Some attribute grammars are register transducers

Kasper Engelen
Department of Computer Science, University of Antwerp, Belgium
kasper.engelen@student.uantwerpen.be

Abstract

There exist many formalisms that are suited for computing regular string functions, among which are the register transducers. In this paper we propose that attribute grammars are also capable of computing these functions, and we exactly define the class of attribute grammars that are as expressive as the class of register transducers. Additionally, we propose a method to translate register transducers into attribute grammars and a method to perform the reverse translation. Finally, we argue that our findings make regular functions more accessible and that attribute grammars are very expressive and under-studied.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory → Automata extensions → Transducers

Keywords and phrases regular functions, string transducers, register transducers, attribute grammars

Acknowledgements I would like to thank Guillermo A. Pérez and Tim Leys for their advice and support, as well as Douéneau-Tabot et al. since one of their works serves as inspiration for the title of this paper.

1 Introduction

Formal language theory and automata theory are important fields within theoretical computer science. Throughout the years, much research has been performed on these subjects and applications can be found in the optimisation of algorithms [21], software verification [1, 22], and even linguistics [14]. An important contribution of these fields is the introduction of various formalisms, each of which may be characterised as a machine, as is common in automata theory [24], but also as a formal grammar, which is more usual for formal language theory [27]. Additionally, there are also characterisations related to algebra and logic [8].

Transducers. Many of the aforementioned formalisms form a model for a corresponding formal language. The machine or grammar will then be used to verify whether or not a certain word belongs to the language [11]. A natural extension of this is to provide an output for every input that is found to be part of the defined language [5]. Such a string-to-string function is called a *transducer* [6]. In the literature there are many automata-based representations of transducers, although alternative ones also exist [8]. There also exist transducers that operate on trees called the *tree transducers* [10], with the associated literature being extensive as is the case with string-to-string transducers.

Motivations. Our aim is to further the theory on transducers by introducing an attribute grammar based representation of the regular string-to-string functions. We motivate this goal using three main points.

First, we note that the importance and value of mathematical concepts often depends on the amount of characterisations of such concepts. Take for example the regular string-to-string functions, for which there exists a multitude of representations: logical, automata-theoretic, and algebraic [8, 15]. Having many representations for a certain class of functions is beneficial because each representation introduces new and useful properties [2, 15]. These may in turn be used to better understand such functions and how they are related to other types of functions. Examples of useful properties are equivalence [15], closure under composition [3], complexity [15], etc.

A second motivation is the study of class membership problems, potentially making use of the aforementioned characterisations and properties. An example of such a problem could be: “for an algorithm that belongs to a certain complexity class, find an equivalent algorithm in another class of lower complexity” [5, 6].

The third and final aspect is that we want to make the string-to-string functions more accessible to a wider audience. Currently the concept of regular functions is available to people familiar with various fields such as automata theory, logic, algebra, etc. By adding a grammar-based representation we hope to encourage people familiar with grammars to build and use regular string functions. Additionally, an attribute grammar can be implemented using existing parser tools, thereby providing automatic code generation for regular functions [16].

Contributions. In the case of tree-transducers, there exists literature depicting attribute grammars that operate on trees [10], as well as how they relate to other classes of transducers [7, 10]. Even though attribute grammars that operate on strings also have the ability to produce output through the use of attributes, no mention of grammar-based representations of string functions exist to our knowledge.

Mikołaj Bojańczyk [2] depicts the regular string functions as being part of a hierarchy. An overview of the different families of transducers is also given by Choffrut and Culik [4]. As stated earlier, we will present an attribute grammar that is capable of computing regular functions and storing the output in the parse tree produced by the grammar. Other families of string functions, transducers, and grammars such as the polyregular functions [2], the pushdown transducers [4], or the context-free grammars, are not included in this research.

Structure of this paper. Section 1 is the introduction. In section 2 we will introduce important background information and properties regarding transducers and grammars. In section 3 we will present the main theorem, namely the translation of register transducers to attribute grammars, as well as the reverse: translating specific attribute grammars into register transducers. Section 4 will contain a number of avenues for future research. Finally, in section 5 we will conclude this paper.

2 Preliminaries

An alphabet Σ is a finite set of symbols. The set Σ^* is the set of all possible words over the alphabet Σ, including the empty word ϵ. We denote by $w = w_1 \cdots w_n \in \Sigma^*$ a word w over the alphabet Σ which results from concatenating the letters $w_1, \cdots, w_n \in \Sigma$. We denote by $L(G)$ the language of a grammar G. Given $f : A \to B$ a function, then $\mathrm{dom}(f) \subseteq A$ is the domain of that function. We assume that the reader has background knowledge on automata, grammars, and formal languages (see [3], and [11]).

Deterministic Finite Automaton. We will first introduce an important class of machines. These are the deterministic finite automata (DFA) that read the input in one direction and whose semantics exactly correspond to the regular languages [26].

▶ **Definition 1.** *A deterministic finite-state automaton [26] can be defined as a 5-tuple $A = (Q, q_0, F, \Sigma, \delta)$:*

- *a finite set of states Q,*
- *an initial state $q_0 \in Q$,*
- *a set of final states $F \subseteq Q$,*
- *a finite input alphabet Σ,*
- *a transition function $\delta : Q \times \Sigma \to Q$.*

We will now explain the meaning and workings of DFAs. We first define a run [8, 19] of a DFA A on $w = w_1 \cdots w_n \in \Sigma^*$ as a sequence of transitions $\rho(w) = (q_0, w_1, q_1) \cdots (q_{n-1}, w_n, q_n)$ so that $\forall i \in \{1, \cdots, n\} : \delta(q_{i-1}, w_i) = q_i$. Such a run is *accepting* if $q_n \in F$. We can then define the language of A as $L(A) = \{w \,|\, \rho(w) \text{ is accepting}\}$. According to Kleene's theorem, the *language* of a DFA is regular (i.e. the set of words accepted by the DFA is a regular set), and also for every regular language there exists a DFA that accepts it [18].

Register transducers. We can extend the DFA with a finite number of registers whose contents are updated every time the machine takes a transition. Such machines are the so-called *register transducers*, also called the *streaming string transducers* [3].

▶ **Definition 2.** *We will define a register transducer [3, 6] as an 8-tuple $T = (\Sigma, \Delta, Q, R, q_0, v_0, \delta, F)$:*

- *a finite input alphabet Σ,*
- *a finite output alphabet Δ,*
- *a finite set of states Q,*
- *a finite set of registers R,*
- *an initial state $q_0 \in Q$,*
- *an initial assignment of register values $v_0 : R \to \Delta^*$,*
- *a transition function $\delta : Q \times \Sigma \to Q \times (R \to (R \cup \Delta)^*)$,*
- *a terminal output function $F : Q \to (R \cup \Delta)^*$.*

The workings of the register transducer are based on the underlying DFA $(Q, q_0, \text{dom}(F), \Sigma, \delta)$. The only difference is that we also have to keep track of the values of the registers and update those values on each transition.

▶ **Definition 3.** *A run of a register transducer on a string $w = w_1 \cdots w_n \in \Sigma^*$ is a sequence $\rho(w) = (q_0, w_1, q_1, v_1) \cdots (q_{n-1}, w_n, q_n, v_n)$ so that $\forall i \in \{1, \cdots, n\} : \delta(q_{i-1}, w_i) = (q_i, u_i)$, and v_i is a function that maps registers $r \in R$ to the values in Δ^* obtained after applying the update function u_i. Such a run is accepting if $q_n \in dom(F)$ [9].*

Next we will define the values of the registers [3]. The idea is that we take the update function u and resolve the elements in the image of the function that are also in the set R, by substituting them for the current values of those registers. To this end we will define a helper function val_f.

▶ **Definition 4.** *Let $val_f(x)$ be a homomorphism $val_f : (R \cup \Delta)^* \to \Delta^*$ that can be used to take a sequence in $(R \cup \Delta)^*$ and apply a function $f : R \to \Delta^*$ on elements of R, while elements of Δ are left alone:*

$$val_f(x) = \begin{cases} val_f(x_1) \cdots val_f(x_{|x|}) & \textbf{\textit{if}}\, |x| > 1 \\ x, & \textbf{\textit{if}}\, x \in \Delta \\ f(x), & \textbf{\textit{if}}\, x \in R. \end{cases}$$

▶ **Example 5.** Given $a, b \in (R \cup \Delta)^*$ it is true that $\text{val}_f(ab) = \text{val}_f(a)\text{val}_f(b)$. As such, it is clear that val_f is a homomorphism [11].

We are now ready to define the functions v_i that map registers in R to strings in Δ^*.

▶ **Definition 6.** *We define a function $v_i : R \to \Delta^*$ with $v_i(r) = val_{v_{i-1}}(u_i(r))$ so that v_i gives the value of the register r after reading the i-th input symbol. Such a register value is obtained by resolving the image of $u_i(r)$ using the old register values v_{i-1}. If $r \notin dom(u_i)$, let $v_i(r) = v_{i-1}(r)$. Note that initial register value function v_0 is part of the definition of the register transducer.*

▶ **Example 7.** Given a register transducer so that $\Delta = \{a, b\}, R = \{x\}$ and let there be a register update function $u : R \to (R \cup \Delta)^*$ so that $u(x) = axa$, and let us define a register value function $v_1 : R \to \Delta^*$ so that $v_1(x) = bbb$. If we then create a function $v_2(r) = \text{val}_{v_1}(u(r))$ for $r \in R$, then it is true that $v_2(r) = av_1(x)a = abbba$.

Lastly, in order to define the function that is computed by a register transducer, we will define the output of a run.

▶ **Definition 8.** *The output of a run $\rho(w)$ over a string $w \in \Sigma^*$ denoted by $out(\rho(w))$ is obtained by applying the terminal output function F to a final state $q_n \in dom(F)$, and replacing all elements of R by the current values of those registers contained in v_n [3]. More formally we can say that $out(\rho(w)) = val_{v_n}(F(q_n))$.*

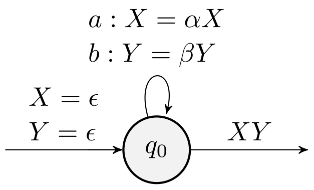


Figure 1 A visualisation of a register transducer.

▶ **Example 9.** In illustration 1 we see a register transducer that will output a string $\alpha^m\beta^n$ after reading m times the symbol a and n times the symbol b. The initial value function v_0 will map X and Y to the empty string ϵ. The transition function δ is defined so that $\delta(q_0, a) = (q_0, \{X \mapsto \alpha X\})$, and $\delta(q_0, b) = (q_0, \{Y \mapsto \beta Y\})$. Lastly, the terminal output function F is defined so that $F(q_0) = XY$. Note that the order in which the symbols a and b are read is not of importance.

Copyless register transducers. In the literature there is a distinction between *copyful* and *copyless* register transducers [3, 6, 9]. A register transducer is by default copyful, unless it conforms to the *copyless restriction.* This restriction means that on each transition, each register may only be used once by the update function, thus preventing register contents from being duplicated.

▶ **Definition 10.** *A register transducer $(\Sigma, \Delta, Q, R, q_0, v_0, \delta, F)$ conforms to the copyless restriction if:*

$$\forall q \in Q, \forall a \in \Sigma : \delta(q, a) = (\star, u), \forall r \in R : \sum_{x \in dom(u)} N_{u(x)}(r) \leq 1$$

Where $N_w(a)$ is the number of occurrences of the letter a in the string w. Which means that for every transition, a register r may only referenced once by the image of the update function u for that transition [6].

▶ Remark 11. It must be noted that copyless transducers only have linear-sized output, while the output of copyful transducers may be polynomial or even exponential [6].

▶ Remark 12. Note that the copyless restriction does not apply to the output function F. An output function for which $F(q) = XX$, with $q \in Q$ and $X \in R$, does not violate the copyless restriction [3].

Context-free grammars. Aside from automata, one can use a formal grammar to describe a formal language. There are many similarities between automata and grammars, with some of these models representing the same classes of languages [11]. We will first describe the context-free grammars (CFG).

▶ **Definition 13.** *Formally a CFG [11] is a 4-tuple* (V, T, P, S)*:*

- *A finite set of non-terminals or variables* V*,*
- *A finite set of terminals* T*,*
- *A set* $P \subseteq V \times (V \cup T)^*$ *that contains production rules,*
- *A start symbol* $S \in V$*.*

Production rules. The elements of the set P are called production rules. These are frequently written as $\alpha \to \beta$ with $\alpha \in V$ and $\beta \in (V \cup T)^*$. Here, α is called the head of the production, and β is called the body of the production [25]. Such production rules can also be seen as rewrite rules, that allow us to replace occurrences of variables in V with other symbols in $(V \cup T)$.

Derivations. In order to define the semantics of a CFG (i.e. the language accepted by a CFG), we use the productions of that grammar to infer that certain strings in $(V \cup T)^*$ are in the language of certain symbols in V. One way to do this is through derivation [11].

▶ **Definition 14.** *Let* $A \in V$ *and* $\alpha, \beta \in (V \cup T)^*$*, as well as* $A \to \gamma \in P$*. We can then define the derivation relation as* $\alpha A \beta \Rightarrow \alpha\gamma\beta$ *[11]. If* $u \Rightarrow v$*, we say that* u *directly yields* v *[25].*

▶ **Definition 15.** *If there exist* $u, v, \gamma_1, \cdots, \gamma_n \in (V \cup T)^*$ *so that* $u = \gamma_1$*,* $v = \gamma_n$*, and* $\gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n$*, we say that* u *yields* v*. We denote this by* $u \overset{*}{\Rightarrow} v$*. We can thus repeatedly apply the derivation relation [11].*

Language of the grammar. We have established that we may use derivation to infer the strings that are in the language of a symbol in V. As a result we can use derivation to infer strings that are in the language of the starting symbol S. We can use this fact to determine the language of a context-free grammar [11].

▶ **Definition 16.** *If we derive strings* $\alpha \in (V \cup T)^*$ *from the start symbol* S *(i.e.* $S \overset{*}{\Rightarrow} \alpha$*), we say that* α *is a sentential form [11].*

▶ **Definition 17.** *The language of the context-free grammar* $G = (V, T, P, S)$ *is the set of all sentential forms that are in* T^**, so that* $L(G) = \{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$ *[11].*

Parse trees. A useful way to represent a derivation, is through the use of parse trees. For each such derivation procedure, there exists a parse tree [11].

▶ **Definition 18.** *A parse tree [11] for the context-free grammar $G = (V, T, P, S)$ is a tree that conforms to the following conditions:*

- *each interior node is labelled by a variable in V,*
- *the label of each leaf is a terminal, a variable, or the empty string ϵ. In the latter case, such a leaf must be the only child of its parent, and*
- *an interior node, with label $X_0 \in V$ and children with labels $X_1, X_2, \cdots, X_k \in (V \cup T)$, corresponds with a rule $X_0 \to X_1X_2 \cdots X_k$ in P. Note that a node in the parse tree together with its children thus represents the application of a production rule in a derivation.*

Such a tree can be constructed during the derivation process as follows. The derivation will start with a variable $E \in V$, the resulting tree consists of a single node labelled E. Every time a rule $X_0 \to X_1X_2 \cdots X_k$ is applied on a variable during the derivation, take the node that corresponds to X_0, and add a child node for each variable $X_1, X_2, \cdots, X_k$.

Yield of a parse tree. If we take all the leaf nodes of the parse tree and concatenate them from left to right, we obtain the yield of the parse tree. If the root of the parse tree is the starting symbol S, such a yield is a sentential form. If the yield is a string in T^*, the yield is part of the the language of the grammar $L(G)$ [11].

▶ **Corollary 19.** *We can re-define the language of a context-free grammar by using the yield of the parse trees. This results in the language of the grammar being the set of yields in T^* of the parse trees that have the start symbol S as the root.*

Regular grammars. There exists a restriction we can place upon the class of context-free grammars. The grammars that conform to this restriction are either right-regular or left-regular and form the class of regular grammars. Regular grammars describe precisely the regular languages, and are thus equally powerful as finite-state automata [11, 28]. There exists an algorithm that transforms regular grammars into finite automata and vice versa [13]. Figure 2 illustrates two parse trees produced by regular grammars.

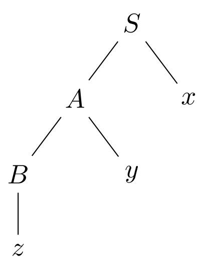
▶ **Definition 20.** *Given variables $A, B \in V$, and a terminal $a \in T$, a context-free grammar $G = (V, T, P, S)$ is said to be right-regular, also called right-linear, if it only contains rules in P of the following forms [28]:*

- *$A \to a$,*
- *$A \to aB$,*
- *$A \to \epsilon$.*

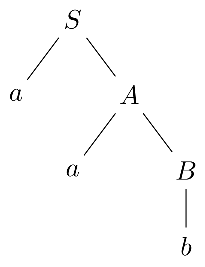
▶ **Definition 21.** *Given $A, B \in V$, and $a \in T$, a context-free grammar $G = (V, T, P, S)$ is said to be left-regular, also called left-linear, if it only contains rules in P of the following forms [28]:*

- *$A \to a$,*
- *$A \to Ba$,*
- *$A \to \epsilon$.*

Just like the regular grammars result from a restriction on the context-free grammars, the associated parse trees are a restricted form of the more general concept of a parse tree. In the following lemma we will describe the exact shape of such a “regular” parse tree.



(a) Parse tree for a left-linear grammar, with yield zyx.



(b) Parse tree for a right-linear grammar, with yield abc.

Figure 2 Examples of parse trees for regular grammars.

▶ **Lemma 22.** *Given a regular grammar $G = (V, T, P, S)$, let there be a parse tree that corresponds with the derivation of a string $w \in T^*$ using G. Such a parse tree will have the following properties:*

- *at every level, there will be at most one node labelled with a non-terminal,*
- *every node will have at most two children,*
- *if a node has two children, then the left (resp. right) child will be labelled with a non-terminal if G is left-linear (resp. right-linear), and*
- *if a node only has one child, then that child will be labelled with a terminal symbol or the empty string ϵ.*

Proof. We will perform a proof by induction on the length of the derivation, and we will assume G to be right-regular.

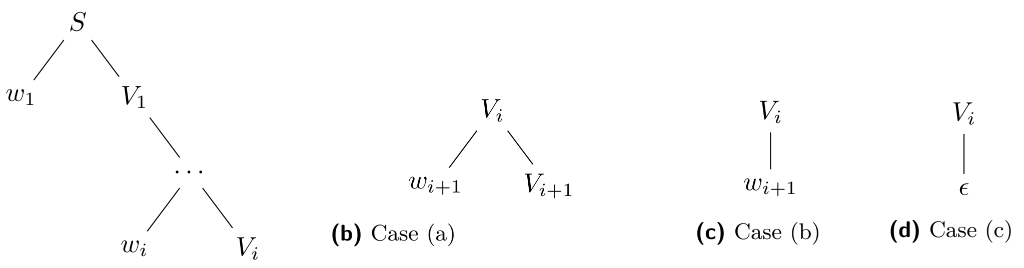
Base case: No production rules have been applied. The derivation now consists of only the starting symbol S. The corresponding parse tree consists of a single node N_0 labelled with S.

Induction step: The first i symbols of w have been derived. The derivation is of the form $S \overset{*}{\Rightarrow} w_1 \cdots w_i V_i$ and the corresponding parse tree is of the shape depicted in figure 3a. There are now, by definition, three types of production rules that may exist and which may be applicable in this situation, given a variable V_{i+1}, and a terminal symbol w_{i+1}:

1. rules of the form $V_i \to w_{i+1} V_{i+1}$,
2. of the form $V_i \to w_{i+1}$, or
3. of the form $V_i \to \epsilon$.

In case of (1) we obtain a derivation of the form $S \overset{*}{\Rightarrow} w_1 \cdots w_i V_i \Rightarrow w_1 \cdots w_i w_{i+1} V_{i+1}$. This means that we will add a left-child to N_i with label w_{i+1}, and a right child with label V_{i+1}. In case of (2) we obtain a derivation of the form $S \overset{*}{\Rightarrow} w_1 \cdots w_i V_i \Rightarrow w_1 \cdots w_i w_{i+1}$. This means that we will add a single child to N_i with labelled with w_{i+1}. In case of (3) we obtain a derivation of the form $S \overset{*}{\Rightarrow} w_1 \cdots w_i V_i \Rightarrow w_1 \cdots w_i$. This means that we will add a single child to N_i with the empty string ϵ as label. These three cases are illustrated in figure 3b, figure 3c, and figure 3d.

In the case of the left-regular grammar the proof will be similar, the difference being that we will derive the string from end-to-start, and that the parse tree shape is mirrored. ◀



(a) Before applying the induction step. **(b)** Case (a) **(c)** Case (b) **(d)** Case (c)

Figure 3 The different parse tree shapes that appear in the induction step of proof 2.

Attribute grammars. Aside from language recognition, a grammar may be used to produce an output. To this end we will assign every node in the parse tree a number of values called attributes. The value of such an attribute can be determined based on the attributes of other nodes in the parse tree. The dependencies between the attributes allow information to flow between the nodes of the tree. The output produced by the attribute grammar is then the parse tree itself, with attribute values being defined for the nodes of the parse tree.

▶ **Definition 23.** *An attribute grammar [17, 20] is a 6-tuple $G = (V, T, P, S, O, A, E)$:*

- *a finite set of non-terminals or variables V,*
- *a finite set of terminals T,*
- *a set $P \subseteq V \times (V \cup T)^*$ that contains production rules,*
- *a start symbol $S \in V$,*
- *an output alphabet O,*
- *a set of attribute symbols A,*
- *a set of semantic evaluation rules E that map variables and their attributes to a corresponding value.*

For the grammar we define a set of *attribute symbols* A. Such attribute symbols can be defined for one or more variables, so that if X is a variable then $A_X \subseteq A$ represents the set of attribute symbols defined for X [20].

Additionally, we define sets of *attribute instances* associated with each node in the parse tree. This means that if M is a node in the parse tree labelled with the symbol X, we have a set $\{a(X) \mid a \in A_X\}$ where $a(X)$ is such an attribute instance. The attribute instance will represent the value that an attribute has at a specific node in the parse tree [20].

We must now determine the value of such an attribute instance. Let there be a production rule p of the form $X_0 \to X_1 \cdots X_n$. For this production rule there are evaluation rules in $E_p \subseteq E$ which are then used to assign a value to an attribute instance $a(X_i)$.

Such a rule is of the form $a(X_i) = f(y_1, \cdots, y_r)$ and uses a function f to calculate the value based on other attribute instances $y_j \in \cup_{k=0}^{n} \{a(X_k) \mid a \in A_{X_k}\}$ (i.e. y_j is an attribute instance of one of the variables in the production rule) [20]. While f may be any kind of function, in this paper we will only use the concatenation function. In the future work section we will make are remark on other kinds of functions.

▶ Remark 24. Note that if we want to see an attribute grammar as a function that computes a single value, then we have to take the parse tree and transform it. This can, for example, be done by applying a function to the attribute instances of the nodes of the parse tree. In this paper however, the output of the function defined by an attribute grammar will simply be an attribute of the root of the parse tree.

▶ Remark 25. The value of an attribute may depend on the value of another attribute. In order to guarantee that all attributes are assigned a value during the evaluation process there must not be any circular dependencies between evaluation rules [20]. Knuth specifies an algorithm [12] to decide whether or not an attribute grammar contains circular attribute evaluation rules, and improvements [17] on this algorithm have since been made.

▶ Remark 26. There may be multiple evaluation rules that are applicable to an attribute at a given situation. In order to guarantee that the value of the attribute is independent of the order in which the evaluation rules are executed there must be exactly one rule for each attribute instance [20]. We can verify this by making sure that for each production rule, the associated evaluation rules all have a unique left-hand-side.

Synthesised and inherited attributes. We can divide the attributes and their evaluation rules into two disjoint categories. Let there be a production rule $X_0 \to X_1 \cdots X_n$. Then if a rule evaluates an attribute instance of the form $a(X_0)$ (i.e. an attribute of the head of the production), then the evaluation rule is called *synthesising*. If an attribute instance of the form $a(X_i)$ with $i > 0$ is evaluated (i.e. an attribute of the body of the production), then the rule is *inheriting* [20].

The category of the attribute depends on the category of the evaluation rule. If an attribute is evaluated using a synthesising evaluation rule, then the attribute is called *synthesised*. Likewise, if the attribute is evaluated using an inheriting rule, it is called *inherited*.

▶ Remark 27. Whether attributes are synthesised or inherited depends on the direction in which information flows within the parse tree. Synthesised attributes cause information to flow from the leaves to the root, while inherited attributes cause an information flow from the root to the leaves.

Copyless attribute grammars. If we look at the copyless restriction from definition 10, we can see that a similar restriction may be defined for attribute grammars. Let us take a set of evaluation rules E_p associated with a production rule $p \in P$. If it is true that every attribute $a \in A$ is referenced at most once by the right-hand-sides of the rules in E_p, then the set E_p may be called copyless. An attribute grammar as a whole is copyless if and only if all such sets E_p are copyless.

L-attributed grammars. A special class of attribute grammars are the so-called L-attributed grammars. An attribute grammar is L-attributed if for all production rules of the form $X_0 \to X_1 \cdots X_n$ it is true that the inherited attributes of X_j with $j > 0$ only depend on:

- the attributes of the variables $X_1, \cdots, X_{j-1}$,
- the inherited attributes of X_0, but not its synthesised attributes [23].

Restricting a grammar to be L-attributed allows all the attributes to be evaluated in a single depth-first left-to-right traversal of the parse tree [23].

3 Correspondence of register transducers and L-attributed regular grammars

The main goal of this paper is to describe a link between the class of register transducers and the class of attribute grammars. We will show that for each register transducer there exists a regular attribute grammar that computes the same string function. Additionally, we will exactly define the class of regular attribute grammars for which there exist register transducers that compute the same function. More formally, we can say that the class of string functions computed by the register transducers is a subset of the class of functions computed by the regular attribute transducers.

We will first demonstrate the translation from register transducers to regular attribute grammars. This translation is inspired by algorithm 4 of the paper by Zhang and Qian [13], and the intuition is as follows:

- the states of the transducer are represented by non-terminals (i.e. variables) in the grammar,
- the registers are replaced by attributes,
- the transitions are replaced by production rules,
- the update function associated with a transition is translated into semantic evaluation rules,
- the terminal output function of the transducer is replaced by ϵ-productions and their associated evaluation rules,
- the root of the parse tree will have an attribute called *out* which contains the output value of the function computed by the attribute grammar.

3.1 Construction of the attribute grammar

We will define a number of properties to which the attribute grammars will conform in theorem 32, and we will introduce a method to translate a register transducer into an attribute grammar that is compatible with such properties. Afterwards we will prove that by applying this method, the properties do indeed hold.

We will first make a number of trivial statements that mostly involve re-using some symbols associated with the register transducer in order to construct the associated attribute grammar.

▶ **Definition 28.** *When transforming a register transducer $M = (\Sigma, \Delta, Q, R, q_0, v_0, \delta, F)$ into an attribute grammar we let $G = (Q, \Sigma, P, q_0, \Delta, A, E)$ so that:*

- *A is equal to $R \cup \{out\}$,*
- *the set of states Q is also the set of variables,*
- *Σ is the set of input symbols,*
- *the starting symbol is the same as the starting state q_0,*
- *let every attribute in A be defined for each variable v in Q so that $A_v = A$, $\forall v \in Q$,*
- *for each attribute $a \in R$, its value associated with the start symbol q_0 is given by v_0 so that $a(q_0) = v_0(a)$ for all $a \in R$.*

All that is left is to construct the set of productions P and the set of evaluation rules E based on δ and F, which are respectively the transition function and terminal output function.

Translating the transition function δ into a set of productions P is very straightforward: for every transition $(q_1, a, q_2, u) \in \delta$ there exists a rule p in P of the form $q_1 \to aq_2$. In order to construct the associated set of semantic evaluation rules E_p, we first have to come up with a method to represent the update function u using such evaluation rules.

Recall that the update function is of the form $u : R \to (R \cup \Delta)^*$. We may additionally reinterpret evaluation rules as functions that map a variable and an attribute to a string in $(A \cup \Delta)^*$, since they calculate the attribute value based on other attribute values and output symbols. Both the update function and the reinterpreted evaluation rules map their inputs to the same set of outputs, given that R is equal to A if we ignore the *out* attribute.

The goal is, however, to map variables and their attributes to strings in Δ^*. To this end we will re-use the construction used in definitions 4 and 6 to construct the actual evaluation rules. We first take the image of the update function u, map it onto the set Δ^* using the val_f function, and then we assign the result to the attributes of q_2. Retrieving the attribute values of q_1 is done by using a helper function f_v (definition 29). Additionally, for the registers that are not part of the domain of u, the values must remain unchanged. To this end we will construct evaluation rules that simply copy the attribute value in q_1 to the attribute of q_2.

▶ **Definition 29.** *We define the function f_v as $f_v : A \to \Delta^* : f_v(a) = a(v)$. This function maps an attribute $a \in A$ on the value in Δ^* that the attribute has for the variable $v \in V$.*

Additionally, we must incorporate the terminal output function F into our grammar. Recall that F, when applied to the state q that the automaton is currently in, will combine the values of the registers into a single value. This value is then seen as the output of the register transducer. Similarly, we also want our attribute grammar to produce a single output value.

The idea here is to construct an epsilon production $q \to \epsilon$ for each state q in the domain of F. We then also create a synthesising evaluation rule that takes the attributes of q (which correspond to the values of the registers of the transducer when it is in state q), and combines them into a single value. All of this will be done based on the function value of F for the state q. Retrieving the attribute values of q will be done by using the helper function f_q from definition 29, and applying F will be done using the function val_f from definition 4.

Finally, we want this *out* value to appear at the root. To this end we add a synthesising evaluation rule of the form $out(q_1) = out(q_2)$ for all production rules of the form $q_1 \to aq_2$. This will cause the output value produced by the attribute grammar to "bubble up" to the root of the parse tree.

We thus come to the second definition that describes the conversion of register transducers into attribute grammars. Here, we formalise the translation of transitions into production rules.

▶ **Definition 30.** *When translating a register transducer M with transition function δ and terminal output function F into an attribute grammar G, we will construct the set of production rules P as follows: $P = \{q_1 \to aq_2 \,|\, (q_1, a, q_2, u) \in \delta\} \cup \{q \to \epsilon \,|\, q \in dom(F)\}$.*

Finally, we provide a definition that describes creating the semantic evaluation rules based on the update and terminal output functions.

▶ **Definition 31.** *If we want to construct an attribute grammar G based on a register transducer M with transition function δ and terminal output function F, we can construct the set of evaluation rules E as follows. For every production $p \in P$ of the form $q_1 \to aq_2$, constructed from the transition $(q_1, a, q_2, u) \in \delta$, we construct a set*

$$\begin{aligned} E_p = &\{r(q_2) = val_{f_{q_1}}(u(r)) \,|\, r \in R \cap dom(u)\} \\ &\cup \{r(q_2) = r(q_1) \,|\, r \in R - dom(u)\} \\ &\cup \{out(q_1) = out(q_2)\}. \end{aligned}$$

Additionally, for every production rule $p \in P$ of the form $q \to \epsilon$ we create a set $E_p = \{out(q) = val_{f_q}(F(q))\}$. Finally, we define $E = \cup_{p \in P} E_p$ to be the complete set of all evaluation rules in the grammar.

Algorithm 1 provides a more procedural illustration of the translation between register transducers and attribute grammars.

Algorithm 1 An algorithm that translates the transition function, the associated update functions, and the terminal output function into production and evaluation rules.

```
Inputs : The transition function $\delta$ and the terminal output function $F$ of the
         register transducer and the set of registers $R$.
Output : A set of production rules $P$, and a set of semantic evaluation rules $E$.
1  foreach $(q_1, a, q_2, u) \in \delta$ do
2      Create a production rule $p$ in $P$ of the form $q_1 \to aq_2$;
3      foreach $r \in R$ do
4          if $r \in dom(u)$ then
               /* Apply the update function to the attribute value.        */
5              Create an inheriting evaluation rule $r(q_2) = val_{f_{q_1}}(u(r))$ and add it to $E_p$;
6          else
               /* Copy the unchanged attribute value if the update function
                  is not applied.                                            */
7              Create an inheriting evaluation rule $r(q_2) = r(q_1)$ and add it to $E_p$;
8          end
9      end
10 end
11 foreach $q \in dom(F)$ do
       /* Add production rules for final states and apply output function
          F.                                                                 */
12     Create a production rule $p$ of the form $q \to \epsilon$ and add it to $P$;
13     Create a synthesising evaluation rule $out(q) = val_{f_q}(F(q))$ and add it to $E_p$;
14 end
15 Create the set of semantic evaluation rules $E = \cup_{p \in P} E_p$;
```

3.2 Proof of correctness

Next we will exactly define the class of attribute grammars that will result from translating the register transducers to equivalent transducers based on attribute grammars. This is one of the two main theorems of this paper.

▶ **Theorem 32.** *If we translate a register transducer M into an attribute grammar G using definitions 28, 30 and 31, the resulting grammar will have the following properties:*

1. *G is an L-attributed grammar,*
2. *G is a right-regular grammar,*
3. *the set of evaluation rules E only contains evaluation rules that make use of the concatenation operation,*
4. *every attribute $a \in A$ of G is defined for every variable $v \in V$ of G (i.e. $A_v = A$, $\forall v \in V$),*

5. *the non-epsilon production rules are the following: for every variable $v \in V$ and terminal $t \in T$ there will be exactly one production rule $p \in P$ of the form $v \to tX$ with $X \in V$ (i.e. G is deterministic, except for the epsilon-rules),*
6. *for each non-epsilon production rule and for each attribute there is exactly one inheriting evaluation rule,*
7. *the inheriting evaluation rule of the previous property may not reference the **out** attribute,*
8. *for each non-epsilon production rule there is exactly one synthesising evaluation rule, in which the value of the **out** attribute is copied from child to parent,*
9. *there are no inherited attribute rules that determine the value of the **out** attribute,*
10. *for each epsilon production rule there is exactly one synthesising evaluation rule, which determines the value of the **out** attribute,*
11. *the **out** attribute of the root of the parse tree will contain the output of the string function computed by G, which is the same as the output of the string function computed by M.*

We will use multiple proofs to prove these properties. We will commence by proving that G is an L-attributed grammar (property 1).

Proof of property 1. Recall that an attribute grammar is L-attributed if for all production rules of the form $X_0 \to X_1 \cdots X_n$ it is true that the inherited attributes of X_j with $j > 0$ only depend on:

1. the attributes of $X_1, \cdots, X_{j-1}$ (i.e. variables to the left),
2. the inherited attributes of X_0, but not its synthesised attributes.

In algorithm 30 there are two types of production rules that are created. The first one is of the form $q_1 \to aq_2$, with variables $q_1, q_2 \in V$, and with terminal $a \in T$. Since only q_1 and q_2 contain attributes, rule (1) is not applicable. The L-attributed constraint only concerns inherited attributes, so the synthesised *out* attribute is not relevant. The inherited attributes of q_2 on the other hand only depend on the inherited attributes of q_1. And therefore such attributes are correct due to (2).

The second type of production rule are those of the form $q \to \epsilon$, with $q \in V$. The only attribute here is the synthesised *out* attribute. As such the L-attributed constraint is not applicable here.

We thus conclude that the grammar G is L-attributed. ◀

In what follows we will prove properties 2-10 using trivial proofs that follow from the definitions.

Proof of properties 2-10. Let $M = (\Sigma, \Delta, Q, R, q_0, v_0, \delta, F)$ and let there be an attribute grammar $G = (Q, \Sigma, P, q_0, \Delta, R \cup \{\text{out}\}, E)$ created by applying definitions 28, 30 and 31.

In order to prove property 2, which says that G is regular, we simply note that all the production rules that are created in definition 30 are compatible with the requirements from definition 20.

For property 3 we must show that all evaluation rules only make use of concatenation. In definition 31 we see that the register update and output functions are directly used in the construction of the evaluation rules. Note that in definition 2 the register update function is defined using the Kleene star operation, which itself is a form of concatenation. Additionally definition 31 makes use of the val_f function from definition 4, makes use of concatenation.

Property 4 holds because from definition 28 it follows that all attributes are defined for every variable.

The grammar must be deterministic according to property 5, with the exception that there are a number of epsilon production rules. In definition 30 the transition function δ is used to construct the non-epsilon production rules. Since the register transducer from definition 2 has a deterministic transition function, a production rule will thus be constructed for every $q \in Q$ and $a \in \Sigma$. Which means that for every variable v, and every terminal t there exists a production rule $v \to tX$, with X being a variable in Q.

In definition 31 we can see that the sets $R \cap \mathrm{dom}(F)$ and $R - \mathrm{dom}(F)$ have no common elements, therefore there is no attribute $r \in R$ for which there are two evaluation rules associated with a single production rule. Therefore property 6 will hold.

To prove property 7 we note that *out* is not an element of the set R and thus is not referenced by the update functions u. Given that the inherited evaluation rules are constructed based on u, no such evaluation rule will reference the *out* attribute.

Properties 8-10 follow from definition 31, which specify that the value of the *out* attribute is evaluated in the epsilon production rules and then propagated to the root of the parse tree. ◀

Before we prove the final property, we will introduce an auxiliary lemma which states that the attributes of G and the registers of M have equal values.

▶ **Lemma 33.** *Let M be a register transducer and G be the grammar constructed by using definitions 28, 30 and 31. After processing the first i symbols of a string w, the following things will hold:*

1. *the transducer M will currently be in the state q_i, and the register values of q_i will be given by a function v_i,*
2. *the grammar G will have created a tree $\mathcal{N}$ with depth $i+1$ with a non-terminal leaf node N_i, labelled by q_i,*
3. *the attribute values of q_i will be the same as the register values given by v_i or, more formally, $r(q_i) = v_i(r)$ for all registers r.*

Proof. Let $M = (\Sigma, \Delta, Q, R, q_0, v_0, \delta, F)$ and let there be an attribute grammar $G = (Q, \Sigma, P, q_0, \Delta, R \cup \{\text{out}\}, E)$ created by applying definitions 28, 30 and 31.

Base case: We have not yet read any input characters. The run of M is currently equal to $\rho(\epsilon)$ and is thus empty. This means that the current state is q_0, and that the current registers are given by the function v_0. On the other hand, the current parse tree $\mathcal{N}$ produced by G consists of a single node N_0 labelled by q_0. The attribute values are given by v_0 so that $r(q_0) = v_0(r)$ for all $r \in R$. The lemma thus holds for the base case, with $i = 0$.

Induction step: We have up until now processed the symbols $w_1 \cdots w_{i-1}$. The run of M is of the form $\rho(w_1 \cdots w_{i-1}) = (q_0, w_1, q_1, v_1) \cdots (q_{i-2}, w_{i-1}, q_{i-1}, v_{i-1})$. The current state is q_{i-1} and the current register values are given by v_{i-1}.

As far as the grammar is concerned, it has up until now produced a parse tree $\mathcal{N}$ with non-terminal nodes $N_0, \cdots, N_{i-1}$ (see lemma 22 to get an idea of what the tree currently looks like). Per induction we will say that N_{i-1} is labelled with symbol q_{i-1}, and that $r(q_{i-1}) = v_{i-1}(r)$ for all $r \in R$.

We will now process w_i. Due to the determinism of δ it is true that there exists a transition $(q_{i-1}, w_i, q_i, u) \in \delta$, and as a result we extend the existing run into $\rho(w_1 \cdots w_{i-1} w_i) = (q_0, w_1, q_1, v_1) \cdots (q_{i-1}, w_i, q_i, v_i)$ (point 1). Additionally due to the construction in definition 30 there exists a production rule p of the form $q_{i-1} \to w_i q_i$. If we apply this production rule the node N_{i-1} gets a left-child labelled with w_i and a right-child N_i labelled by q_i (point 2).

By definition 6 the register values of M will now be $v_i(r) = val_{v_{i-1}}(u(r))$ for all $r \in R$. By construction the attribute values of q_i are given by $r(q_i) = val_{f_{q_{i-1}}}(u(r))$. Recall that because of definition 29 $f_{q_{i-1}}(r) = r(q_{i-1})$ for all $r \in R$, and that by induction $r(q_{i-1}) = v_{i-1}(r)$. As a result $v_i(r) = r(q_i)$ for all $r \in R$ (point 3).

Because of points (1), (2), and (3) we can conclude that the lemma holds. ◀

We are now ready to prove property 11 of theorem 32, which says that the attribute grammar G and the register transducer M compute the same function. For reasons of clarity we repeat property 11 in the following proposition.

▶ **Proposition 34.** *Let M be a register transducer, and let G be the attribute grammar constructed using definitions 28, 30 and 31. The **out** attribute of the root node of the parse tree, created using G by reading a string w, will contain the output of the function computed by G, and will be the same as the output of the function computed by M.*

Proof. Let $M = (\Sigma, \Delta, Q, R, q_0, v_0, \delta, F)$ and let there be an attribute grammar $G = (Q, \Sigma, P, q_0, \Delta, R \cup \{\text{out}\}, E)$ created by applying definitions 28, 30 and 31. Additionally, let $w \in \Sigma^*$ be a string of length n in the language of the underlying DFA of M so that when w is read by M a run $\rho(w) = (q_0, w_1, q_1, v_1) \cdots (q_{n-1}, w_n, q_n, v_n)$ is produced with $q_n \in \text{dom}(F)$.

When we parse w using G, we may use lemma 33. We then obtain a tree $\mathcal{N}$ of depth $n+1$ with non-terminal nodes $N_0, \cdots, N_n$, where each N_i is labelled by q_i. Because of definition 30 there will exist a production rule $q_n \rightarrow \epsilon$ because $q_n \in \text{dom}(F)$. If we apply this rule, we obtain a tree with yield $w_1 \cdots w_n \epsilon = w_1 \cdots w_n = w$ so that w is in the language of G (corrollary 19).

Recall from definition 8 that the output of a run ρ on a string w is defined as $\text{out}(\rho(w)) = val_{v_n}(F(q_n))$, and that according to definition 31 a production rule $q_n \rightarrow \epsilon$ will have an evaluation rule $\text{out}(q_n) = val_{f_{q_n}}(F(q_n))$. The latter is also described by property 10 of theorem 32. Due to lemma 33 it goes that $f_{q_n}(r) = v_n(r)$ for all $r \in R$, and thus it is also true that $\text{out}(q_n) = \text{out}(\rho(w))$.

The only thing that is left to prove, is that this value is equal to the *out* attribute of the root node of the parse tree. In definition 30, as well as in property 8 of theorem 32, we have defined a synthesising evaluation rule, for each non-epsilon production rule, that simply copies the *out* attribute from the child node to the parent node. Also note that in the construction of the parse tree, an epsilon production rule was only applied at the end.

We can thus conclude that for the *out* attribute of the root node it is true that $\text{out}(q_0) = \text{out}(\rho(w))$ (i.e. the *out* attribute of the root contains the output of M). ◀

3.3 Translating (some) attribute grammars into register transducers

Since we have demonstrated that all register transducers may be translated into an equivalent attribute grammar, it can also be interesting to do the opposite: translate attribute grammars back into register transducers. We therefore arrive at the second important theorem of this paper.

▶ **Theorem 35.** *If G is an attribute grammar with the properties described in theorem 32, or if G can be transformed so that the properties hold, then G can be translated into a register transducer M so that G and M compute the same function.*

Before proving the theorem we will first give some intuition. The basic idea is that we reverse the construction of definitions 28, 30 and 31. If we take a quick look at these definitions we can quite easily see that they should not be too difficult to reverse. A central part of our argument will be that such a reversal is possible if the properties of theorem 32 hold.

More practically, the construction will first re-use a number of symbols of G in order to construct M. The transition function and associated update functions will be derived from the non-epsilon production rules. The evaluation rules associated with the epsilon production rules of G will be transformed into the terminal output function of M.

Proof. To start off, we will introduce some properties of theorem 32 that facilitate the construction of M based on G:

- G is a regular grammar according to property 2, which makes it possible to translate G into an automaton-based formalism,
- in property 3 it is stated that the evaluation rules will only use the concatenation operation, which makes it possible to translate the rules into update and output functions (definitions 2 and 8),
- according to property 4, all attributes are defined for all production rules, this means that we can at all times map them to registers of M,
- the determinism of the transition function will be guaranteed by property 5.

Let $G = (V, T, P, S, O, A, E)$. We then reverse definition 28 and let $M = (T, O, V, R, S, v_0, \delta, F)$, where $R = A - \{\text{out}\}$ and so that $v_0 : R \to O^*$ provides the values of the attributes associated with the starting symbol S.

Next, we will construct the transition function δ. For every rule p of the form $q_1 \to aq_2$ in P with $a \in T$ and $q_1, q_2 \in V$, there exists a transition $(q_1, a, q_2, u) \in \delta$. This will be the case for all $q_1 \in V$ and $a \in T$ due to property 5, thus making δ deterministic.

Constructing the update function u can easily be done since due to properties 3 and 6 there will exist an evaluation rule for p that only makes use of the concatenation operation. We can thus translate these into update functions. The evaluation rules that simply copy attribute values are discarded.

We then take the epsilon production rules and transform them into the terminal output function F. For every production rule of the form $q \to \epsilon$, there will exist an evaluation rule that determines the value of the *out* attribute by using the concatenation operation (properties 3 and 8). We can thus define F more formally so that $F(q) = y_1 \cdots y_n$ for all rules $\text{out}(q) = y_1 \cdots y_n$ associated with a production rule $q \to \epsilon$.

In the proof of proposition 34 we showed that the value of the *out* attribute of the head of an epsilon production is the same as the *out* attribute of the root of the parse tree. According to property 11 the *out* attribute of the root contains the output of the function computed by G. As a result the transducer M will produce the same output as G, since its output is defined by the terminal output function F (see definition 8). ◀

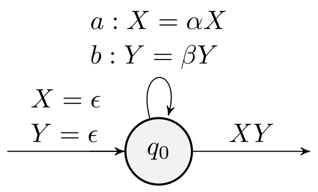


Figure 4 A visualisation of a register transducer.

3.4 An illustrative example

We will demonstrate the proposed translation between register transducers and attribute grammars by giving an example. We begin by taking the register transducer from example 9 illustrated again in figure 4. More formally this is a register transducer $M = (\Sigma, \Delta, Q, R, q_0, v_0, \delta, F)$ so that:

- $\Sigma = \{a, b\}$,
- $\Delta = \{\alpha, \beta\}$,
- $Q = \{q_0\}$,
- $R = \{X, Y\}$,
- $\delta(q_0, a) = (q_0, \{X \mapsto \alpha X\})$ and $\delta(q_0, b) = (q_0, \{Y \mapsto \beta Y\})$,
- the function F with $F(q_0) = XY$ and $\mathrm{dom}(F) = \{q_0\}$.

We start off by defining an attribute grammar $G = (Q, \Sigma, P, q_0, \Delta, A, E)$ according to definition 28, with $A = R \cup \{\mathrm{out}\}$. When we apply definitions 30 and 31 we obtain three production rules in P and the associated evaluation rules in $E = E_{q_0 \to aq_0'} \cup E_{q_0 \to bq_0'} \cup E_{q_0 \to \epsilon}$:

$$P = \left\{ \begin{array}{ll} q_0 \to aq_0', & \text{since } \delta(q_0, a) = (q_0, \{X \mapsto \alpha X\}) \\ q_0 \to bq_0', & \text{since } \delta(q_0, b) = (q_0, \{Y \mapsto \beta Y\}) \\ q_0 \to \epsilon & \text{since } q_0 \in \mathrm{dom}(F) \end{array} \right\}$$

$$E_{q_0 \to aq_0'} = \left\{ \begin{array}{rcl} X(q_0') & = & val_{f_{q_0}}(\alpha X) = \alpha f_{q_0}(X) = \alpha X(q_0), \\ Y(q_0') & = & Y(q_0), \\ out(q_0) & = & out(q_0') \end{array} \right\}$$

$$E_{q_0 \to bq_0'} = \left\{ \begin{array}{rcl} X(q_0') & = & X(q_0), \\ Y(q_0') & = & val_{f_{q_0}}(\beta Y) = \beta f_{q_0}(Y) = \beta Y(q_0), \\ out(q_0) & = & out(q_0') \end{array} \right\}$$

$$E_{q_0 \to \epsilon} = \left\{ out(q_0) = val_{f_{q_0}}(XY) = f_{q_0}(X) f_{q_0}(Y) = X(q_0)Y(q_0) \right\}$$

The next step is to demonstrate the reverse translation: converting the attribute grammar G back into a register transducer M'. We do this by following the construction described by the proof of theorem 35. To remain brief, we will only show how to construct a transition function δ' and an output function F' out of the sets of production and evaluation rules P and E.

Constructing the transition function is possible by taking every production rule of the form $q_1 \to aq_2$ and then defining δ' so that $\delta'(q_1, a) = (q_2, u)$. The update function u is then defined by taking the associated set of evaluation rules and discarding those that only copy a value. We then end up with the transition function δ' for M' so that it is identical to the transition function δ of M:

$$\begin{array}{rcll} \delta'(q_0, a) & = & (q_0, \{X \mapsto \alpha X\}) & \text{due to prod. rule } q_0 \to aq_0' \text{ and eval. rule } X(q_0') = \alpha X(q_0) \\ \delta'(q_0, b) & = & (q_0, \{Y \mapsto \beta Y\}) & \text{due to prod. rule } q_0 \to bq_0' \text{ and eval. rule } Y(q_0') = \beta Y(q_0) \end{array}$$

In order to come up with an output function F' we will take the epsilon production rule $q_0 \to \epsilon$ and the associated evaluation rule $out(q_0) = X(q_0)Y(q_0)$. We thus end up with $F'(q_0) = XY$, which is identical to F of the original register transducer M.

4 Future work

Transducers. Our idea of using attribute grammars for regular string functions began with the idea of investigating whether or not pebble transducers could be represented using attributed context-free grammars. When investigating the pebble transducers and the polyregular functions [2] they represent, we came to the conclusion that there are many other types of string functions that can be computed by transducers. We also saw that there was no mention in the literature of using attribute grammars to compute such string functions. As such we chose the regular string functions as a starting point since there exists an elegant translation between register transducers and attribute grammars. That still leaves many types of transducers for which no corresponding attribute grammars are defined, and we encourage further study on the subject.

Generalisations. There exist other automata that are capable of performing computations on registers, similarly to register automata. For example, there are *cost register automata* that are DFAs equipped with registers and a way to update those registers. The difference with ordinary register automata is that they are more expressive because they are not restricted to the free monoid. A CRA can, for example, perform algebraic operations on numbers. One can see that the same generalisation can be applied to the evaluation rules used by attribute grammars. In our view it would be interesting to study such generalisations and develop grammars that are as powerful as the different types of CRAs.

5 Conclusion

We have proposed a way to translate register transducers into attribute grammars that compute the same string functions. Additionally, we described the resulting class of attribute grammars and we have provided a reverse translation. This lead us to conclude that the class of string functions computed by register transducers is a subset of the class of string functions computed by the attribute grammars.

In doing so we have introduced the attribute grammars as an appropriate way to represent the string functions, which to our knowledge had not been done up until now. This has the benefit that properties of register transducers apply to certain attribute grammars and vice versa. A second benefit is making the regular string functions known to people that are familiar with grammars, so that string functions computed by transducers may now be automatically synthesised by parser generators.

Finally, we have proposed a number of future research possibilities. The first one involves finding attribute grammars that represent other types of string transducers. The second one being the generalisation of the concepts explored in this paper: register transducers, attribute grammars, and the translation between those two concepts. We can clearly see that attribute grammars are very expressive and under-studied, especially in the context of computing string-to-string functions.

References

1 Rajeev Alur and Pavol Cerny. Algorithmic verification of single-pass list processing programs, 2011. `arXiv:1007.4958`.

2 Mikołaj Bojańczyk. Polyregular functions. *CoRR*, abs/1810.08760, 2018. URL: `http://arxiv.org/abs/1810.08760`, `arXiv:1810.08760`.

3 Mikołaj Bojańczyk and Wojciech Czerwiński. *An Automata Toolbox*. 2018.

4 Christian Choffrut and Karel Culik II. Properties of finite and pushdown transducers. *SIAM Journal on Computing*, 12(2):300–315, 1983. `doi:10.1137/0212019`.

5 Gaëtan Douéneau-Tabot. Pebble transducers with unary output, 2021. `arXiv:2104.14019`.

6 Gaëtan Douéneau-Tabot, Emmanuel Filiot, and Paul Gastin. Register transducers are marble transducers, 2020. `arXiv:2005.01342`.

7 Joost Engelfriet and Sebastian Maneth. Macro tree transducers, attribute grammars, and mso definable tree translations. *Information and Computation*, 154(1):34–91, 1999. URL: `https://www.sciencedirect.com/science/article/pii/S0890540199928079`, `doi:https://doi.org/10.1006/inco.1999.2807`.

8 Emmanuel Filiot and Pierre-Alain Reynier. Transducers, logic and algebra for functions of finite words. *ACM SIGLOG News*, 3(3):4–19, August 2016. `doi:10.1145/2984450.2984453`.

9 Emmanuel Filiot and Pierre-Alain Reynier. Copyful Streaming String Transducers. In *11th International Workshop on Reachability Problems (RP 2017)*, London, United Kingdom, September 2017. URL: `https://hal.archives-ouvertes.fr/hal-01782442`.

10 Z. Fülöp and H. Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers.* Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 1998. URL: `https://books.google.be/books?id=VnchAQAAIAAJ`.

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

12 Donald Ervin Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2:127–145, 2005.

13 Jie lan Zhang and Zhong sheng Qian. The equivalent conversion between regular grammar and finite automata. *Journal of Software Engineering and Applications*, 6:33–37, 2013.

14 Mehryar Mohri. Finite-state transducers in language and speech processing. *Comput. Linguist.*, 23(2):269–311, jun 1997.

15 Anca Muscholl and Gabriele Puppis. The Many Facets of String Transducers. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, volume 126 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:21, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2019/10241`, `doi:10.4230/LIPIcs.STACS.2019.2`.

16 T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810, jul 1995. `doi:10.1002/spe.4380250705`.

17 Michael Rodeh and Mooly Sagiv. Finding circular attributes in attribute grammars. *J. ACM*, 46(4):556–ff., jul 1999. `doi:10.1145/320211.320243`.

18 G. Rozenberg, G.R.A. Salomaa, and A. Salomaa. *Handbook of Formal Languages: Volume 1. Word, Language, Grammar.* Handbook of Formal Languages. Springer, 1997.

19 Frédéric Servais and JF Raskin. Visibly pushdown transducers. *ULB, Belgique*, 2011.

20 Martti Tienari. On the definition of attribute grammar. In *Semantics-Directed Compiler Generation, Proceedings of a Workshop*, page 408–414, Berlin, Heidelberg, 1980. Springer-Verlag.

21 Moshe Y. Vardi. Nontraditional applications of automata theory. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, TACS '94, page 575–597, Berlin, Heidelberg, 1994. Springer-Verlag.

22 Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986)*, pages 332–344. IEEE Computer Society Press, June 1986.

23 Wikipedia contributors. L-attributed grammar — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=L-attributed_grammar&oldid=968092926`, 2020. [Online; accessed 31-October-2021].

24 Wikipedia contributors. Automata theory — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Automata_theory&oldid=1054656338, 2021. [Online; accessed 18-December-2021].

25 Wikipedia contributors. Context-free grammar — Wikipedia, the free encyclopedia, 2021. [Online; accessed 29-October-2021]. URL: https://en.wikipedia.org/w/index.php?title=Context-free_grammar&oldid=1046808016.

26 Wikipedia contributors. Deterministic finite automaton — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Deterministic_finite_automaton&oldid=1011556620, 2021. [Online; accessed 30-September-2021].

27 Wikipedia contributors. Formal language — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Formal_language&oldid=1058399401, 2021. [Online; accessed 18-December-2021].

28 Wikipedia contributors. Regular grammar — Wikipedia, the free encyclopedia, 2021. [Online; accessed 29-October-2021]. URL: https://en.wikipedia.org/w/index.php?title=Regular_grammar&oldid=1024323125.