University of Copenhagen
Computer Science Department

Data-Parallel Compilation
Lexical analysis & Syntax Tree Construction

William Henrich Due (mcj284)
Submitted: 5th of April 2024

Abstract

Abstract.

1 Introduction

Introduction.

2 Theory

Hills paper “Parallel lexical analysis and parsing on the AMT distributed array processor” [1] describes a method to obtain the path in a deterministic finite automata given a input string. This section will describe the theory of this method and extend the it for tokenization.

2.1 Data-parallel Lexical Analysis

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

Definition 2.1 (DFA)**.** A deterministic finite automata [2] [6] 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

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

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

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

The reason for doing so is because 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 $f(q)$, 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 implementation 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 δ functions needs to be closed under function composition.

Proposition 2.1 (DFA Composition Closure)**.** Given a total DFA then the set of endofunctions $\Delta : Q^Q$ 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 circt 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 endofunctions that contructs Δ and composition is associative then all elements of Δ can be expressed of the form.

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

If all permutations with replacement of Δ_1 of any sequence length are members of Δ then Δ would be closed under composition. Futhermore, it is known that Δ is finite so the sequences at some point $\Delta_i = \Delta_{i+1}$ would only add new sequences but no new endofunctions. 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 endofunctions.

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

By the induction hypothesis Δ_{i+1} must contain every sequence of length i or less 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. $\square$

Now since Δ is closed under an arbitrary binary associative operations 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 a sequence of endofunctions.

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

2. Reduce the endofunction into a single endofunction $\delta' : Q \to Q$.

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

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

2.2 Data-parallel Tokenization

For data-parallel tokenization we need to extent data-parallel algorithm 2.1 will be needed to be extended. 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 [1] 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 while no options to traverse further. To do so we first need af function 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 $Q' = Q \cup \{d\}$ where d is the dead state. Additionally a new transition function δ' is defined.

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

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

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

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 endofunction. 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 previous definitions can be put together to 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 $s = [a_1, a_2, \dots, a_{n-1}, a_n]$ be a string that will be tokenized then partially apply δ to every symbol.

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

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

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

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[1].

```
$\textbf{let}\ f = \lambda i \to$
    $\textbf{let}\ s = \textbf{if}\ i = 0\ \textbf{then}\ x[i](q_0)\ \textbf{else}\ (z[i-1] \circ x[i])(q_0)$
    $\textbf{in}\ (i = n - 1 \lor (z[i](q_0) = q_0 \land s \in F), s)$
$\textbf{in map}\ f\ (\textbf{iota}\ n)$
```

5. Remove every state that is not an ending state.

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

6. Assert that the ending state is an accepting state.

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

7. Produce the token sequence.

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

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

Proof. We wish to show that the algorithm produces an array of paths where each path starts in q_0 and is not included in the path. Knowing where these paths ends it can then be trivially used to construct the token sequence using the token function $\mathcal{T}$. We will also show that it can be correctly determine wether the DFA paths taken are correct.

We know in step 1-2. every endofunction is created and we know at this point either it maps to some orignal states or some maps to q_0 because of safe composition. At step 3. a prefix sum using function composition is computed of these endofunctions. Which by Algorithm 2.1 can give us all states traversed in the DFA. But due to safe composition $\oplus$ is used then an accepting state followed by an dead states will become the initial state q_0. This almost gives all the paths, since now when a final state in a path is reach it becomes the initial state and the traversal i restarted.

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 will 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 wrongly map to q_0 by definition of safe composition.
  - If $z[i-1](q_0)$ maps wrongly to q_0 then the traversal of the DFA will be reset and $(z[i-1] \circ x[i])(q_0)$ will map to the correct state of a 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 possible a final state was mapped to q_0 by safe composition. 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 safe composition meaning then 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 $s \in F$. Only the last state of the last path is needed since all previous path would had been valid else it would had not been reset by safe composition. Finally just map every state to the token it has. $\square$

Example 2.1 (Problem 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 would get the span $(3,5)$ and then the algorithm would not recongnize any token with the pattern $)a$.

2.3 Grammar Extension for Syntax Trees

The way we will extended the grammar is by adding productions that can be uniquely be related to the terminals. We wish to do so such that when producing a production sequence then its terminals will show up in its production sequence. This can then be use when constructing the syntax tree. To do this extension we start by creating a secondary 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.

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

This does not affect which grammars can be parsed. To show this is correct for LLP parsing we first will need to show it is correct for LL parsing. The definition of a LL(k)[2] parsing table is as followed.

[2]This exact definition was found in my Bachelor thesis.

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. If we have a grammar G that is LL(k). Then for a production $A \to \delta \in P$ and its corresponding production $A \to \delta[A_{t_1}/t_1, \ldots A_{t_n}/t_1] = A' \to \delta' \in P_r$ we have.

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

Since the first and follow-sets are 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. □

Now to show that this extension does not change the grammars that can be parsed we need to be reminded of the PSLS 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:

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

Proof. If we are given an admissible pair (x, y) where $|\text{PSLS}(x, y)| = 1$ for some LL grammar G and 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 $(y, \alpha, ()) \vdash^* (y', \omega, \pi)$ where for any a that may be in the second set we have $a = \alpha$.

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

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

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

We know $B\gamma$ and $B'\gamma'$ are both produced by leftmost derivations. If we have two leftmost derivations one from G the other from G' with a common terminal string i.e. $S \Rightarrow^*_{lm} vC\mu$ and $S' \Rightarrow^*_{lm} vC'\mu'$. Then we would know they only differ by the need to derive on A_t. Since both G and G' are both LL grammars so there are not multiple ways of deriving v so the same production sequence is used but G' has the added A_t productions. Therefore there is a mapping of this kind of derivation from G' to G by substituting A_t with its terminal t. So therefore $(B'\gamma')[t_1/A_{t_1}, \dots t_n/A_{t_n}] = B\gamma$ and we can also go the other way $B'\gamma' = (B\gamma)[A_{t_1}/t_1, \dots A_{t_n}/t_1]$. We can do this since any A_t in $B'\gamma'$ would have not yet been derived due to it being a leftmost derivaiton seperated by a nonterminal. Since there is a one-to-one relation 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 $B\gamma$. Since you could map the multiple α' to mutiple α.

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

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

Where there only exists a single $a = \text{FIRST}^G_1(y)$ 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.

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

Once again we can use the same argumet, that there is a mapping between a leftmost derivations in G and in G' i.e. $(B'_a\gamma')[t_1/A_{t_1}, \dots t_n/A_{t_n}] = a\gamma$ and $B'_a\gamma' = (a\gamma)[A_{t_1}/t_1, \dots A_{t_n}/t_1]$. Therefore there is a one-to-one mapping between these two derivations. Knowing this and $B'_a \Rightarrow a$ then there only exist a single $a = \text{FIRST}^{G'}_1(B'_a)$ 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 $a = \mathrm{FIRST}_1^{G'}(B'_a) = \alpha'$ then α' is the only such prefix. □

3 Implementation

3.1 Tokenization

The implementation of tokenization is done by a variation of 2.1. We do not have to do fixed point iteration on alle of the endofunctions but only endofuctions that are neighbouring eachother. Then when you compose two transitions then you would need to add them as new transitions. 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 endofunctions.

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

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 $\delta(b) \circ \delta(c)$. 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.

3.2 Tree Construction

The alpacc parser generator is a parser generator which creates parser for the LLP grammer class. The LLP parsers described in the LLP parsing paper [4] only touches upon the verification and production sequence construction of parsing not the creation of the syntax tree. This would 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 [5, pp. 32–33] about compilation on the GPU. Here it is described how to create a syntax tree using the representation from Hsu PhD dissertation [3, pp. 77–81] where you have a parent that can be used for multiple transformations of a syntax tree.

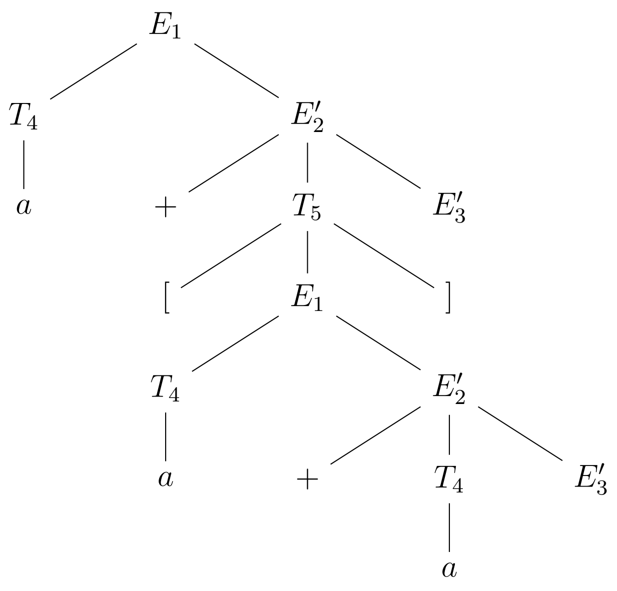
The representation of the tree will be an array of nodes where each node has the index to its parent node. The nodes will also be given in a preorder traversal of the tree. This kind of representation is much more fit for data parallelism than som recursive datatype, since recursion does not play well with data-parallelism.

3.2.1 Overview of 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.

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

This problem can be solved by extending a given 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.

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

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

Parent Index	Node
0	E_1
0	T_4
1	a
0	E'_2
3	$+$
3	T_5
5	$[$
5	E_1
7	T_4
8	a
7	E'_2
10	$+$
10	T_4
12	a
10	E'_3
5	$]$
3	E'_3

4 Conclusion

Conclusion.

References

[1] 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.

[2] 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.

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

[4] 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.

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

[6] 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.