each terminal is associated with a unique production. That way a production in the production sequence will have a direct correspondent to a terminal. This can then be used when constructing the syntax tree.

$$[E_1, T_4, A_a, E'_2, A_+, T_5, A_[, E_1, T_4, A_a, E'_2, A_+, T_4, A_a, E'_3, A_], E'_3]$$

Here A_t is a production that produces the terminal it represents. Then we can just map everyone of these productions to its terminal like so.

$$[E_1, T_4, a, E'_2, +, T_5, [, E_1, T_4, a, E'_2, +, T_4, a, E'_3,], E'_3]$$

A concern with this method is if this changes the number of grammars that can be parsed by LLP or if it changes the size of the table used for LLP parsing. These problems does not exists for this extension and it is later shown these problems does not exists.

The parent vector can then be constructed with the following algorithm.

Algorithm 2.3 (Parent Vector)**.** Given a left parse a_0 then we can construct the the parent vector as followed.

1. For each production map it to its arity i.e. the number of nonterminals in the right-hand side of the production.

$$a_1 = \textbf{map}\ arity\ a_0$$

   Here *arity* is a function that maps productions to its arity.

2. Substract one from each arity.

$$a_2 = \textbf{map}\ (-1)\ a_1$$

3. Do an exclusive scan using addition on the arities.

$$a_3 = \textbf{scan}\ (+)\ a_2$$

4. Find the index of the previous smaller or equal element for each element from the previous step.

$$a_4 = \textbf{map}\ prev\ a_3$$

   Here *prev* is a function that finds the index of the previous smaller or equal element in a_3.

5. Set the first elements parent to be it self. $a_5 = a_4[0] = 0$

Using this algorithm we can show the computation of parent vector for the example.

Arity	Stack Change	Stack Depth	Parent Index	Node	Index
2	1	0	0	E_1	0
1	0	1	0	T_4	1
0	-1	1	1	a	2
3	2	0	0	E'_2	3
0	-1	2	3	$+$	4
3	2	1	3	T_5	5
0	-1	3	5	$[$	6
2	1	2	5	E_1	7
1	0	3	7	T_4	8
0	-1	3	8	a	9
3	2	2	7	E'_2	10
0	-1	4	10	$+$	11
1	0	3	10	T_4	12
0	-1	3	12	a	13
0	-1	2	10	E'_3	14
0	-1	1	5	$]$	15
0	-1	0	3	E'_3	16

Table 1: Example of how the parent vector is constructed.

Note how the arity for the original productions ends up being the number of symbols in the right-hand side while each terminal symbol ends up being zero. This is due to the extension done to the original grammar.

Step 4. in algorithm 2.3 can be solved in different ways. A possible way to do it in a data-parallel manner is doing a linear backwards search. This would lead to $O(n^2)$ work and $O(n)$ span. This would be a pretty huge bottle neck-compared to the $O(n)$ work and $O(\log n)$ span for both tokenization and parsing. So a method for finding the previous smaller or equal element is used. This can be done in $O(n \log n)$ work and $O(\log n)$ span as described in Voetters master thesis [7, pp. 32–33].

Algorithm 2.4 (Previous Smaller or Equal Element)**.** Given an array of a_0 where $|a_0| = 2^n$ and the array contains some set of elements A where $(\leq, A)$ is a total ordering.

1. Construct tree of minima.

$$
\begin{aligned}
&\textbf{let } create\ arr = \\
&\quad \textbf{if } |arr| = 1 \\
&\quad \textbf{then } arr \\
&\quad \textbf{else let } arr' = \textbf{map } (\lambda i \to \min\ arr[2i]\ arr[2i+1])\ (\textbf{iota } |arr|) \\
&\qquad \textbf{in } create\ arr' +\!\!+ arr
\end{aligned}
$$

   This will compute a binary tree where index 0 of the array is the root. Furthermore for a node at index i you can find its parent using $\lfloor \frac{i-1}{2} \rfloor$, its left child $2i+1$ and its right child $2i+2$ assuming they exists.

$$\textbf{let } tree = create\ a_0$$

2. To find the previous smaller or equal element of a value v with index i. We will first find the common ancestor of the elements from i assuming some previous smaller or equal element exists.

$$
\begin{aligned}
&\textbf{let } ascent\ j = \\
&\quad \textbf{if } j \neq 0 \wedge (left\ j \vee tree[sibling\ j] > v) \\
&\quad \textbf{else } ascent\ (parent\ j) \\
&\quad \textbf{then } j
\end{aligned}
$$

   Here $left$ is a predicate that determines if j is the index of a left node i.e. if it is odd. While *sibling* determines the neighbouring node. Then compute the index $k = ascent\ (2^n + i)$, if $k = 0$ then the previous smaller or equal element does not exists.

3. Now descent to the right most element that is smaller or equal to v and adjust for the offset.

$$
\begin{aligned}
&\textbf{let } descent\ j = \\
&\quad \textbf{if } j < 2^n \\
&\quad \textbf{else if } tree[2j+2] \leq v \textbf{ then } descent\ (2j+2) \textbf{ else } descent\ (2j+1) \\
&\quad \textbf{then } j - 2^n
\end{aligned}
$$

Note that the use of recursion can be easily rewritten to loops.

2.2.2 Grammar Extension

We will first need a precise definition of the grammar extension. This is needed to show it does not lead to fewer grammars being LLP or larger LLP tables.

Given a grammar (N, T, P, S) where N is the set of nonterminalsm, T is the set of terminals, P is the set of productions, and S is the initial nonterminal. We first construct the following set of nonterminals.

$$N_T = \{A_t : t \in T\}, \text{ where } A_t \notin N \text{ for any } t \in T$$

And we also create an secondary set of productions.

$$P_T = \{A_t \to t : t \in T\}, \text{ where } A_t \to t \notin P \text{ for any } t \in T$$

Now we create a new set of productions by replacing every terminal in the right-hand side of a production by its corresponding nonterminal.

$$P_r = \{A \to \delta[A_{t_1}/t_1, \dots A_{t_n}/t_1] : A \to \delta \in P\}$$

The final grammar that will be used for parsing is now.

$$G' = (N', T', P', S') = (N \cup N_T, T, P_r \cup P_T, S)$$

First we will need to show that any LL grammar extended can still be parsed using LL parsing. The definition of a LL(k)[3] [1] parsing table is as followed.

Definition 2.6 (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.

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

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

Proof. Given a LL(k) grammar G, then for a production $A \to \delta \in P$ and its corresponding extended production $A \to \delta[A_{t_1}/t_1, \dots A_{t_n}/t_1] = A' \to \delta' \in P_r$ we have.

$$\text{FIRST}_k^G(\delta) \overset{k}{\bullet} \text{FOLLOW}_k^G(A) = \text{FIRST}_k^{G'}(\delta') \overset{k}{\bullet} \text{FOLLOW}_k^{G'}(A)$$

[3] I do not know where else to cite from. The definition is found in my Bachelor thesis and is partly based on other definitions.

The first and follow-sets must be unchanged for productions in P_r because all added nonterminals directly becomes their corresponding terminals in a derivation. Therefore none of these productions leads to a table conflict. Then for a production $A_t \to t \in P_T$ we have.

$$i \in \tau(A_t, s) \text{ where } s \in \text{FIRST}_k^{G'}(t) \overset{k}{\bullet} \text{FOLLOW}_k^{G'}(A_t)$$

Since A_t only has a single production rule then it will always hold that $|\tau(A_t, s)| \leq 1$. Meaning the extended grammar G' will also be a LL(k) grammar. □

Something to note is this extention does lead to larger LL tables but this is not the case for LLP. So now we will show this extension does not lead to larger LLP tables and does not change if a LLP grammar can be parsed. To do so we need to be reminded of the PSLS[4] [6] definition.

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

$$\begin{aligned}\text{PSLS}(x,y) = \{&\alpha : \exists S \Rightarrow_{lm}^* wuA\beta \Rightarrow wxB\gamma \Rightarrow^* wxy\delta,\\ &w, u, y' \in T^*, a \in T, A, B \in N, \alpha, \beta, \gamma, \delta \in (N \cup T)^*, u \neq x,\\ &y = ay', \alpha \text{ is the shortest prefix of } B\gamma\\ &\text{such that } (y, \alpha, ()) \vdash^* (y', \omega, \pi)\}\\ \cup \{&a : \exists S \Rightarrow^* wuA\beta \Rightarrow wxa\gamma \Rightarrow^* wxy\delta,\\ &a = \text{FIRST}_1^G(y), w, u \in T^*, \beta, \gamma, \delta \in (N \cup T)^*, u \neq x\}\end{aligned}$$

We will also need the following Lemma.

Lemma 2.1 (Grammar Extension Bijection)**.** Let G be an LL grammar and let G' be the extended grammar of G. If we have the following two leftmost derivations from each grammar $S \Rightarrow_{lm}^* wA\delta \Rightarrow w\gamma$ and $S' \Rightarrow_{lm}^* wA'\delta \Rightarrow w\gamma'$ then the following derivations hold.

- $S \Rightarrow_{lm}^* w\gamma'[A_{t_1}/t_1, \ldots A_{t_n}/t_n]$
- $S' \Rightarrow_{lm}^* w\gamma[t_1/A_{t_1}, \ldots t_n/A_{t_n}]$

Hence there exists a bijection between these two derivations.

[4]This is a slightly modified definition by Vladislav Vagner because the old definition in the paper does not work.

Proof. We know that G and G' produces the same language and due to them both being LL then there is only one way of deriving their leftmost derivations. It is also known that since A is a nonterminal in G then it must not be a A_t nonterminal in G' else they would not share a common terminal string w. So the derivation of A' must use a production of the form $A' \to \alpha[A_{t_1}/t_1, \ldots A_{t_n}/t_1] \in P_r$ where $A \to \alpha \in P$. Then we know that when the single derivation on A' is performed then not a single A_t nonterminal in γ' has been derived to its terminal t. Therefore it must be the case that $\gamma = \gamma'[A_{t_1}/t_1, \ldots A_{t_n}/t_n]$. Likewise A' has just been derived so no A_t nonterminals have been derived meaning $\gamma' = \gamma[t_1/A_{t_1}, \ldots t_n/A_{t_n}]$. □

Now we will be able to show what we wanted to show.

Proof. The proof will be done by showing that given a LLP grammar G i.e. for any admissible pair (x, y) we have $|\text{PSLS}(x, y)| = 1$. Then it will follow that for its corresponding extended grammar G' we must also have $|\text{PSLS}(x, y)| = 1$.

- If we are given an pair (x, y) where $|\text{PSLS}(x, y)| = 1$ for some LLP grammar G and we have some derivation.

  $$S \Rightarrow^*_{lm} wuA\beta \Rightarrow wxB\gamma \Rightarrow^* wxy\delta$$

  Where there only exists a single shortests prefix α of $B\gamma$ such that $(y, \alpha, ()) \vdash^* (y', \omega, \pi)$ and for any a that may be in the second set we have $a = \alpha$.

  Given the extended grammar G', then for the pair (x, y) to be in the first set of the PSLS definition then it must be of the form.

  $$S' \Rightarrow^*_{lm} wu'A_tB'\gamma' \Rightarrow wxB'\gamma' \Rightarrow^* wxy\delta'$$

  Both of the leftmost derivations will have the common subderivation wx since they still produce the same language and they both must produce x.

  By Lemma 2.1 we have a bijection between derivations of $wxB'\gamma'$ and $wxB\gamma$ then there must also exist a single shortest prefix α' of some $B'\gamma'$ such that $(y, \alpha', ()) \vdash^* (y', \omega', \pi')$. Since if there were multiple different shortest prefixes of some $B'\gamma'$ then so would $B\gamma$.

- If we are given an pair (x, y) where $|\text{PSLS}(x, y)| = 1$ for some LLP grammar G and we have some derivation.

  $$S \Rightarrow^* wuA\beta \Rightarrow wxa\gamma \Rightarrow^* wxy\delta$$

Where there only exists a single $a = \text{FIRST}_1^G(y)$ where for any α that may be in the first set we have $a = \alpha$.

Then for the extended grammar G' then a derivation of the following form must also exist.

$$S' \Rightarrow_{lm}^* wu'A_tB'_a\gamma' \Rightarrow wxB'_a\gamma' \Rightarrow^* wxy\delta'$$

By Lemma 2.1 we have a bijection between derivations of $wxB'_a\gamma'$ and $wxa\gamma$. Knowing this and $B'_a \Rightarrow a$ then there only exist a single $a = \text{FIRST}_1^{G'}(B'_a)$ for this form of derivation.

We would now need to show that B'_a would exist in the first set of the PSLS definition since it cannot exist in the second set. We see the derivation is of the same form as a derivation as in the first set. We also see the shortest prefix α' of $B'_a\gamma'$ is such that $(y, \alpha', ()) \vdash^* (y', \varepsilon, \pi')$ is B'_a. By $a = \text{FIRST}_1^{G'}(B'_a) = \alpha'$ then α' is the only such prefix.

Thereby the extended grammar is also LLP and since the bijection grammar and its extended version then we know the PSLS table does not get larger. □

3 Implementation

3.1 Tokenization

To implement tokenization we run into the problem that we want to represent an partially aplied delta function in Futhark. Futhark does not allow for represting functions as an array element. Even if it was possible then it might not be the most efficient representation. Instead if everyone of of states in the DFA is assigned a unique number then we could represent it as an array. So `delta[i] = j` would mean the partially applied function `delta` maps `i` to `j`. We could then also compose this function like so.

```
let compose a b = map(\i -> a[i]) b
```

This was at some time the implementation of Alpacc but it is possible to go faster. You can enumerate every composed partially applied delta function and create a table of compositions. Then your function composition would become.

```
let compose a b = composition_table[a, b]
```

The way you would then enumerate every composed partially applied delta function is by first constructing the set of such functions and then assign each some number. A way to construct this set is by Proposition 2.1. You

would just have to keep track of the composed PADFs used to construct each composed PADF to make a table. Something one might notice is it is actually computationally wasteful since transitions and subpaths that are not neighbouring will result in a valid path in the DFA. So as an alternative one could only compose the composed PADF that are neighbouring eachother. Then when you compose two transitions then you would need to add them as new transitions. Then forall the composed PADFs found you would have to define a dead composed PADF. It can then be used when some composition of existing composed PADFs are not defined in the table.

So if you had the following pattern you wanted to match abc then a is a neighbor to b and b is a neighbor to c. So initially we have a set of PADFs.

$$\{\delta(a), \delta(b), \delta(c)\}$$

Then we compose both $\delta(a)$ with $\delta(b)$ and $\delta(b)$ with $\delta(c)$ since both pairs are neighbouring transitions. Then the set of the new composition together with the old compositions results in the set.

$$\{\delta(a), \delta(b), \delta(c)\} \cup \{\delta(a) \circ \delta(b), \delta(b) \circ \delta(c)\}$$

Now $\delta(a) \circ \delta(b)$ is composed with $\delta(c)$ and $\delta(a)$ is composed with $\delta(b) \circ \delta(c)$. Here both $\delta(a) \circ \delta(b)$ and $\delta(b) \circ \delta(c)$ are recomputed. The set of these compositions are then unioned with the old set.

$$\{\delta(a), \delta(b), \delta(c), \delta(a) \circ \delta(b), \delta(b) \circ \delta(c)\} \cup \{\delta(a) \circ \delta(b) \circ \delta(c)\}$$

This ends up being the final set. This can also be done on the set of compositions that were made safe. Currently this implementation is extremely slow, it does some recomputations which could be mitigated using memoization or removing connections that have been computed before.

4 Tests

4.1 Tokenization

4.1.1 General Testing

To test the tokenization a data-parallel lexer is compared with a sequential version. The sequential version works much like the data-parallel version. It will traverse the DFA until it reaches a dead state afterwards it will determine if the previous state was accepting, if so then reset the traversal else the input could not be tokenized.

The testing strategy is to choose some set of patterns and strings of interest. Then compare if the sequence of tokens with their spans are the same. One of the set of patterns that was deemed interesting are the one given in Example 2.1.

$$\tau_1 = a(a \qquad \tau_2 = a+ \qquad \tau_3 = a)a \qquad \tau_4 = \backslash s|\backslash n|\backslash t|\backslash r$$

Here something like $a(aa)a$ can get recongnized while $a(aaa)a$ can not get recongnized. This test is to make sure concat, alternation, and the one-or-more pattern works as expected. This is tested multiple times using other test cases also. It also assures that $a(aaa)a$ will fail. Another pattern of interest is.

$$\tau_1 = \textbf{if} \qquad \tau_2 = \textbf{then} \qquad \tau_3 = \textbf{else} \qquad \tau_4 = [a\text{-}z] + \qquad \tau_5 = \backslash s|\backslash n|\backslash t|\backslash r$$

Here we want the key words **if**, **then**, and **else** to be associated with a token before it is associated with τ_4. Here the order the tokens are defined are the order they are prioritized from most important to least. There were also some other interesting patterns and inputs but these examples gives a good idea of cases tested for. [5]

A problems with this method of testing is it is limited to single examples and what properties the author finds to be interesting. There might be edge cases that were missed and not accounted for since they did not seem to be of interest.

4.1.2 Large Lisp Input

Since the general testing is only on small inputs then it would also be good to test on large randomized inputs. This would give greater confidence in the lexer generator working as expected. This is done by a data-parallel Lisp lexer being compared with another Lisp lexer made with the Logos lexer library. Here a large randomized input that could be tokenized by a Lisp lexer is created randomly and the input is about $100MiB$.

4.2 Parent Vector

To assert the correctness of the parent vector construction the algorithm for finding the previous equal or smaller element is tested. The reason for only testing this is the steps 1-3. in algorithm 2.3 do not offer much complexity. A problem with this testing method is it assumes that some parts of the algorithm is correct. It also assumes as a whole the algorithm is correct.

[5] The different cases tested for can be found here.

Also something to consider is why not testing that the concrete syntax trees correspond to a sequential version. This is because the parent vector is the last step in testing since the production sequence is already tested in Alpacc [1].

The strategy for testing is using property based testing. The property tested for is given an array a of 32-bit integers then f maps algorithm 2.4 over a. This results in $f(a)$ should be the same as $g(a)$. Here g is an implementation that for every element in the array does a backwards linear search to find the previous smaller or equal element. Then the following property should be fulfilled.

$$\forall a. f(a) = g(a)$$

Since not all a can be tested then a is randomized. Also note if the previous or smaller element does not exists then f and g will map to -1.

5 Benchmarks

5.1 Lisp lexer

To assert the performance of a data-parallel lexer some benchmarks of Lisp lexers are compared. These Lisp lexers recongnize the following tokens patterns.

$$\tau_1 = (\qquad \tau_2 =) \qquad \tau_3 = [a\text{-}zA\text{-}Z0\text{-}9]+ \qquad \tau_4 = (\backslash s|\backslash n|\backslash t|\backslash r)+$$

Here τ_1 and τ_2 are the open and closing parenthesis in S-expressions. τ_3 are the atoms while τ_4 is the spacing between the important tokens for the grammar τ_1, τ_2, and τ_3.

The benchmarks are of three different lexers and are named by the languages they are written in.

- Rust: A lexer made with the Logos library and every token is kept in memory.
- C: A lexer written by Troels Henriksen. Not all tokens are kept in memory.
- Futhark: A data-parallel lexer from Alpacc [1] using the method described and keeps every token in memory.

The benchmarks are measured using a Ryzen 5 1600x CPU and a Nvidia RTX 3060 GPU. Which may not seem like a fair comparison since the CPU was released in 2017 and the GPU was released in 2021. This was the chosen method of benchmarking since I was not able to get consistent measurements on the compute cluster Hendrix. Also single threaded performance should have stalled the past years so it did not seem like a large concern.

Input Size (MiB)	Rust (μs)	C (μs)	Futhark (μs)
10	44,866	24,440	3,388
20	86,898	47,876	6,960
30	131,183	70,986	10,667
40	180,367	98,906	14,400
50	214,258	118,350	17,744
60	274,776	148,982	23,108
70	330,414	175,402	26,358
80	370,787	200,867	30,512
90	430,997	234,474	34,591
100	469,772	248,722	39,199

Table 2: Lisp Lexer Benchmarks. The Futhark benchmarks uses the CUDA backend.

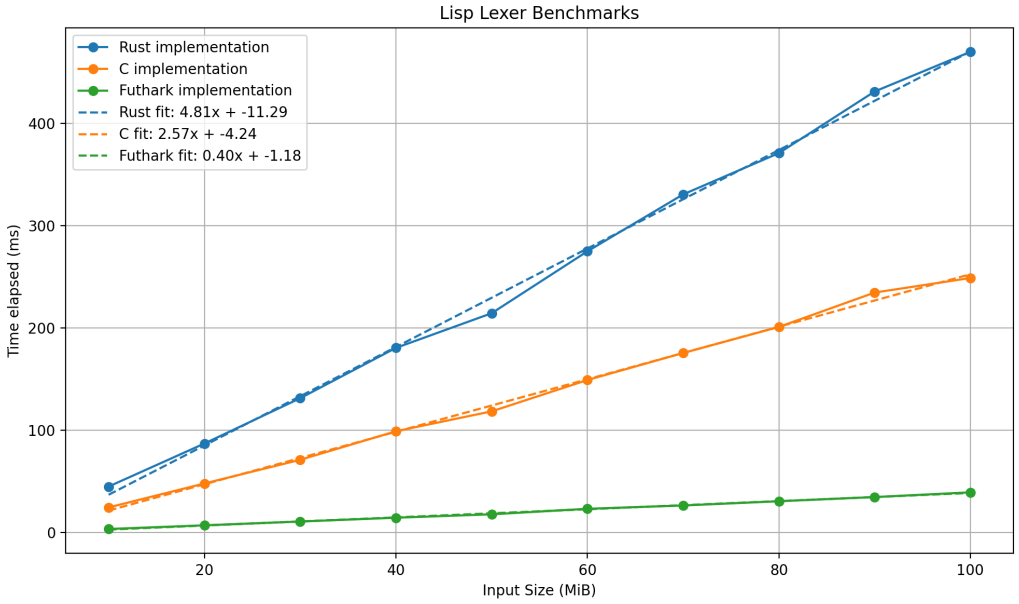


Figure 1: Lisp Lexer Benchmarks

As we see from the plot the time elapsed grows roughly linearly with the input size for all the benchmarks. So we can use the linear equations found to solve for each lexers throughput in GiB/s.

	Rust (μs)	C (μs)	Futhark (μs)
Througput (GiB/s)	0.2	0.4	2.5

Table 3: Lisp Lexers Throughput

As we see a data-parallel Lisp lexing is about 5 times faster than the C Lisp lexer and 12 times faster than the rust lexer.

6 Discussion

A problem with this method is it has to do more passes when lexing unlike a sequential implementation. Meaning this method would likely not be great for smaller inputs. But For large datasets this method does seem like a possible usecase, such as tokenization of English. It could also be useful for formal language processing for large datasets such as JSON and CSV. CSV is a very simple formal language and the correctness of a CSV file could possibly be asserted by counting the number of tokens in each line. Maybe also some type checking for each column in the CSV file.

In the case of JSON we would have to utilize the LLP parser in Alpacc [1] and the concrete tree constructions presented. The LLP parsing method should not be more expensive than the lexing since the tokens in JSON are normally quite long. The lexing mainly relies on a fast radix sort like CUBs which can sort 15GB/s 32-bit keys on a Titan X [5]. We would also have to keep in mind that a Titan X (Pascal) is more powerful than a RTX 3060 so the comparison is unfair[6].

The concrete syntax tree construction should neither be a problem. This is again because it works on a smaller input because only the tokens and productions are considered. A problem is it does seem like if parsing can be done with a throughput 15GB/s then lexing is quite the bottle-neck. Maybe the lexing could be done faster by hand optimized CUDA code. But the lexer is a very simple implementation and Futhark has a mature compiler that will optimize the CUDA code.

[6] The performance difference does not seem that great.

7 Conclusion

The implementation of the lexer generator and the concrete syntax tree construction is believed to work. A fast data-parallel lexer implementation is also created but it does not seem fast enough. A data-parallel Lisp lexer on an RTX 3060 is only 12 times faster than Logos Lisp lexer. This seems unexpected since the Lisp lexer only uses a few data-parallel scans and maps. It could be the case that the scan using table lookups for function compositions is the problem.

It is believed the concrete syntax tree construction and parsing of Alpacc should be fast enough. These parts are actually not benchmarked but they will most likely work on inputs smaller than the lexer. Also for the case of parsing it is known that it is almost just a radix sort so its throughput should be fast. And for the case of concrete syntax tree construction it will roughly be three maps so it is not concerning.

Overall the project did result in a working and faster than a sequential implementation of a lexer. The project also contributed with theoretical explanations of data-parallel lexing and concrete syntax tree construction. In regards to the tree it was shown that the LLP parser can be extended to create concrete syntax trees while still being able to parse every LLP grammar.

References

[1] William Henrich Due. *Parallel Parsing using Futhark*. 2023. URL: https://futhark-lang.org/publications.html.

[2] Jonathan M.D Hill. "Parallel lexical analysis and parsing on the AMT distributed array processor". In: *Parallel Computing* 18.6 (1992), pp. 699–714. ISSN: 0167-8191. DOI: https://doi.org/10.1016/0167-8191(92)90008-U. URL: https://www.sciencedirect.com/science/article/pii/016781919290008U.

[3] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321455363.

[4] Aaron Wen-yao Hsu. "A Data Parallel Compiler Hosted on the GPU". PhD thesis. Indiana University, 2019.

[5] Elias Stehle and Hans-Arno Jacobsen. "A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs". In: *CoRR* abs/1611.01137 (2016). arXiv: 1611.01137. URL: http://arxiv.org/abs/1611.01137.

[6] 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.

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

[8] Wikipedia contributors. *Deterministic finite automaton — Wikipedia, The Free Encyclopedia*. [Online; accessed 4-February-2024]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Deterministic_finite_automaton&oldid=1192025610.