Automatic Theorem Proving in Walnut

Hamoon Mousavi
February 29, 2016

Contents

1 Introduction

Walnut is a software package that implements a *mechanical decision procedure* for deciding certain combinatorial properties of some special words referred to as *automatic words* or *automatic sequences*. To learn more about automatic words and their applications, see [5]. To learn about decision procedures for automatic words, see Schaeffer's Master's thesis [12] and the survey paper [13]. To read more about decidable properties of automatic words, refer to [6]. To read about another software package that provided a similar mechanical decision procedure for automatic words, and was developed before Walnut, read Goc's Master's thesis [7]. To see applications of Walnut, refer to [3, 4, 8–11].

The aim of this article is to introduce Walnut and explain its core features. This article consists of four parts: basics, syntax, implementation, and the Walnut guide. In the first part, Section 2, we establish the basic notation and concepts. We go over words, automata, number systems, automatic words, and Presburger arithmetic. We learn what it means for an automaton to accept a predicate. We also learn how to automatically decide properties of automatic words.

The second part, Section 3, talks about the building blocks of predicates: constants, variables, operators, and different types of expressions. The semantics of predicates in Presburger arithmetic are well-known and are not explained, whereas semantic rules for calling and indexing, with which we extend the Presburger arithmetic to include automatic words, are explained in detail.

The third part, Sections 4 and 5, explains the decision procedure implemented in Walnut. The cross product of two automata, which is behind the construction of automata for all binary logical operators, is introduced. Building on that, we see how to construct automata for predicates from automata for subpredicates. In Section 5, we talk about two types of automata that do not appear often in Walnut, but are nevertheless important to understand.

The fourth and last part, Sections 6–8, starts with Walnut's installation and goes over all of its commands, i.e., exit, eval, def, reg, and load. In Section 8, we learn how to manually define automata in text files. We also learn how to define new number systems.

If you are already familiar with the objects described in the first sentence of this introduction, you can skip Section 2 and come back to it only as a reference. For a more comprehensive treatment of the theory behind decision procedures for automatic words refer to [6, 12, 13].

Since this article is more about Walnut than the theory behind it, when we explain the latter, we use Walnut's notation as opposed to the more familiar mathematical notation. For example, we use & and A for conjunction and universal quantifier as opposed to $\wedge$ and $\forall$ of mathematical logic [1]. As another example, when we define structures such as number systems or objects such as automatic words, we give the definitions that are closer to Walnut's capabilities than the most general theoretical ones possible. This will help the reader make a smoother transition from the theory to its application in Walnut.

You can download Walnut from Jeffrey Shallit's website. Walnut is written in Java and is open source. It is licensed under GNU General Public License. We would appreciate it if users cite this article in their publications. For automata minimization and converting regular expressions to automata, Walnut relies on the automata library in [2]. We would greatly appreciate it if users report bugs to sh2mousa@uwaterloo.ca. The author would like to thank Jeffrey Shallit for revising this article.

[1] Users enter logical predicates in a terminal when they use Walnut. We find that entering latex-like commands in the terminal, e.g., \forall, does not improve the readability.

2 Basics

2.1 Words and Automata

A word $(a_i)_{i\in I}$ for a finite, infinite, or a possibly empty subset I of natural numbers $\mathbb{N}$, is a sequence of symbols a_i over a finite set called an alphabet. The set I usually equals $\mathbb{N}$ or $\mathbb{N}_l = \{k \in \mathbb{N} : k < l\}$ for some l. The set of finite and infinite words over the alphabet Σ are denoted by Σ^* and Σ^ω, respectively. The empty word is denoted by ϵ. For the finite word $w = a_0a_1\cdots a_{l-1}$, the length $|w|$, is defined and equals l. We let Σ^l denote the set of all words over Σ of length l. A subword (sometimes called "factor" in the literature) is a finite and contiguous subsequence of a word. The subword of w starting at position i of length $k \ge 0$ is denoted by $w[i..i+k-1] = a_i \cdots a_{i+k-1}$. Many interesting properties of words can be expressed in terms of their subwords. For example, the property of having two equal and adjacent subwords, referred to as a square, is discussed in numerous papers in the area of combinatorics on words. The product of two words x and y, denoted by xy, is the result of concatenating x by y.

There are cases where our words are defined over alphabets consisting of tuples of symbols, so let us fix our notation regarding these words. For a word w over an alphabet $\Sigma_1 \times \Sigma_2 \ldots \times \Sigma_n$, we let the projection map $\pi_j(w)$ for $1 \le j \le n$ denote the word over Σ_j, obtained from w by looking at the j'th coordinates, i.e., words $\pi_j(w)$ are uniquely defined by

$$w = \prod_{i=0}^{|w|-1} \big(\pi_1(w)[i], \pi_2(w)[i], \ldots, \pi_n(w)[i]\big).$$

For example, for $w = (0,1)(1,1)(0,0)$ over $\{0,1\} \times \{0,1\}$ we have $\pi_1(w) = 010$ and $\pi_2(w) = 110$.

The reader is probably familiar with the notions of deterministic and nondeterministic finite state automata. In Walnut, an automaton M with n inputs (input tapes), is an $(n+4)$-tuple $\big(Q, q_0, F, \delta, \Sigma_1, \Sigma_2, \ldots, \Sigma_n\big)$, where Q is the (finite) set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, $\delta : Q \times \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_n \to Q$ is the transition function, and Σ_i is the alphabet of the i'th input (tape). The automaton's alphabet is defined to be the cross product $\Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_n$, and the notions of accepting a word w or a language over this alphabet is defined as usual. A nondeterministic automaton is defined similarly, except that the transition function is defined by $\delta : Q \times \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_n \to 2^Q$. In Walnut and throughout this article, the Σ_i are finite subsets of integers $\mathbb{Z}$.

Two automata are equal (isomorphic) if their underlying graphs are isomorphic. Two automata are equivalent if they accept the same language. There exists a determinization algorithm that converts a nondeterministic automaton to an equivalent deterministic automaton. There exists a minimization algorithm that converts an automaton to an equivalent automaton with the least number of states (which is unique up to isomorphism). It is known that extending the automata model by allowing multiple initial states (similar to how there can be multiple final states) does not add to the model's expressiveness.

Next we extend the notion of accepting languages to relations, since the latter is more natural in Walnut:

Definition 1 (relations computed by automata)**.** The relation $R \subset \Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_n^*$ computed/accepted by M is defined by

$$R = \big\{\big(\pi_1(w), \pi_2(w), \ldots, \pi_n(w)\big) : M \text{ accepts } w\big\}.$$

Since for every word w, the words $\pi_i(w)$ are all of the same length, the relation R accepted by an automaton is consisted of tuples of the words of the same length, i.e., we have

$$R \subseteq \bigcup_{l\ge 0} \big(\Sigma_1^l \times \Sigma_2^l \times \cdots \times \Sigma_n^l\big) \subset \Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_n^*.$$

For example, the language accepted by the following automaton is $L = (0,0)^*(1,1)(0,0)(0,1)$, whereas the relation accepted is $R = \big\{(w_1, w_2) : w_1 \in 0^*100, w_2 \in 0^*101, \text{ and } |w_1| = |w_2|\big\}$:

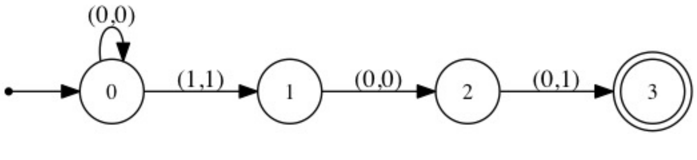


Figure 2.1: Automaton accepting tuples of same length representations of 4 and 5 in binary

In other words, the automaton accepts tuples $t = (w_1, w_2) \in \{0,1\}^* \times \{0,1\}^*$ where w_1 and w_2 are representations of the **the same length**, in the most-significant-digit-first binary system, of natural numbers 4 and 5 respectively. On the other hand, referring to the words w in $(\{0,1\} \times \{0,1\})^*$ that are accepted by this automaton is not very descriptive. That is why, in this article, we prefer the relation (tuple) terminology over the language (word) terminology.

In almost all depictions of the underlying graphs of automata, such as the one in Figure 2.1, when a transition is not specified, it is assumed to be a transition to a dead state. In Walnut we do not store transitions to the dead state. Adding the dead state and all implicit transitions to it, is called totalizing an automaton.

An automaton with output is a tuple $\left(Q, q_0, O, \delta, \Sigma, \Sigma_1, \Sigma_2, \ldots, \Sigma_n\right)$ where Q, q_0, δ, Σ_j are as before, the set Σ is the output alphabet, and, instead of a set of final states, we have a map $O : Q \to \Sigma$. The symbol $O(q)$ is called the output of the state q. An automaton with output can be thought of as an automaton that reads a word over $\Sigma_1 \times \Sigma_2 \cdots \times \Sigma_n$ and outputs whatever is the last state's output. In Walnut, the output alphabet Σ is a finite subset of integers. We can think of ordinary automata as a special case of automata with output by letting the set of final states to be $F = \{q : O(q) \neq 0\}$. This is indeed how ordinary automata are stored in Walnut.

In the next section, we learn how to add more structure to alphabets by defining number systems. As we saw in the example, the automaton in Figure 2.1 accepts binary representations of numbers. In a moment we will extend our definition of automata to $\left(Q, q_0, F, \delta, \mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_n}\right)$, where the $\mathbf{S_j}$ are number systems and concealed in them are alphabets $\Sigma_{\mathbf{S_j}}$ among other things.

2.2 Number Systems

In any course on theory of computation, it is customary to talk about the representations of the objects an algorithm/Turing machine takes as inputs. At the core of Walnut are automata taking natural numbers as inputs, and doing various computations on them, so fixing a representation for natural numbers is essential. We could limit ourselves to binary representations. However, there are many interesting automata accepting representations in number systems other than the binary one. So we are going to define, in general terms, the concept of a number system. Walnut allows number systems to be defined and used (with a few restrictions to the general definition below).

Definition 2 (number systems)**.** A number system $\mathbf{S}$ is a 3-tuple $(\Sigma_{\mathbf{S}}, R_{\mathbf{S}}, []_{\mathbf{S}})$ of alphabet $\Sigma_{\mathbf{S}} \supseteq \{0,1\}$, language $R_{\mathbf{S}} \subset \Sigma_{\mathbf{S}}^*$ of valid representations containing 0^* and at least one of 0^*1 or 10^*, and decoding function $[]_{\mathbf{S}} : R_{\mathbf{S}} \to \mathbb{N}$ that assigns integers to every word in $R_{\mathbf{S}}$ and for which $[]_{\mathbf{S}}(w)$ is usually written as $[w]_{\mathbf{S}}$. The decoding function has the following additional properties:

- $[z]_{\mathbf{S}} = 0$ if and only if $z \in 0^*$
- $[1]_{\mathbf{S}} = 1$
- For all $w \in R_{\mathbf{S}}$, either $zw \in R_{\mathbf{S}}$ and $[zw]_{\mathbf{S}} = [w]_S$ for all $z \in 0^*$, or $wz \in R_{\mathbf{S}}$ and $[wz]_{\mathbf{S}} = [w]$ for all $z \in 0^*$. The former is called an **msd** number system and the latter is called an **lsd** number system[2].
- For all positive $n \in \mathbb{N}$, there exists $w \in R_{\mathbf{S}}$ for which $[w]_{\mathbf{S}} = n$ and $w[0] \neq 0$ if $\mathbf{S}$ is **msd** or $w[|w|-1] \neq 0$ if $\mathbf{S}$ is **lsd**. The word w, if unique, is called the canonical encoding of n in $\mathbf{S}$, and is sometimes denoted by $(n)_{\mathbf{S}}$. We let $(0)_{\mathbf{S}} = \epsilon$.

The addition relation $+_{\mathbf{S}} \subset R_{\mathbf{S}}^3$ is defined such that $(x, y, z) \in +_{\mathbf{S}}$ if and only if x, y, z are of the same length and $[x]_{\mathbf{S}} = [y]_{\mathbf{S}} + [z]_{\mathbf{S}}$. The equality relation $=_{\mathbf{S}} \subset R_{\mathbf{S}}^2$ is defined such that $(x, y) \in =_{\mathbf{S}}$ if and only if x and y are of the same length and $[x]_{\mathbf{S}} = [y]_{\mathbf{S}}$. The less than relation is defined as $<_{\mathbf{S}} \subset R_{\mathbf{S}}^2$ for which $(x, y) \in <_{\mathbf{S}}$ if and only if x and y are of the same length and $[x]_{\mathbf{S}} < [y]_{\mathbf{S}}$. We adopt the in-order notation for $+_{\mathbf{S}}$, $=_{\mathbf{S}}$, and $<_{\mathbf{S}}$, i.e., we write $x = y +_{\mathbf{S}} z$, $x =_{\mathbf{S}} y$, and $x <_{\mathbf{S}} y$ as opposed to the more cumbersome $(x, y, z) \in +_{\mathbf{S}}$, $(x, y) \in =_{\mathbf{S}}$, and $(x, y) \in <_{\mathbf{S}}$ respectively. It follows from the definition that for all $n \in \mathbb{N}$, the set of representations of n in $\mathbf{S}$, defined by $\{w : [w]_{\mathbf{S}} = n\}$ is non-empty.

For example, the most-significant-digit binary system, denoted by **msd_2**, is defined by $(\{0,1\}, \{0,1\}^*, []_{\mathbf{msd\_2}})$ where

$$[w]_{\mathbf{msd\_2}} = \sum_{i=0}^{|w|-1} [w[i]]_{\mathbf{msd\_2}} 2^{|w|-i-1},$$

[2] **msd** and **lsd** are short for most-significant-digit-first and least-significant-digit-first, respectively. However, it should not be taken literally in this definition, as one could define **msd** number systems (in the sense defined here), with no direct correspondence to the notion of most-significant-digit-first representation.

e.g., $[001001]_{\mathbf{msd\_2}} = 0\cdot 2^5 + 0\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 9$. For **msd_2**, we are very fortunate to have simple automata computing all of its important aspects, namely, valid representations $R_{\mathbf{msd\_2}}$, the addition relation $+_{\mathbf{msd\_2}}$, the equality relation $=_{\mathbf{msd\_2}}$, and the less-than relation $<_{\mathbf{msd\_2}}$. See Figures 2.2,2.3,2.4, and 2.5 respectively.

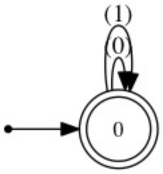


Figure 2.2: Automaton computing $R_{\mathbf{msd\_2}}$

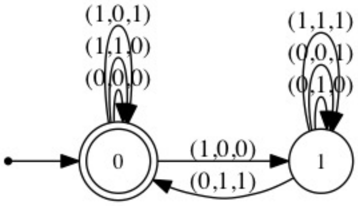


Figure 2.3: Automaton computing $+_{\mathbf{msd\_2}}$

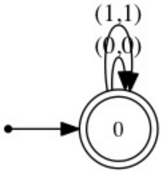


Figure 2.4: Automaton computing $=_{\mathbf{msd\_2}}$

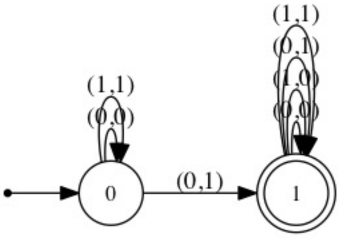


Figure 2.5: Automaton computing $<_{\mathbf{msd\_2}}$

We can define the least-significant-digit-first binary system, denoted by **lsd_2**, in a similar way. In fact, we can define **msd_n** and **lsd_n** for all $n \geq 2$, and for all of them, there are simple automata computing valid representations, addition, equality, and less-than relations. In fact we can define the following:

Definition 3 (number systems in Walnut)**.** Number systems for which the automata for representations, addition, equality, and less-than exist, and equality is the same as word equality, i.e., $x =_{\mathbf{S}} y$ if and only if $x = y$, are exactly the type of number systems one can define and use in Walnut. Note that the alphabet of a number system is restricted to finite subsets of $\mathbb{Z}$ due to the same restriction on automata in Walnut.

In addition to base-n number systems, Walnut has a built-in definition for the Fibonacci number system.

The most-significant-digit-first Fibonacci system, denoted by **msd_fib**, is defined by $(\{0,1\}, 0^*(\epsilon \mid 1)(0 \mid 01)^*, []_{\mathbf{msd\_fib}})$ where

$$[w]_{\mathbf{msd\_fib}} = \sum_{i=0}^{|w|-1} [w[i]]_{\mathbf{msd\_fib}} F_{|w|-i-1},$$

where F_i is the i'th Fibonacci number given by $F_0 = 1, F_1 = 2$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$. For example, $[001001]_{\mathbf{msd\_fib}} = 0 \cdot F_5 + 0 \cdot F_4 + 1 \cdot F_3 + 0 \cdot F_2 + 0 \cdot F_1 + 1 \cdot F_0 = 6$. The set of valid representations is exactly the set of binary words avoiding consecutive 1s. The avid reader might want to verify that **msd_fib** is a number system. There are automata computing all major aspects of **msd_fib**. For example, here is the automaton accepting $R_{\mathbf{msd\_fib}}$[3]:

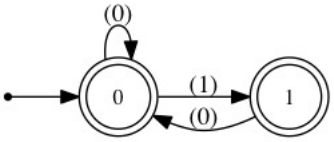


Figure 2.6: Automaton computing $R_{\mathbf{msd\_fib}}$

In cases, where an automaton's inputs are representations of integers in some number system, which by far are the most important type of automata in Walnut, we would like to signify these number systems instead of the input alphabets. For example, we might write $(Q, q_0, F, \delta, \mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_n})$ to mean $(Q, q_0, F, \delta, \Sigma_{\mathbf{S_1}}, \Sigma_{\mathbf{S_2}}, \ldots, \Sigma_{\mathbf{S_n}})$. It should be understood that in these cases, if for a word w input $\pi_j(w)$ is not a valid representation in $\mathbf{S_j}$, it does not mean that the automaton's behavior is not defined for w. This just means that w is, by default, not going to get accepted. The behaviors of both automata and automata with output that are taking representations of numbers in some number systems as inputs are defined for all words (even those not representing numbers in the given number systems).

2.3 Automatic Words

An automatic word $W = (a_i)_{i \geq 0}$ is a word in Σ^ω for which there exists a number system $\mathbf{S}$ and an automaton with output $M(Q, q_0, O, \delta, \Sigma, \mathbf{S})$ for which reading $x \in R_{\mathbf{S}}$ outputs $W[[x]_{\mathbf{S}}] = a_{[x]_{\mathbf{S}}}$. In other words, for an automatic word, the symbol at position i for all i can be effectively computed by running an automaton with output on any single representation of i in a number system. As usual we assume Σ is a finite subset of $\mathbb{Z}$.

The word T for which the symbol at position i, is the number of 1s in any binary representation of i, modulo 2, is called the Thue-Morse word. The Thue-Morse word is well-defined since all the infinitely many different binary representations of an integer have the same number of 1's. It is instantly clear that T is an automatic word over **msd_2** if one notes the automaton with output in Figure 2.7.

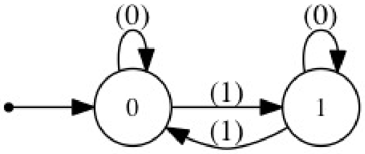


Figure 2.7: The Thue-Morse word

In the introduction, we mentioned that Walnut decides some properties of automatic words. Recall from Section 2.1 that squares are non-empty words of the form xx. It is easy to see that T has square subwords. The following predicate captures

[3] The automaton accepting $+_{\mathbf{msd\_fib}}$ has 16 states, which is too big to be represented here.

this property:

$$\exists i \exists n \forall j, j < n \implies T[i+j] = T[i+n+j].$$

Walnut provides a decision procedure that takes predicates like this and decides whether they are true or false. Walnut does so, by constructing automata for every subpredicate in the predicate above; see Section 2.4 for more details. It starts by constructing from the automaton in Figure 2.7 an automaton $M_1\big(Q, q_0, F, \delta, \mathbf{msd\_2}, \mathbf{msd\_2}, \mathbf{msd\_2}\big)$ for subpredicate $P_1 := T[i+j] = T[i+j+n]$. This means (see Section 2.4) that M_1 is constructed so that it accepts tuples $t = (w_1, w_2, w_3)$ if and only if $|w_1| = |w_2| = |w_3|$ and substitutions $i = [w_1]_{\mathbf{msd\_2}}$, $j = [w_2]_{\mathbf{msd\_2}}$, and $n = [w_3]_{\mathbf{msd\_2}}$ are satisfying P_1. Walnut then using M_1 constructs an automaton M_2 for $P_2 := \forall j, j < n \implies T[i+j] = T[i+n+j]$. The automaton M_2 takes two inputs representing the two free variables i and n in P_2. Walnut continues by constructing the automaton M_3 for $P_3 := \exists n \forall j, j < n \implies T[i+j] = T[i+n+j]$. In the end, Walnut returns true if M_3 accepts anything. The fact that M_1,M_2, and M_3 exist is explained in Section 2.4. The details of how Walnut constructs these automata are explained in Section 4. The details of what comprises a valid predicate is explained in Section 3. To see more examples of the properties of the Thue-Morse word and their proofs see Section 7.1.

We can extend the definition of automatic words to higher dimensions. The (n-dimensional) automatic word

$$W = \big(a_{i_1, i_2, \ldots, i_n}\big)_{i_1 \geq 0, i_2 \geq 0, \ldots, i_n \geq 0}$$

is an infinite word over Σ for which there exist number systems $\mathbf{S_j}$ and an automaton with output

$$M\big(Q, q_0, O, \delta, \Sigma, \mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_n}\big)$$

for which reading x, such that $\pi_j(x) \in R_{\mathbf{S_j}}$ for all j, outputs

$$W\big[[\pi_1[x]]_{\mathbf{S_1}}\big]\big[[\pi_2[x]]_{\mathbf{S_2}}\big] \cdots \big[[\pi_n[x]]_{\mathbf{S_n}}\big] = a_{[\pi_1[x]]_{\mathbf{S_1}}, [\pi_2[x]]_{\mathbf{S_2}}, \ldots, [\pi_n[x]]_{\mathbf{S_n}}}.$$

2.4 Automata accepting Predicates

In Walnut, we are interested in automaton M accepting same-length representations in number systems $\mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_n}$ of integers $x_1, x_2, \ldots, x_n$ satisfying some predicate P. When this is the case we say that automaton M accepts the predicate P (or equivalently$M$ accepts relation R of tuples satisfying P). We already saw a few examples of such automata in Figures 2.1–2.6. From [1], also see [12], and as it will be proved again in Section 4, for predicate P in Presburger arithmetic such an automaton always exists. Presburger arithmetic is the first-order theory of natural numbers, in which predicates are consisted of constants (natural numbers), variables over natural numbers, existential quantifiers, universal quantifiers, logical operators (conjunction, disjunction, negation, exclusive disjunction, implication, equivalence), arithmetic operators (addition, subtraction, multiplication and division by constants), and comparison operators (equality, less than, greater than, less than or equal, greater than or equal)[4].

You can find the list of all operators in table 3.1. This list has three operators, namely, reverse ` ,indexing [], and calling $, that are not allowed in Presburger arithmetic. By indexing we mean indexing into an automatic word, e.g., writing things like $W[i+j] = W[i+n+j]$; see Section 3.6 for more details. In [13],[6],[12], and also in Section 4.6 we learn that extending Presburger arithmetic to include indexing is still decidable. In Section 3.7 we learn about calling and in Section 4.5 we learn that it is just a syntactic sugar and does not add to the power of the extended Presburger arithmetic (one that includes indexing into automatic words). We learn about reverse operation in Section 4.3. From here on, by "predicate" we mean a predicate over this extended Presburger arithmetic (extended to include indexing into automatic words) and until we see the proof in Section 4, we accept the fact that there exist automata accepting such predicates.

In Section 3 we formally define what constitutes a predicate, but first let us see a few examples:

- $P_1 := a = 4 \,\&\, b = 5$
- $P_2 := a = b + c$
- $P_3 := \mathrm{A}x\ \mathrm{E}y\ x = 2 * y \mid x = 2 * y + 1$
- $P_4 := T[i+j] = T[i+n+j]$

[4] Presburger arithmetic in its formal definition recognizes only a minimal subset of constants and operators: 0,1,+,=,<,∀, but it is not difficult to show that all the other objects and operators we mentioned, e.g., multiplication by constants, does not add to the power of Presburger arithmetic and can be derived from that minimal set of objects. See Section 3.2 for more details. One thing to note here is that subtraction $a - b$ exists only when there exists a non-negative number c for which $b + c = a$.

We adopt the terminology of free variables from mathematical logic, i.e., a variable that is not bound to a quantifier (quantified). For example P_3 has no free variables, and can be regarded as a constant, in this case it is always true.

We have seen that, given a predicate P, for any ordering $x_1, x_2, \ldots, x_n$ of free variables and for every assignment of number systems $\mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_n}$ to those variables, there exists an automaton M accepting such a predicate, i.e., a tuple of same length words $t = (w_1, w_2, \ldots, w_n)$ is accepted by M if and only if the substitutions $x_i = [w_i]_{\mathbf{S_i}}$ satisfy P.

For example, consider the predicate P_1. The automaton in Figure 2.1 accepts P_1. Furthermore there exists automaton M accepting tuples (x, y) for which $|x| = |y|$ and substitutions $a = [y]_{\mathbf{msd\_2}}$, and $b = [x]_{\mathbf{msd\_2}}$ are satisfying P_1. There also exists an automaton N accepting tuples (x, y) for which $|x| = |y|$ and substitutions $a = [x]_{\mathbf{msd\_fib}}$ and $b = [y]_{\mathbf{lsd\_2}}$ are satisfying P_1. By definition, both M and N also accept the predicate P_1.

We would like to annotate predicates so that they contain information on number systems without ambiguity (we will see how shortly). For such an annotated predicate P and the ordering $x_1, x_2, \ldots, x_n$ on free variables, there exists a unique minimized automaton accepting the predicate. We denote this unique automaton by

$$(x_1, x_2, \ldots, x_n) : P.$$

The ordering we fix on variables, in Walnut and throughout this article, is the lexicographic ordering on the variables' name.

The following are examples of annotated predicates[5]:

- $P_1' :=$?msd_2 $a = 4 \,\&\, b = 5$
- $P_2' :=$?msd_fib $a = b + c$

From the annotated predicate P_1' we understand that $a, b, 4, 5$ should all be interpreted in **msd_2** and = should be interpreted as $=_{\mathbf{msd\_2}}$. Hence $(a, b) : P_1'$ is the automaton accepting **msd_2** representations of 4 and 5 as its first and second inputs respectively. Also from annotation ?msd_fib in P_2' it is clear what to expect from automaton $(a, b, c) : P_2'$.

We can annotate a predicate with multiple number systems, e.g., see Figure 2.9. Here are the rules with which we assign number systems to constants, variables, and operators in a predicate:

- If ?S appears inside a pair of parentheses or brackets, then the number system **S** is effective from the place it occurs in the predicate to the nearest closing parenthesis or bracket[6].
- If ?S appears outside all parentheses and brackets, then the number system **S** is effective from the place it occurs in the predicate to the end of predicate.
- If none of the rules above applies, the number system is assumed to be **msd_2** by default.
- It is assumed that the number systems do not contradict each other, i.e., a single variable cannot have two different number systems in one predicate, and all operands of an arithmetic or comparison operator must belong to the same number system.

We saw in Figure 2.1, the *unique* automaton $(a, b) : P_1'$. In Figure 2.8, we see the automaton $(a, b) : a = 4 \,\&\, b = 13$ (recall that when the number system is not specified it is assumed to be **msd_2**):

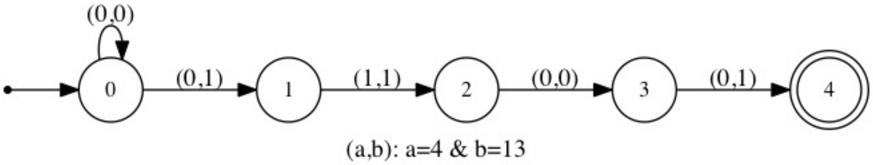


Figure 2.8: The automaton accepting $a = 4 \,\&\, b = 13$, does not accept all representations of 4

Note how this automaton fails to accept $t = (100, w_2)$ for any w_2. This is obviously due to the fact that 13 does not have a representation of length 3 in **msd_2**. So we stress again that when we say automaton M accepts predicate P, we mean

[5] Names for variables, words, and automata in Walnut start with a letter and can contain alphanumerics and underscores. So to distinguish number system annotations in a predicate we use the prefix ?.

[6] Brackets [] only appear in indexing expressions. See Sections 3 and 3.6 for more details.

that M accepts all (tuples of) *equal length representations* of $x_1, \ldots, x_n$ satisfying P. Therefore this example conforms to the definition.

Let us see an example of an automaton having multiple number systems. Figure 2.9 depicts the automaton $(a,b) : a = 1\,\&\,(\text{?lsd\_2 } b = 1)$.

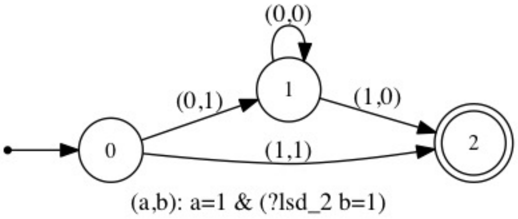


Figure 2.9: Automaton accepting $a = 1\,\&\,(\text{?lsd\_2 } b = 1)$

3 Syntax and Semantic of Predicates in Walnut

3.1 Alphabets

We mentioned in earlier sections that all input and output alphabets of automata are subsets of integers in Walnut. Specifically for any automatic word W, we can assume $W[i]$ is an integer.

3.2 Arithmetic and Alphabetic Constants

Arithmetic constants in a predicate are allowed to be natural numbers only. There is, however, another type of constant: the alphabetic constant. Alphabetic constants are useful when referring to symbols at particular positions in automatic words. For example, the predicate that accepts positions for which the automatic word W is 1 is written as $W[i] = @1$. In order to draw the distinction between alphabetic and arithmetic constants, we use alphabetic constants with a prefix of @. The reason we call these constants alphabetic (as opposed to arithmetic) is due to the fact that Walnut does not allow (and it does not make much sense to allow) predicates that are comparing indexing expressions 3.6 and arithmetic expressions 3.5, e.g., expressions such as $W[i] = a + b$ is not allowed. As we will see in Section 3.8, the only objects that can be compared with indexing expressions are alphabetic constants and indexing expressions themselves.

Alphabetic constants are ordered like ordinary integers, so we can compare alphabetic constants, just like we can compare arithmetic constants. For example, $@-1 < @1$ is a valid predicate, and it is always true; see Sections 3.6 and 3.8. *However, we cannot use alphabetic constants in arithmetic expressions.*

3.3 Variables

A variable's name must start with a letter and can contain upper- and lower-case alphanumerics and underscores. A variable's name cannot be E or A.

3.4 Operators

The full list of operators allowed in predicates can be found in Table 3.1[7]. This list has operator precedences. The lower this number is, the higher the precedence is. For example, multiplication by constant has the highest precedence. Parentheses override all precedences. All operators are associative from left to right, except for complement ~, reverse ` , quantifiers E and A, calling $, and indexing [] which are all associative from right to left.

[7] we prefer this notation to those familiar from mathematical logic, because we want to liken our notation to those of programming languages, as Walnut is ultimately a programming language.

precedence	operator	explanation	examples
1	$*$	multiplication by a constant	$2 * x$ and $x * 2$
1	/	division by a constant	$x/2$ but not $2/x$
2	+	addition	
2	−	subtraction	
3	=	equality	
3	!=	inequality	
3	<	less than	
3	>	greater than	
3	<=	less than or equal	
3	>=	greater than or equal	
4	~	complement	
4	`	reverse	
5	&	conjunction	
5	|	disjunction	
5	^	exclusive disjunction	
6	=>	implication	
7	<=>	equivalence	
8	E	existential quantifier	Ex, y, z or Ex Ey Ez
8	A	universal quantifier	Ax, y, z or Ax Ay Az
9	$	calling	\$M(x, y)
9	[]	indexing	$T[i + j]$

Table 3.1: List of operators in Walnut

3.5 Arithmetic Expressions

The permissible arithmetic operators are $+, -, *, /$. Equality = is *not* an arithmetic operator. A constant expression is an expression involving only constants and arithmetic operators that evaluates to a natural number, e.g., $4, 3+2, 6/4, 2*3$ but not -3 nor $2-3$. An arithmetic expression is defined recursively in the usual way:

- A constant expression is an arithmetic expression, e.g., 2,10,$7-4$, but not -1.
- A variable is an arithmetic expression, e.g., x, y, z,etc.
- For arithmetic expression e, the expression (e) is also arithmetic.
- For arithmetic expression e_1 and e_2 both of $e_1 + e_2$ and $e_1 - e_2$ are arithmetic expressions.
- For variable x and constant expression c all of $x * c$,$c * x$, and x/c are arithmetic expressions.
- For arithmetic expression e and constant expression c all of $(e) * c$, $c * (e)$, and $(e)/c$ are arithmetic expressions.

An arithmetic expression on its own is not a predicate, and it is not meaningful to talk about an automaton accepting an arithmetic expression. For example, talking about an automaton accepting $x + y + z = 0$ makes sense, while talking about an automaton accepting $x + y + z$ is not meaningful. Walnut reports an error if the user tries to construct an automaton for an arithmetic expression.

See Section 4.4 to see how Walnut constructs automaton for valid predicates like

$$(y_1 \otimes y_2 \otimes \cdots \otimes y_m) \circledcirc (x_1 \otimes x_2 \otimes \cdots \otimes x_n),$$

where the x_i and y_j are variables or arithmetic constants, $\otimes$ are arithmetic operators, and $\circledcirc$ is a comparison operator.

3.6 Indexing Expressions and Their Semantic Rules

For an n-dimensional automatic word W, an indexing expression is $W[e_1][e_2]\cdots[e_n]$ where the e_i are either arithmetic expressions or predicates with one free variable.

An indexing expression on its own is not a valid predicate, and it is not meaningful to talk about automata accepting indexing expressions. Smallest predicates involving indexing expressions are defined in Section 3.8 and they involve comparison operators.

We use indexing expressions to refer to positions indicated by e_i. The semantic of predicates involving indexing expressions can be derived from the following rule:

Definition 4 (semantic rule regarding indexing)**.** Suppose automatic word W, expressions $e_1, e_2, \ldots, e_n$ where the e_i are either arithmetic expressions or predicates with one free variable, free variables $x_1, x_2, \ldots, x_m$ occurring in the e_i, and an alphabetic constant α are given. Predicate $W[e_1][e_2]\cdots[e_n] = @\alpha$ is satisfied by substitutions $x_k = v_k$ for all k, if all of the following hold:

- If e_i is an arithmetic expression, then a_i is the value of the e_i when evaluated at $x_k = v_k$ for all k.
- If e_i is a predicate with one free variable, then it is satisfied by substitutions $x_k = v_k$ for all k. Let a_i equals v_k when x_k is the free variable in e_i.
- The symbol $W[a_1][a_2]\ldots[a_n]$ equals α.

Having this rule, coming up with similar rules for other comparison operators, e.g., $W[e_1][e_2]\cdots[e_n] < @\alpha$, and even predicates involving comparison of two automatic words, e.g., $W_1[e_1][e_2]\cdots[e_m]$>=$W_2[e'_1][e'_2]\cdots[e'_n]$, should be straightforward. Recall that alphabetic constants are ordered just like integers.

3.7 Calling Expressions and Their Semantic Rules

For an automaton M with n inputs a calling expression is $\$M(e_1, e_2, \ldots, e_n)$ where the e_i are either arithmetic expressions or predicates with one free variable. For such an expression, we say that M is called with arguments $e_1, e_2, \ldots, e_n$. A calling expression on its own is a valid predicate, as we will see in Section 3.8.

Definition 5 (semantic rule regarding calling)**.** Suppose M is the automaton $y_1, y_2, \ldots, y_n : P$ for some predicate P. Suppose expressions $e_1, e_2, \ldots, e_n$ where the e_i are either arithmetic expressions or predicates with one free variable, and free variables $x_1, x_2, \ldots, x_m$ occurring in the e_i are given. Predicate $\$M(e_1, e_2, \cdots, e_n)$ is satisfied by substitutions $x_k = v_k$ for all k, if all of the following hold:

- If e_i is an arithmetic expression, then a_i is the value of e_i when evaluated at $x_k = v_k$ for all k.
- If e_i is a predicate with one free variable, then it is satisfied by substitutions $x_k = v_k$ for all k. Let a_i equals v_k when x_k is the free variable in e_i.
- P is satisfied by substitutions $y_i = a_i$ for all i.

3.8 Relative Expressions

Comparison operators are =,!=,<,>,<=, and >=. A relative expression is any of the following:

- An expression $e_1 \olessthan e_2$ where e_1 and e_2 are arithmetic expressions and $\olessthan$ is any comparison operator.
- An expression $e_1 \olessthan e_2$ where e_1 and e_2 are indexing expressions and/or alphabetic constants and $\olessthan$ is any comparison operator.
- A calling expression is a relative expression.

We stress that $W[a] = b+2$ is not a relative expression based on the definition above, since $W[a]$ is an indexing expression and $b+2$ is an arithmetic expression. We will see shortly that any relative expression is a predicate. Section 4.4 explains how to construct automata accepting relative expressions.

3.9 Predicates

A predicate is an expression formed from relative expressions and logical operators:

- Every relative expression is a predicate.
- For every predicate P all of (P), ~ (P) and `(P) are predicates.
- For every predicate P_1 and P_2 all of $P_1 \,\&\, P_2$, $P_1 \mid P_2$, $P_1 \wedge P_2$, P_1 => P_2, P_1 <=> P_2 are predicates.
- For every predicate P and free variables $x_1, x_2, \ldots, x_n$ both of $\mathrm{E}x_1, x_2, \ldots, x_n\ P$ and $\mathrm{A}x_1, x_2, \ldots, x_n\ P$ are predicates.

The semantic rules with which we assign true and false values to predicates defined here can be obtained by adding the semantic rules for indexing and calling to the well-known semantics of first-order logic and Presburger arithmetic.

Walnut provides two commands for converting predicates to automata accepting them: eval and def; see Sections 7.1 and 7.2, respectively.

4 Decision Procedure: Walnut's Implementation

In this section, we learn about a procedure that takes a predicate and constructs an automaton accepting that predicate. The procedure explained here is what implemented in Walnut, and we shall call it the decision procedure.

For every defined number system, Walnut knows the automata for valid representations, addition, equality, and less-than predicates/relations. Every predicate is ultimately built out of these four predicates using logical operators. So we only need to explain the construction of automata for complex predicates from automata for simpler subpredicates. We start by explaining cross product in Section 4.1, which is the core object when constructing automata for predicates formed from binary logical operators, i.e., &,|,∧,=>,<=>. Then we move on to quantification in Section 4.2, explaining the construction of automata for predicates formed from E and A operators. In Section 4.3, we discuss construction of automata for the complement ~ and reverse ` operators. With these tools at our disposal, we are on the right track to construct automata for complex predicates formed from comparison and arithmetic operators, e.g., ∗,/,>,<=,etc. which we explain in Section 4.4.

4.1 Cross Product

Let $M\big(Q, q_0, F, \delta, \mathbf{S_1}, \ldots, \mathbf{S_m}\big)$ and $M'\big(Q', q_0', F', \delta', \mathbf{S_1'}, \ldots, \mathbf{S_n'}\big)$ be the automaton $(x_1, \ldots, x_m) : P$ and $(x_1', \ldots, x_n') : P'$ respectively. Let us assume that if $x_i = x_j'$ then $\mathbf{S_i} = \mathbf{S_j'}$. Let $\{x_1'', \ldots, x_p''\}$ where $p \le m+n$ be the union of $\{x_1, \ldots, x_m\}$ and $\{x_1', \ldots, x_n'\}$ and further assume that the x_i'' are appearing in lexicographic order. Depending on whether $x_k'' = x_i$ or $x_k'' = x_j'$, let $\mathbf{S_k''}$ denote $\mathbf{S_i}$ or $\mathbf{S_j'}$ respectively. Then the cross product of M and M' denoted by $M \times M'$ is the tuple

$$\big(Q \times Q', (q_0, q_0'), \delta'', \mathbf{S_1''}, \ldots, \mathbf{S_p''}\big)$$

where the transition function is defined to be

$$\delta''\big((q, q'), (\gamma_1, \ldots, \gamma_p)\big) = \big(\delta(q, (\alpha_1, \ldots, \alpha_m)), \delta'(q', (\beta_1, \ldots, \beta_n))\big)$$

for γ_k equals α_i or β_j depending on whether $x_k'' = x_i$ or $x_k'' = x_j'$ respectively. Note that $M \times M'$ is not an automaton since a set of final states is not specified. For $F'' \subseteq Q \times Q'$, let $(M \times M')(F)$ denote the automaton $\big(Q \times Q', (q_0, q_0'), F'', \delta'', \mathbf{S_1''}, \ldots, \mathbf{S_p''}\big)$.

Theorem 6. *For $F'' = \big\{(q, q') : q \in F \text{ and } q' \in F'\big\}$, the automaton $(M \times M')(F'')$ accepts predicate $P \,\&\, P'$. Furthermore, minimizing $(M \times M')(F'')$, we obtain automaton $(x_1'', \ldots, x_p'') : P \,\&\, P'$.*

Proof. Based on the definition for cross product, for $M \times M'$ to be defined, the same variables in P and P' have to have the same number systems assigned in P and P'. But that is exactly the same condition that needs to hold for number system annotations in $P \,\&\, P'$ to be consistent (in the sense defined in the last bullet in Page 8).

Let $t = (w_1, \ldots, w_m)$ and $t' = (w_1', \ldots, w_n')$ such that $w_i \in \Sigma_{\mathbf{S_i}}^*$ and $w_j' \in \Sigma_{\mathbf{S_j'}}^*$ where $|w_i|$ and $|w_j'|$ are all equal and $w_i = w_j'$ whenever $x_i = x_j'$. Let $t'' = (w_1'', \ldots, w_p'')$ such that $w_k'' = w_i$ or $w_k'' = w_j'$ depending on whether $x_k'' = x_i$ or $x_k'' = x_j'$.

We have the following equivalent statements:

1. $(M \times M')(F'')$ accepts t''.

2. There is a path from (q_0, q'_0) to $(q, q') \in F''$ in $M \times M'$ reading t''.
3. There is a path from q_0 to q in M reading t, and there is a path from q'_0 to q' in M' reading t'.
4. M accepts t and M' accepts t'.
5. P is satisfied by substituting $x_i = [w_i]_{\mathbf{S_i}}$ for all i, and P' is satisfied by substituting $x'_j = [w'_j]_{\mathbf{S'_j}}$ for all j.
6. $P \,\&\, P'$ is satisfied by substituting $x''_k = [w''_k]_{\mathbf{S''_k}}$.

□

Obviously both the construction of cross product and minimizing automata can be carried out using algorithmic procedures. Therefore Theorem 6 gives us a procedure for constructing the automaton for conjunction.

With proper definitions for F'', we have similar theorems for $P \otimes P'$ when $\otimes$ is any other binary logical operator.

Let us construct the automaton $(a, b) : a = 1 \,\&\, b = 2$ from $(a) : a = 1$ in Figure 4.1 and $(b) : b = 2$ in Figure 4.2.

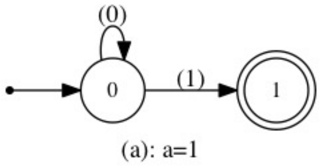


Figure 4.1: Automaton $(a) : a = 1$

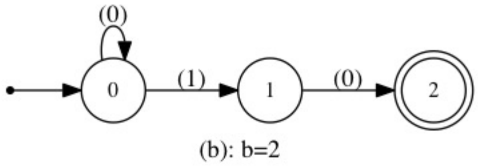


Figure 4.2: Automaton $(b) : b = 2$

Recall that transitions not depicted are transitions to a dead state. The cross product operation is depicted below:

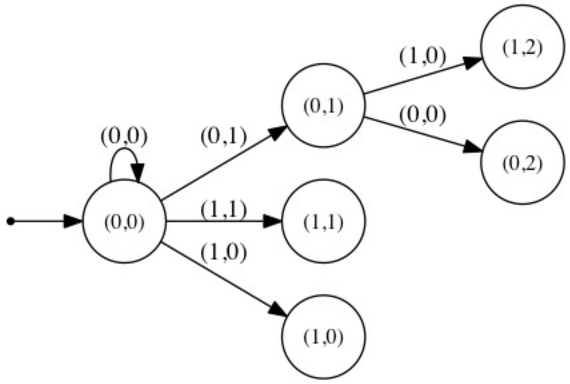


Figure 4.3: Cross product $(a) : a = 1 \times (b) : b = 2$

Making $(1, 2)$ a final state, minimizing, and renaming the states, we get the automaton in Figure 4.4.

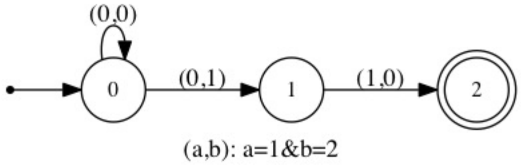


Figure 4.4: Automaton $(a, b) : a = 1 \,\&\, b = 2$

4.2 Quantification

In this section we learn how to construct an automaton $(x_1,\dots,x_{i-1},x_{i+1},\dots,x_m) : \mathrm{E}x_i\ P$ from automaton $(x_1,\dots,x_m) : P$. Let $M\big(Q,q_0,F,\delta,\mathbf{S_1},\dots,\mathbf{S_m}\big)$ be the automaton $(x_1,\dots,x_m) : P$ and let P' be the predicate $\mathrm{E}x_i\ P$. We first construct the nondeterministic automaton $E(M,i)$

$$\big(Q,q_0,F,\delta',\mathbf{S_1},\dots,\mathbf{S_{i-1}},\mathbf{S_{i+1}},\dots,\mathbf{S_m}\big)$$

from M by eliminating the i'th input (coordinate) on all transitions, i.e., letting

$$\delta'\big(q,(\alpha_1,\dots,\alpha_{i-1},\alpha_{i+1},\dots,\alpha_m)\big) = \big\{\delta(q,(\alpha_1,\dots,\alpha_{i-1},\alpha_i,\alpha_{i+1},\dots,\alpha_m)) : \text{ for all } \alpha_i \in \Sigma_{S_i}\big\}.$$

For example, letting M be the automaton $(a,b) : a = 1\,\&\,b = 2$ depicted in Figure 4.4, the automaton $E(M,2)$ is depicted as follows:

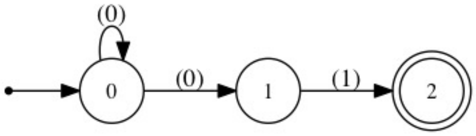


Figure 4.5: Non-deterministic automaton $E(M,2)$

By the definition of transition function of $E(M,i)$, i.e., δ', it is easy to see that if M accepts $(w_1,\dots,w_{i-1},w_i,w_{i+1},\dots,w_m)$, then $E(M,i)$ accepts

$$(w_1,\dots,w_{i-1},w_{i+1},\dots,w_m).$$

However, there might be $t = (w_1,\dots,w_{i-1},w_{i+1},\dots,w_m)$ where the $|w_j|$ are equal for all $j \neq i$ and substitutions $x_j = [w_j]_{\mathbf{S_j}}$ for all $j \neq i$, satisfies P' but $E(M,i)$ does not accept t. In other words, there are cases where $E(M,i)$ does not accept P'.

In our example M accepts $(0^n1, 0^{n-1}10)$ for all $n \geq 1$, and as it is clear $E(M,2)$ accepts (0^n1) for all $n \geq 1$. However $E(M,2)$ does not accept (1), whereas (1) should be accepted by any automaton accepting $\mathrm{E}b\ (a = 1\,\&\,b = 2)$.

Therefore, we have to do more work on $E(M,i)$, to get to an automaton for P'. However as we will see in Lemma 1, the automaton $E(M,i)$ might only miss an insignificant portion of accepted tuples of an automaton accepting P'. These insignificant tuples missed by $E(M,i)$ are those with leading or trailing zeros. The good news is that with a little bit of technical work, it is possible to revive even these insignificant tuples.

Lemma 1. *Let M,P,P', and i be as in the discussion above, and suppose $t = (w_1,\dots,w_{i-1},w_{i+1},\dots,w_m)$ is some tuple of same length words. If P' is satisfied with substitutions $x_j = [w_j]_{\mathbf{S_j}}$ for $j \neq i$, then there exists a constant $k \geq 0$ and $t_k = (w_{k,1},\dots,w_{k,i-1},w_{k,i+1},\dots,w_{k,m})$ such that for all $j \neq i$ we have $w_{k,j} = 0^k w_j$ or $w_{k,j} = w_j 0^k$ depending on whether $\mathbf{S_j}$ is* **msd** *or* **lsd***, and t_k is accepted by $E(M,i)$. It is also the case that whenever t_k for any $k \geq 0$, with the appropriate substitutions, is satisfying P', then t is also satisfying P'.*

Proof. Substitutions $x_j = [w_j]_{\mathbf{S_j}}$ for $j \neq i$ satisfying predicate $P' := \mathrm{E}x_i\ P$ means that there exists a natural number v, such that the substitutions above together with $x_i = v$ is satisfying the predicate P. By definition of number systems, there exists a word w_i such that $v = [w_i]_{\mathbf{S_i}}$. Also by definition of number systems for any integer y and word w, if we have $y = [w]_{\mathbf{S}}$, then either $y = [0^k w]_{\mathbf{S}}$ for all $k \geq 0$ or $y = [w0^k]_{\mathbf{S}}$ for all $k \geq 0$ depending on whether **S** is **msd** or **lsd**. Therefore there exists an integer k such that $w_{k,i}$ is either $0^k w_i$ or $w_i 0^k$ depending on whether $\mathbf{S_j}$ is **msd** or **lsd** and $v = [w_{k,i}]_{\mathbf{S_i}}$ and $|w_{k,i}| = |w_j| + k$ for all $j \neq i$. Therefore $(w_{k,1},\dots,w_{k,i-1},w_{k,i},w_{k,i+1},\dots,w_{k,m})$ is accepted by M where for all j we have $w_{k,j} = 0^k w_j$ or $w_{k,j} = w_j 0^k$ depending on whether $\mathbf{S_j}$ is **msd** or **lsd**. Now by definition of $E(M,i)$, we know that $t_k = (w_{k,1},\dots,w_{k,i-1},w_{k,i+1},\dots,w_{k,m})$ is accepted by $E(M,i)$. This completes the first part of the lemma.

The second part follows very easily from the same properties of number systems mentioned in the proof of the first part of the lemma. □

Based on Lemma 1, to get $(x_1,\dots,x_{i-1},x_{i+1},\dots,x_m) : \mathrm{E}x_i\ P$ we just have to construct an automaton from $E(M,i)$ such that whenever t_k for any $k \geq 0$ is accepted, t is also accepted. For the case where all $\mathbf{S_j}$ for $j \neq i$ are either all **msd** or all **lsd**, we can come up with an easy algorithm to revive tuples t from t_k. In case of all **msd**, let I be the set of all states in $E(M,i)$ reachable from the initial state by reading $(0,\ldots,0)^*$, or in case of all **lsd**, let F' be the set of all states reaching to a final state by reading $(0,\ldots,0)^*$. We can compute I or F' using breadth-first search. In the case of **msd** the nondeterministic automaton[8]

$$(Q, I, F, \delta', \mathbf{S_1}, \ldots, \mathbf{S_{i-1}}, \mathbf{S_{i+1}}, \ldots, \mathbf{S_n})$$

and in the case of **lsd** the nondeterministic automaton

$$(Q, q_0, F', \delta', \mathbf{S_1}, \ldots, \mathbf{S_{i-1}}, \mathbf{S_{i+1}}, \ldots, \mathbf{S_n})$$

is equivalent to $(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_m) : \mathrm{E}x_i\ P$. Determinizing and minimizing this automaton gives us $(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_m) : \mathrm{E}x_i\ P$.

In Figure 4.5, the variable a is over **msd_2**. So the set I is $\{0,1\}$, therefore the following nondeterministic automaton accepts Eb $a = 1\ \&\ b = 2$:

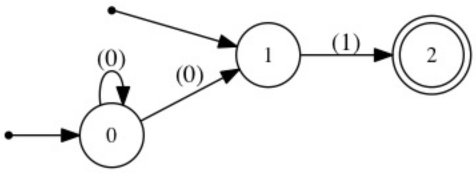


Figure 4.6: Non-deterministic automaton accepting Eb $a = 1\ \&\ b = 2$

Now determinizing and minimizing this automaton gives us $(a) : \mathrm{E}b\ a = 1\ \&\ b = 2$:

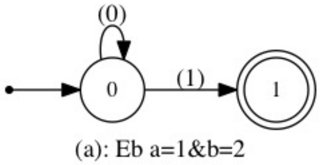


Figure 4.7: $(a) : \mathrm{E}b\ a = 1\ \&\ b = 2$

Currently if for $(x_1,\ldots,x_n) : P$ it is not the case that for all $j \neq i$ number systems $\mathbf{S_j}$ are all **msd** or all **lsd**, then Walnut only constructs $E(M,i)$ for Ex_i P, which is not theoretically accurate. So the user has to be very cautious when quantifying predicates over mixed **msd** and **lsd** number systems, or in cases where the quantified automaton is non-arithmetic. For a definition of the latter see Section 5.2.

To obtain an automaton for Ax_iP, note its equivalence to $\sim(\mathrm{E}x_i \sim (P))$, where $\sim$ is the logical complement (negation). See the next section to learn about the complement operator.

4.3 Complement and Reverse

To obtain $(x_1,\ldots,x_m) :\sim (P)$ from $(x_1,\ldots,x_m) : P$, one has to add all transitions to dead state (in Walnut, we call this totalizing an automaton), and then switching final and non-final states, but one also has to make sure that the resulting automaton is intersected with the automaton accepting $R_{\mathbf{S_1}} \times \cdots \times R_{\mathbf{S_m}}$ where $\mathbf{S_i}$ is the number system assigned to x_i in (annotated) predicate P. (Recall that $R_{\mathbf{S}}$ is the set of all valid representations in the number system **S**. Also recall that to define and use a number system in Walnut, one has to provide automaton accepting the set of all representations in that number system, therefore automaton accepting $R_{\mathbf{S_1}} \times \cdots \times R_{\mathbf{S_m}}$ could be constructed easily using cross product explained in Section 4.1.)

Take a look at automaton $(a) :$?msd_fib $a = 1$ depicted in Figure 4.8 that accept words representing 1 in **msd_fib**.

Now to obtain $(a) :\sim$ (?msd_fib $a = 1$), we first add the dead state and all the transitions to it:

[8] This is an automaton with multiple initial states. One can show that for every nondeterministic automaton with multiple initial states, there is an equivalent automaton with only one initial state.

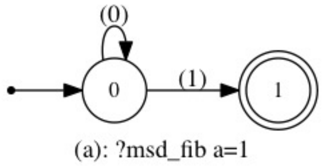


Figure 4.8: Number 1 in Fibonacci

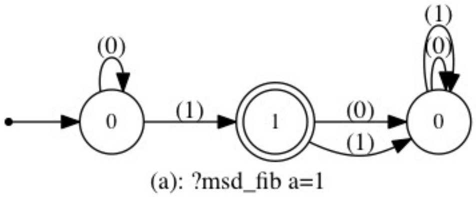


Figure 4.9: Totalized automaton

Switching final and non-final states we obtain an automaton accepting $\{0,1\}^* \setminus 0^*1$:

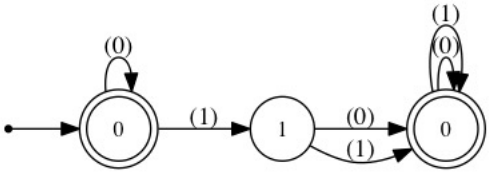


Figure 4.10: Switching final and non-final states

But then this automaton accepts words that have consecutive 1's which are not acceptable Fibonacci representations. So to get the final answer we have to intersect this automaton with the one depicted in Figure 2.6. The result is depicted in 4.11.

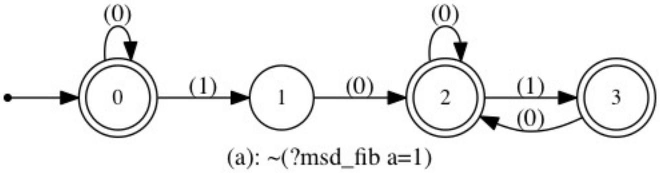


Figure 4.11: Automata accepting all numbers in Fibonacci except 1

The reverse operator is not a logical operator per se, but we include it because it is useful when working with automata. The operand of the reverse operator is an automaton [9]. The result is an automaton with all its transitions reversed.

4.4 Arithmetic and Comparison Operators

Recall that for every number system **S** that we use in Walnut the three automata $(a,b,c):?\mathrm{S}\ a=b+c$, $(a,b):?\mathrm{S}\ a=b$[10], and $(a,b):?\mathrm{S}\ a<b$ are defined. In this section we show that using these three automata and the decision procedure outlined in Sections 4.1–4.3, we can construct automata for more complex relative expressions with lots of arithmetic operators.

[9] Unlike the reverse operator, operands for logical operators are predicates. The reader however understands the very thin and superficial distinction between automata and predicates in this article.

[10] This automaton does not need to be defined explicitly by the user, because we assumed for all number systems **S** in Walnut $a =_{\mathbf{S}} b$ if and only if $a = b$.

For a constant $c > 0$, a natural number, automata (a) : ?S $a = c$ can be constructed recursively using automata (b) : ?S $b = c'$ and (a,b) : ?S $a = b+1$ where c' is the predecessor of c, i.e., $c'+1 = c$. For example, predicate ?S $a = 2$ is equivalent to ?S Eb $a = b+1$ & $b = 1$. Similarly ?S $b = 1$ is equivalent to ?S Eb_2 $b = b_2+1$ & $b_2 = 0$. Based on Definition 2, for all number systems **S**, the automaton for ?S $b_2 = 0$ is the simple automaton accepting 0^*. To construct automaton (a,b) : ?S $a = b+1$, just note that the predicate is equivalent to ?S Ec $a = b+c$ & $c = 1$.

A similar recursive argument can be applied to obtain $(a,b) : a = c*b$ for a constant $c > 0$, i.e., one can construct $(a,b) : (a = b_2 + b)\,\&\,(b_2 = c' * b)$ where c' is the predecessor of c. The similar argument can be applied to obtain automata for division by constants or subtraction.

To construct (a,b) : ?S $a <= b$, note its equivalence to (a,b) : ?S $a < b \mid a = b$. With similar arguments, one can construct automaton for other comparison operators.

It is important to understand Walnut's construction of

$$(y, x_1, x_2, \ldots, x_n) : \text{?S } y \olessthan (x_1 \otimes_1 x_2 \otimes_2 \cdots \otimes_{n-1} x_n)$$

where $n \geq 3$. Here $\olessthan$ denotes an arbitrary comparison operator, and the $\otimes_i$ are arbitrary arithmetic operators. Also let y and x_i be variables or arithmetic constants. All arithmetic operators in Walnut are associative from left to right; see Section 3.4. Based on this, Walnut first transforms the predicate to an equivalent predicate

$$(y, x_1, x_2, \ldots, x_n) : \text{?S E}y_1, \ldots, y_{n-2}\ (y_1 = x_1 \otimes_1 x_2)\,\&\,(y_2 = y_1 \otimes_2 x_3)\,\&\cdots\&\,(y = y_{n-2} \otimes_{n-1} x_n).$$

Now Walnut has all the resources necessary to construct this last automaton.

For example, to construct (a) : 0<=$(a-1+1)$, Walnut first transforms it to (a) : Eb $(b = a-1)\,\&\,(0 = b+1)$. The automaton is depicted below:

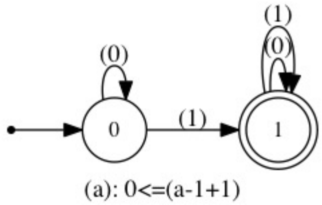


Figure 4.12: Automaton for 0<=$a-1+1$ does not accept 0^*

There is something here that is worth noting. Note how this automaton does not accept 0? In arithmetic over integers $a = 0$ satisfies the predicate. However in Presburger arithmetic setting $a = 0$ gives $b = -1$, which is not acceptable, since Presburger arithmetic is defined over natural numbers. In order to fix this issue, try to always postpone subtraction and division to the rightmost position in your predicates. For example, writing $(a) : 0 <= (a+1-1)$ results in

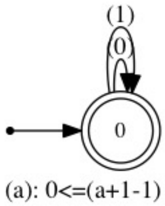


Figure 4.13: Automata for 0<=$a+1-1$ accepts 0^*

4.5 Calling an Automaton

In Section 3.7, we learned about the syntax and semantic of calling an automaton. A calling expression is a kind of syntactic sugar to save some space when writing long and complicated predicates. Suppose we already have computed the automaton $(x_1, x_2, \ldots, x_n) : P$ and given it the name M. We can refer to P in a predicate P' without writing P all over again, by just writing $\$M(e_1, e_2, \ldots, e_n)$, where \$ symbol is to signify that M is an automaton, and the e_i are either arithmetic expressions or predicates with exactly one free variable. In such case, we say, predicate P' is calling M (or is calling predicate P).

To construct automaton for $\$M(e_1, e_2, \ldots, e_n)$, Walnut constructs the equivalent automaton:

$$\mathrm{E} x_1, x_2, \ldots, x_n \; P \,\&\, (x_1 = e'_1) \,\&\, (x_2 = e'_2) \,\&\cdots\&\, (x_n = e'_n) \,\&\, (e_{j_1}) \,\&\, (e_{j_2}) \,\&\ldots\&\, (e_{j_k})$$

where $x_1, x_2, \ldots, x_n$ are the free variables in P, k is the number of predicates in $e_1, e_2, \ldots, e_n$, $j_1, j_2, \ldots, j_k$ are indices of predicates among $e_1, e_2, \ldots, e_n$, and if e_j is an arithmetic expression, then $e'_j = e_j$, otherwise e_j is a predicate, and e'_j is the free variable occurring in e_j.

The fact that this predicate is equivalent to $\$M(e_1, e_2, \ldots, e_n)$ could be obtained easily using the semantic rule explained in Section 3.7. Walnut's implementation includes some considerations to improve efficiency. For example, obviously when e_j is a variable, we do not need to introduce a new variable x_j.

Calling an automaton inside a predicate P' is also more efficient than copying P over and over again in P'. This is because Walnut does not need to construct M every time we write $\$M$ in P'.

The commands def and eval in Walnut are responsible for constructing the automaton M from predicate P. Unlike eval, the command def saves the automaton M so it can be called later from other predicates like P'. See Section 7.2 for more information on def command.

To see an example, let M be the automaton $(a, b) : a + b = 10$, and let Q be the predicate $\$M(x, y) \,\&\, y = 8$. The automaton in Figure 4.14 accepts Q.

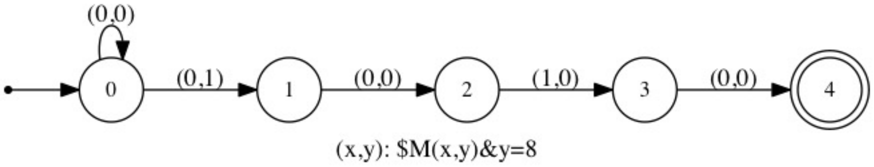


Figure 4.14: Automaton accepting Q

Please refer to Section 7.2, which is devoted to examples of calling automata.

When calling an automaton M, one has to make sure that the j'th argument is in the same number system as j'th input in M for all j.

4.6 Indexing an Automatic Word

Suppose W is an n-dimensional automatic word and $M\big(Q, q_0, O, \delta, \Sigma, \mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_n}\big)$ is its corresponding automaton with output. Also let α be an alphabetic constant. We note that $(x_1, x_2, \ldots, x_n) : W[x_1][x_2]\cdots[x_n] = @\alpha$ is the automaton

$$\big(Q, q_0, F, \delta, \mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_n}\big)$$

when minimized, where $F = \big\{q : O(q) = \alpha\big\}$. Similar arguments can be made for other comparison operators.

Suppose W_1 and W_2 are m- and n-dimensional automatic words, respectively, and let M_1 and M_2 be their corresponding automata with output. We note that $(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n) : W_1[x_1][x_2]\cdots[x_m] = W_2[y_1][y_2]\cdots[y_n]$ is $(M_1 \times M_2)(F)$ when minimized, where F contains all (q_1, q_2) where q_1 and q_2 are states of M_1 and M_2, respectively, and they have the same output. Similar arguments can be made for other comparison operators.

The above statements can be proved easily using the semantic rule explained in Section 3.6. Now what if indices are arithmetic expressions and/or predicates with one free variable? The construction is based on substitutions similar to the ones mentioned for calling expressions in Section 4.5.

5 Special Automata in Walnut

5.1 True and False Automata

In Section 2.4 we saw an example of a predicate with no free variables:

$$\mathrm{A}x\ \mathrm{E}y\ x = 2 * y \mid x = 2 * y + 1$$

This predicate evaluates to true (it is a tautology). Here is an example of a predicate with no free variable that evaluates to false (contradiction):

$$\mathrm{E}x\ x > x + 1$$

Walnut assigns a special automaton called true (false) automaton to predicates with no free variable that evaluate to true (false). However there could be predicates with free variables that are converted to true or false automata. See the following conventions implemented in Walnut:

- Conjunction (disjunction) of true automaton with automaton M yields M (true automaton, respectively).
- Conjunction (disjunction) of false automaton with automaton M yields false automaton (M, respectively).
- Negation of true automaton is false automaton and vice versa.
- Conventions for other logical operators follow from the above.

These conventions are reflecting the following facts from mathematical logic (for a predicate P):

- P & true and P | true are equivalent to P and true respectively.
- P & false and P | false are equivalent to false and P respectively.
- ~ true = false and ~ false = true.

As an example, the automaton $(y) : (\mathrm{A}x\ x < x+1)\ \&\ y = 2$ is exactly the same as automaton $(y) : y = 2$. As another example, the automaton $(y) : (\mathrm{E}x\ x < 0)\ \&\ y = 2$ is the false automaton. As in our last example, note that $\sim (\mathrm{E}x\ x < 0)$ is the true automaton.

Figures 5.1 and 5.2 show the special way Walnut represents true and false automata.

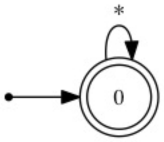


Figure 5.1: True automaton

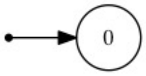


Figure 5.2: False automaton

5.2 Non-arithmetic Automata

There is a need for automata in which inputs (or some of them) do not represent numbers in a specific number system. These automata might accept patterns, or they might be relying on some non-arithmetic instructions.

For example, the pattern 0^*10^* represents powers of 2 in **msd_2**. However, the same pattern represents powers of 2 in **lsd_2**. Therefore, by not assigning a number system to the automaton M accepting the pattern 0^*10^*, we are allowed to call M both in predicates in **msd_2** and in predicates in **lsd_2**. Assigning number systems to automata accepting patterns usually does not make much sense. See more examples in Section 7.3.

Allowing non-arithmetic automata is specially helpful when working with the class of paperfolding words. These words are defined with an automaton that takes two inputs. One input is a number that represents a position in the paperfolding word and the other input is folding instruction that does not represent numbers. To see how Walnut can be used to prove properties of paperfolding words see [3].

6 Installation

Walnut is a command line program. You can run Walnut on any platform as long as you have Java 8 or later (preferably JDK 8 or higher) installed. To see which version of Java is installed on your machine type the following in the terminal (without the command line prompt $):

```
$java −version
```

If you download Walnut as the zipped file Walnut.zip first thing you need to do is to uncompress it. Then open the terminal (or command prompt in Windows), and change the directory to “.../Walnut/bin/”, and run Walnut by typing:

```
$java Main.prover
```

To exit Walnut, type the following command (with the semicolon):

```
exit;
```

To make the distinction that we are typing a Walnut command, names of all Walnut’s commands are written in green. Walnut produces graphical representations of automata among other things. Those representations are files with .gv extensions. In order to open these files you need to install Graphviz, a graph visualization package which is available for all platforms. *All text files that Walnut produces are in the UTF-16 encoding. All text files that Walnut reads have to be in the UTF-16 encoding as well.*

6.1 Eclipse

As explained in the previous section, you can use the terminal to work with Walnut and enter your commands. However, I encourage you to use a Java IDE, like Eclipse, because in my opinion, entering commands in the console of a good IDE is more convenient than doing the same in the terminal. You are only going to run Walnut inside the IDE and use the IDE’s console (not the source code editor) to enter Walnut commands. Here is the instructions on how to run Walnut inside Eclipse for Java:

1. Go to this link and download Eclipse for Java for your specific platform.
2. Downloaded file is probably compressed. To start Eclipse, first uncompress the file, then click on the file named Eclipse.
3. When you run Eclipse, it asks for a workspace address. Feel free to enter the path to your desired directory.

   Now we need to import Walnut into Eclipse as a Java project:
4. When in Eclipse, go to “File > Import ... ”. In the dialog that opens up choose “General > Existing Project into Workspace”.
5. In the dialog that opens up, click browse, find Walnut (that you downloaded and uncompressed in the Installation section) and press open. Then click finish.
6. Close the Welcome page in your Eclipse window.
7. On the "Project Explorer" (probably) on the left of your screen, you can see only one project (the one that you just imported). Click on it. Then click on “src”. Then double click on “prover.java”.
8. You will see a green circle with a white triangle inside it. Click on it. This causes Walnut to run.
9. You can enter your Walnut commands in the console window in your Eclipse. If you are not able to find the console window, go to “Window > Show View > Other > General > Console” to open it.

7 Commands

Every command ends in either a colon or a semicolon. If you want to see the reports on the intermediate steps of a computation use colon, otherwise use semicolon. For example, if we type:

```
eval test "a=b+1":
```

we get an output similar to the following written in the console:

```
a=b+1 has 2 states: 2ms
total computation time: 5ms
```

which explains that the automaton for predicate $a = b + 1$ has 2 states and it took 2 milliseconds to compute it. We use blue to denote predicates. Here we use grey to indicate the output produced by Walnut in the console. We use red to indicate errors in the console.

Whitespace is ignored. You can, for example, span one single command into multiple lines to improve readability. So, for example, you can write the following interchangeably:

```
eval test "a=b+1";
eval test
"a=b+1";
eval test "a
=b+1";
eval test
"a=b+1"
;
```

In case we forget to separate the name test and predicate $a = b + 1$ of the eval command, Walnut catches it by returning an error:

```
eval test"a=b+1":
invalid use of eval/def command
        : eval test"a=b+1":
```

Here is the full list of commands in Walnut and we will go over them one by one in detail:

- exit
- eval <name> <predicate>
- def <name> <predicate>
- reg <name> <number system> <regular expression>
- reg <name> <alphabet> <regular expression>
- load <file name>

7.1 eval: eval <name> <predicate>

This is the most important command in Walnut and it stands for "evaluate." This command takes two arguments. The first argument is a name for the evaluation. Name of the evaluation starts with a letter and could contain alphanumerics and underscore. The files generated as the result of the eval command, all share the name given in the first argument. The second argument is a predicate that we want to evaluate. Predicates are always placed between quotation marks. To see the definition for predicates see Section 3.9. In this article we typeset predicates in math mode in LaTeX. However, the reader should note that this typesetting is different from the one they see in the terminal. Let us see an example:

```
eval four "a=4":
a=4 has 4 states: 3ms
total computation time: 3ms
```

This evaluates to an automaton with one binary input labeled a. This is the automaton $(a) : a = 4$. To learn about the notation $(a) : a = 4$ see Section 2.4. The automaton accepts only if a is the most-significant-digit-first binary representation of 4, i.e., if it belongs to 0^*100. This automaton is drawn and saved in the directory "/Walnut/Result/" in a file named four.gv as shown in Figure 7.1. The graph drawing software Graphviz is required to open this file; see Section 6.

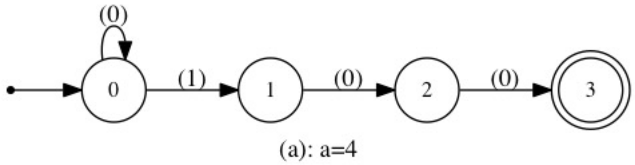


Figure 7.1: Content of the file four.gv

How does Walnut know to use the most-significant-digit-first binary system? Walnut defaults to **msd_2** which is how we show the most-significant-digit-first binary system in Walnut; see Section 2.2 to learn about this notation and to learn about number systems in general. To explicitly mention **msd_2** type:

```
eval four "?msd_2 a=4";
```

Similarly, for the least-significant-digit-first binary type:

```
eval lsd_four "?lsd_2 a=4";
```

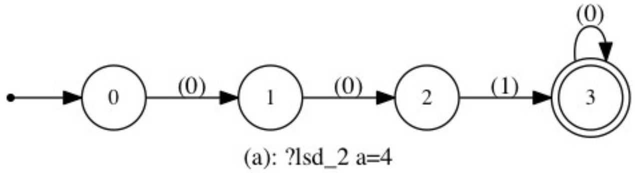


Figure 7.2: lsd_four.gv

Here is another example, this time in **lsd_3**:

```
eval ternary_example "?lsd_3 a<5";
```

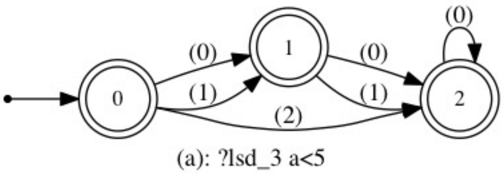


Figure 7.3: ternary_example.gv

This automaton accepts exactly those words representing the numbers 0,1,2,3,and 4 in the least-significant-digit ternary base, i.e., $0^*, 10^*, 20^*, 010^*, 110^*$ respectively. Note the trailing zeros in the representations as opposed to the leading zeros in a most-significant-digit-first (**msd**) number system. Also note that this automaton accepts the empty word. This is because in the definition of number systems we agreed that the empty word represents 0.

Let us see an example of an automaton with 2 inputs:

```
eval two_inputs "b=a+1";
```

This constructs the automaton $(a, b) : b = a+1$ in which the first input corresponds to a, and the second input corresponds to b. Recall from Section 2.4 that Walnut uses lexicographic ordering on the name of variables when constructing automata. So, for example, even though the first variable that appears in $b = a+1$ is b, it corresponds to the second input in the automaton.

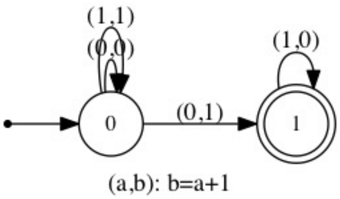


Figure 7.4: two_inputs.gv

Walnut generates two other files as the outcomes of the eval command which can also be found in the directory “/Walnut/Result/”. For the evaluation two_inputs, these two files are named two_inputs_log.txt and two_inputs.txt and they are both text files.

The file two_inputs_log.txt contains the details of the evaluation including the intermediate steps and the time each of those steps took to complete. In our example, there are not many intermediate steps involved:

```
b=2 has 3 states: 0ms
total computation time: 0ms
```

File 1: two_inputs_log.txt

The file two_inputs.txt contains the definition of the automaton in Figure 7.4:

```
1  msd_2 msd_2
2  0 0
3  0 0 -> 0
4  0 1 -> 1
5  1 1 -> 0
6  1 1
7  1 0 -> 1
```

File 2: two_inputs.txt

Line 1 indicates that the first and the second inputs of the automaton are both in **msd_2**. The two states 0 and 1 in Figure 7.4 are declared in Lines 2 and 6. The first zero in Line 2 refers to the state 0 and the second zero refers to its output. Likewise, the first one in Line 6 refers to the state 1 and the second one refers to its output. Note that the automaton for evaluation two_inputs is not an automaton with output, however all automata are stored as automata with outputs in Walnut; see Section 2.1. For an ordinary automaton, states with non-zero outputs are interpreted as final states, and states with zero outputs are interpreted as non-final states. So here state 0 is non-final, whereas state 1 is final. Transitions for states 0 and 1 are declared in Lines 3-5 and 7 respectively. For example, state 0 on $(0,0)$ transitions to itself, and on $(0,1)$ transitions to state 1. Transitions not depicted are transitions to the dead state. For example, state 1 transitions to the dead state on every tuple except $(1,0)$. To learn more about definition of an automaton in text files and how to manually define automata in text files see Section 8.1.

In Section 2.3, we talked about the Thue-Morse word. The Thue-Morse word's corresponding automaton with output, depicted in Figure 2.7, is defined in directory “/Walnut/Word Automata Library/” in a file named T.txt. We can refer to the Thue-Morse word in predicates by typing T. See Section 8.2 on how to define new automatic words in Walnut.

We talked about square subwords in the Thue-Morse word. The following predicate is satisfied by (i,n) if $T[i..i+n-1] = T[i+n..i+2n-1]$, i.e., if there exists a square subword of length $2n$ starting at position i.

```
eval squares_in_thue_morse_word "n>0&(Ak k<n=>T[i+k]=T[i+n+k])";
```

The order of a square is half its length. Now if we want to find all natural numbers n for which there exists a square of order n in the Thue-Morse word, we simply use the existential quantifier E:

```
eval order_of_squares_in_thue_morse_word "Ei n>0&(Ak k<n=>T[i+k]=T[i+n+k])";
```

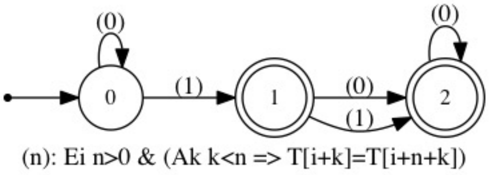


Figure 7.5: order_of_squares_in_thue_morse_word.gv

Based on this automaton, the natural number n with **msd_2** representation of the form $0^*(1|11)0^*$ is an order of a square in the Thue-Morse word. In other words, the set of orders in the Thue-Morse word is

$$\{n : \text{there exists } k \geq 0 \text{ such that } n = 2^k \text{ or } n = 2^{k+1} + 2^k\}$$

Overlaps are the words of the form $axaxa$ where a is a symbol and x is any word, e.g., the word “alfalfa” in English is an overlap. It is a known that the Thue-Morse word avoids overlaps. How do we make sure, using Walnut, that the Thue-Morse word does not have any overlaps? The result of the following predicate must be the true automaton; see Section 5.1, if the Thue-Morse does not have any overlaps:

```
eval thue_morse_does_not_have_overlaps "~(Ei,n n>0&(Ak k<=n=>T[i+k] = T[i+n+k]))":
n>0 has 2 states: 1ms
  k<=n has 2 states: 1ms
    T[(i+k)] = T[((i+n)+k)] has 12 states: 6ms
      (k<=n=>T[(i+k)] = T[((i+n)+k)]) has 25 states: 1ms
        (Ak(k<=n=>T[(i+k)] = T[((i+n)+k)])) has 1 states: 27ms
          (n>0&(Ak(k<=n=>T[(i+k)] = T[((i+n)+k)]))) has 1 states: 0ms
            (Ei,n(n>0&(Ak(k<=n=>T[(i+k)] = T[((i+n)+k)])))) has 1 states: 1ms
              ~(Ei,n(n>0&(Ak(k<=n=>T[(i+k)] = T[((i+n)+k)])))) has 1 states: 0ms
total computation time: 38ms
```

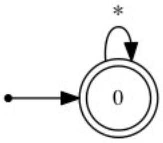


Figure 7.6: thue_morse_does_not_have_overlaps.gv

The automaton in Figure 7.6 is the true automaton. For more information see Section 5.1.

Note that if a predicate is not valid Walnut returns an error:

```
eval invalid "x+y+z";
the final result of the evaluation is not of type automaton
        : eval invalid "x+y+z";
```

To understand why this is not a valid predicate see Section 3.5. In the following examples note how Walnut points to the locations of the errors in the predicates. By saying "char at n", Walnut tries to convey that there is something wrong going on at the vicinity of the n'th character in the predicate.

```
eval invalid2 "(x+y+z=0";
unbalanced parenthesis
        : char at 0
        : eval invalid2 "(x+y+z=0";

eval invalid3 "(~x)=0";
```

```
operator ~ cannot be applied to the operand x of type variable
: char at 1
: eval invalid3 "(~ x) = 0";

eval invalid4 "T[i + j] = i - 1";
operator = cannot be applied to operands T[(i + j)] and (i - 1) of types word and arithmetic
    respectively
        : char at 6
        : eval invalid4 "T[i + j] = i - 1";

eval invalid5 "T[2] = 1";
operator = cannot be applied to operands T[2] and 1 of types word and number literal
    respectively
        : char at 4
        : eval invalid5 "T[2] = 1";
```

The last example can be fixed as follows:

```
eval fixed5 "T[2] = @1";
```

To understand why see Section 3.8.

The last thing to note about the eval command is that Walnut overrides the files generated by an evaluation if the name of the evaluation is used in a new evaluation.

7.2 def: def <name> <predicate>

The word def stands for define. The syntax for this command is exactly the same as the syntax for eval command. The only difference between this command and eval is that the automaton constructed is saved in the directory "/Walnut/Automata Library/" for later use. Suppose we write the following:

```
def sum10 "x + y = 10";
```

This creates as usual the files sum10.gv, sum10.txt, and sum10_log.txt in the directory "/Walnut/Result/". However, it also saves a copy of sum10.txt in the directory "/Walnut/Automata Library/". Any automaton saved in this directory can be called in other predicates by referring to its name and the special character $. To learn about calling see Sections 3.7 and 4.5.

Let us see examples of predicates calling the automaton sum10:

```
eval lessThanThree "Ea a>=8 & $sum10(b, a)";
```

This predicate is satisfied by numbers b for which there exist an $a >= 8$ such that $b + a = 8$, i.e., $0, 1, 2$:

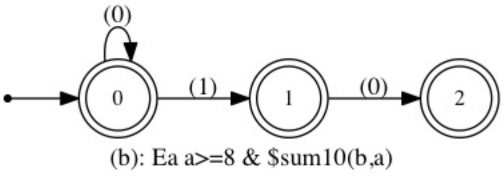


Figure 7.7: lessThanThree.gv

We can send the same variable to both arguments of sum10:

```
eval five "$sum10(a, a)";
```

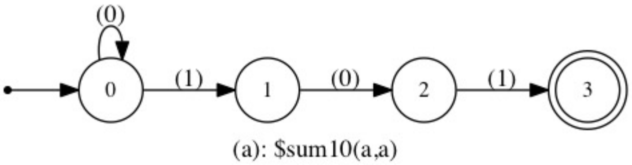


Figure 7.8: five.gv

We can send constants to any arguments of sum10:

```
eval three "$sum10(7,a)";
```

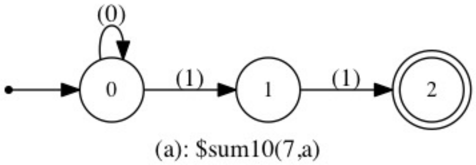


Figure 7.9: three.gv

Indeed, we can send any arithmetic expressions or predicates with one free variable to arguments:

```
eval three "$sum10(a-2,3*a)";
```

```
eval three "Eb $sum10(a,b+3=10)";
```

The resulting automaton for both of these is depicted in Figure 7.9. We can call sum10 to define new automata:

```
def threeSum10 "$sum10(x+y,z)";
```

Now we can write

```
eval three "Ey,z $threeSum10(x,y,z) & y=2 & z=5";
```

The result of this evaluation is again depicted in Figure 7.9.

Now look at the following example:

```
eval nonsense "$sum10(a=b,4)";
argument 1 of function sum10 cannot be an automaton with != 1 inputs
        : char at 1
        : eval nonsense "$sum10(a=b,4)";
```

This is because the first argument is a predicate with two free variables.

We cannot send a variable in **lsd_2** to an automaton that accepts only **msd_2**, and expect getting anything interesting in return. The following example would run fine, but the result is another nonsense:

```
eval another_nonsense "?lsd_2 $sum10(x,4)";
```

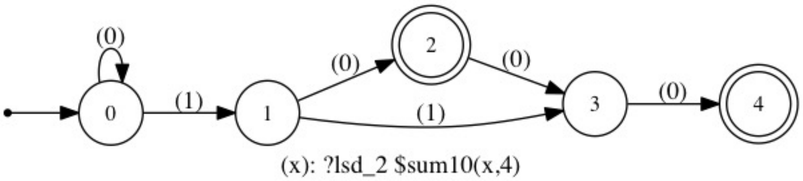


Figure 7.10: another_nonsense.gv

The number of arguments when calling an automaton should match the number of inputs of that automaton:

```
eval invalid "$sum10(x,y,z)";
function sum10 requires 2 arguments
        : char at 1
        : eval invalid "$sum10(x,y,z)";
```

Always remember the roles of the inputs to an automaton created by the def command. For example, look at the following examples of the def command:

```
def f1 "y<x";
def f2 "x<y";
```

Now the following evaluates to an automaton accepting representations of numbers > 1:

```
eval greater_than_1 "$f1(a,1)";
```

whereas the following evaluates to an automaton accepting representations of numbers < 1:

```
eval less_than_1 "$f2(a,1)";
```

This is because f1 is an automaton for which the first argument is greater than the second argument, whereas, f2 is an automaton for which the first argument is less than the second argument. *Always remember that Walnut sorts inputs of an automaton based on their labels' lexicographic order.*

7.3 reg

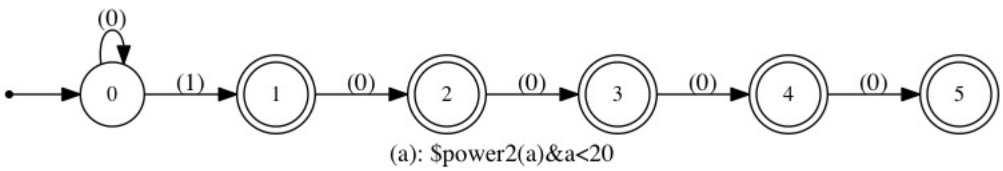
The word reg stands for regular expression. Before we talk about this command in detail, let us motivate the need for it through an example. Suppose we need an automaton accepting **msd_2** representations of powers of 2 that are less than 20. There is no straightforward way of constructing an automaton accepting representations of powers of 2 using eval and def commands[11]. Remember how def command saves automata definition in directory "/Walnut/Automata Library/"? We can manually create a file power2.txt in this directory and write in it the definition of an automaton accepting binary representations of powers of 2:

```
msd_2
0 0
0 -> 0
1 -> 1
1 1
0 -> 1
```

File 3: power2.txt

See Section 8.1 to learn the syntaxes of defining an automaton in a text file. Now we can write a predicate for powers of 2 that are less than 20:

```
eval power2LessThan20 "$power2(a) & a < 20";
```



(a): $power2(a)&a<20

Figure 7.11: power2LessThan20.gv

The better approach to this problem is to use the reg command. This command can be used in two different ways:

[11] The set of powers of 2 is not expressible in Presburger arithmetic. However the extended Presburger arithmetic that involves automatic words is powerful enough to express this set (why?)

1. reg <name> <number system> <regular expression>
2. reg <name> <alphabet> <regular expression>

To construct an automaton that accepts **msd_2** representations of powers of 2 we can use the first syntax:

```
reg power2 msd_2 "0*10*";
```

Similar to eval and def command, the second argument is a name. The third argument is a number system, and the last argument is a regular expression. This will construct an automaton for the regular expression, and saves the result in a file named power2.txt in "/Walnut/Automata Library/", in addition to saving, as usual, the drawing of the automata in power2.gv in directory "/Walnut/Result/". The file power2.txt is exactly the same as File 3.

Note that 0^*10^* is also the **lsd_2** representations of powers of 2. For this reason, there needs to be a way of defining an automaton from a regular expression that is not restricted to a particular number system. We call such an automaton a non-arithmetic automaton; see Section 5.2. To create a non-arithmetic automaton accepting a pattern we can use the second version of the reg command in which instead of a number system we specify an alphabet:

```
reg general_power2 {0,1} "0*10*";
```

The file general_power2.txt generated by this command is the following:

```
{0,1}
0 0
0 -> 0
1 -> 1
1 1
0 -> 1
```

File 4: general_power2.txt

The only difference between Files 3 and 4 is the first line; see Section 8.1 for more information.

Since general_power2 is not restricted to a particular number system both of the following are valid:

```
eval power2Less20_msd "?msd_2 $general_power2(a) & a < 20";
eval power2Less20_lsd "?lsd_2 $general_power2(a) & a < 20";
```

Note that 0^*10^* is also **msd_n** and **lsd_n** representations of powers of n for any $n > 1$. So what if we write the following:

```
eval invalid "?msd_3 $general_power2(a) & a < 20";
in computing cross product of two automata, variables with the same label must have the same
    alphabet
        : char at 12
        : eval power3_less10_msd "?msd_3 $general_power2(a) & a < 20";
```

Here Walnut is complaining about the fact that **msd_3**'s alphabet is $\{0,1,2\}$, whereas general_power2's input alphabet is $\{0,1\}$; see File 4. Walnut is very strict about matching alphabets, which we understand is sometimes a drawback, for example in the above example. We will improve this feature in future releases of Walnut.

We use the automata library in [2] for converting regular expressions to automata. To see the syntax for regular expressions refer to this website.

Here is a summary of the important syntax:

The alphabet in the second version of reg command could only be a subset of $\{0,1,\ldots,9\}$. Therefore the following is not allowed:

```
reg invalid {0,-1,-2} "-20*";
the input alphabet of an automaton generated from a regular expression must be a subset of
    {0,1,...,9}
        : reg invalid {0,-1,-2} "-20*";
```

The last thing to note about the reg command is that for any regular expression r, the resulting automaton from reg command is the intersection of the automaton for r with Σ^* where Σ is the alphabet given as the third argument of the reg command. For example:

$*$	zero or more occurrences of an expression
$+$	one or more occurrences of an expression
|	union, e.g., (0 | 1)2*
.	any single character, e.g., 2.*
[]	character class, e.g., [1 – 4] means any of 1, 2, 3, 4
$\wedge$	complement of a character class, e.g., [^2 – 9] is any of 0, 1

Table 7.1: syntax summary for regular expressions

```
reg note_the_intersection {2,3} "2.*2";
```

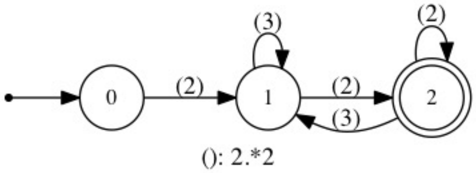


Figure 7.12: note_the_intersection.gv

7.4 load: load <file name>

We can write any series of legitimate Walnut commands in a text file and save it in the directory "/Walnut/Command Files/". Then we can load it by writing the following in Walnut:

```
load file_name.txt;
```

This runs all commands in file_name.txt in the order they appear. Recall that the file's encoding must be UTF-16.

8 Working with Input/Output

Throughout this section it is assumed that all files have UTF-16 encoding.

8.1 Defining Automata in Text Files

In this section we learn how to manually define all automata types in text files. Recall that an ordinary automaton can be thought of as an automaton with output, in which states with non-zero outputs are treated as final states; see Section 2.1. Therefore suppose $M\big(Q, q_0, O, \delta, \Sigma, \mathbf{S_1}, \mathbf{S_2}, \dots, \mathbf{S_n}\big)$ is an automaton with output with m states and n inputs over number systems $\mathbf{S_i}$. Furthermore suppose that the states are labeled 0 to $m-1$, i.e., $Q = \big\{0, 1, \dots, m-1\big\}$, and that $q_0 = 0$[12]. To define M in a text file, first create a text file M.txt[13]. The first line must be

$$S_1\ S_2\ \cdots\ S_n$$

which declares inputs' number systems. The second line is declaring state 0 as follows:

$$0\ \alpha$$

where $\alpha = O(0)$. Next lines are declarations of transitions of state 0 which can come in any order. For every $\alpha_1 \in \Sigma_{\mathbf{S_1}}, \alpha_2 \in \Sigma_{\mathbf{S_2}}, \dots, \alpha_n \in \Sigma_{\mathbf{S_n}}$ transitions are of the following form

$$\alpha_1\ \alpha_2\ \cdots\ \alpha_n \text{ -> } q$$

[12] If an automaton does not follow these criteria we can always come up with an isomorphic one that does.

[13] File names in Walnut start with letters and can contain alphanumerics and underscore.

whenever $\delta\big(0,\alpha_1,\alpha_2,\dots,\alpha_n\big) = q$. There is no need to declare transitions to a dead state. For any pair $(\alpha_1,\alpha_2,\dots,\alpha_n)$ that no declaration of the form above is mentioned, it is assumed that $\delta(0,\alpha_1,\alpha_2,\dots,\alpha_n)$ is a dead state. We can use $*$, the wildcard matching symbol, in place of any symbol α_i. If there is a transition of the form

$$\alpha_1\ \alpha_2\ \dots\ \alpha_{i-1}\ *\ \alpha_{i+1}\ \dots\ \alpha_n\,\text{->}\,q$$

it is understood that $\delta\big(0,(\alpha_1,\alpha_2,\dots,\alpha_{i-1},\beta,\alpha_{i+1},\dots,\alpha_n)\big) = q$ for every $\beta \in \Sigma_{\mathbf{S_i}}$. After transitions of the state 0 are declared, we declare state 1 followed by its transitions. We continue like this until all states and their transitions are declared. Note that nowhere in M.txt we are defining the output alphabet Σ. The output alphabet is inferred indirectly by looking at the state declarations. To see examples refer to Files 2–4.

A non-arithmetic automaton is defined in the same way, except that in the first line, for inputs that do not have number systems associated with them, we write down the alphabet between curly brackets. Alphabets can be any subset of integers. As an example see File 4.

Defining true or false automata in text files is easy. They have only one line and it is either true or false.

As one last example, the paperfolding words are given by the following automaton; see Section 5.2 and article [3] for more details:

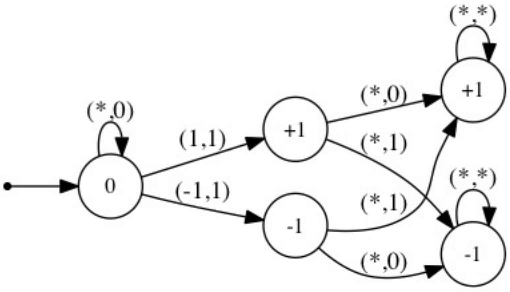


Figure 8.1: Automata for paperfolding words

This automaton is defined in the file PF.txt in directory "/Walnut/Word Automata Library/":

```
{−1,1} lsd_2

0 0
* 0 −> 0
1 1 −> 1
−1 1 −> 2

1 1
* 1 −> 4
* 0 −> 3

2 −1
* 0 −> 4
* 1 −> 3

3 1
* * −> 3

4 −1
* * −> 4
```

File 5: PF.txt

8.2 Defining New Automatic Words

The eval and def commands always produce automata accepting a predicate, therefore the result is never an automaton with output. So to define an automatic word W, we need to manually define its corresponding automaton with output in the directory "/Walnut/Word Automata Library/". For example, the Thue-Morse word is defined in the file "/Walnut/Word Automata Library/T.txt" as follows:

```
msd_2
0 0
0 -> 0
1 -> 1
1 1
0 -> 1
1 -> 0
```

File 6: T.txt

8.3 Defining New Number Systems

Based on Definition 3, to define a new number system **S**, we need to define automata for $R_{\mathbf{S}}$, $+_{\mathbf{S}}$, and $<_{\mathbf{S}}$. We do not need to define an automaton for $=_{\mathbf{S}}$, because it can be generated easily, since we assumed that $w_1 =_{\mathbf{S}} w_2$ if and only if $w_1 = w_2$ for any two words w_1 and w_2 of the same length. The automata for number systems must be defined in the directory "/Walnut/Custom Bases/". For example, for the number system **S**, assuming it is **msd**, one needs to create msd_S.txt, msd_S_addition.txt, and msd_S_less_than.txt for $R_{\mathbf{S}}$, $+_{\mathbf{S}}$, and $<_{\mathbf{S}}$ respectively. If **S** is **lsd**, file names must be lsd_S.txt, lsd_S_addition.txt, and lsd_S_less_than.txt respectively. The number system **S** defined in this way can be used in predicates by typing ?msd_S or ?lsd_S depending on whether **S** is **msd** or **lsd**. If the automaton for $<_{\mathbf{S}}$ is not defined by the user, Walnut assumes that $<_{\mathbf{S}}$ is the lexicographic ordering, i.e., if w_1 and w_2 are of the same length, then $w_1 <_{\mathbf{S}} w_2$ if and only if w_1 comes before w_2 in lexicographic order[14]. If the automata for $R_{\mathbf{S}}$ is not given, then $R_{\mathbf{S}}$ is assumed to be $\Sigma_{\mathbf{S}}^*$. The alphabet $\Sigma_{\mathbf{S}}$ is inferred from the automaton for $+_{\mathbf{S}}$ which is always given.

Note that reversing all automata for **msd_n** we get the corresponding automata for **lsd_n**. The same goes with **msd_fib** and **lsd_fib**. Thus for a number system **msd_S** if we only define files for **msd_S**, but then typing ?lsd_S in a predicate, Walnut automatically creates automata for **lsd_S** by reversing those of **msd_S** and vice versa. However the user should be cautious since there could very well be number systems for which the difference between **msd** and **lsd** is more than the direction of the arrows in their corresponding automata.

8.4 Converting .gv files to .jpeg

The drawings of automata in Walnut are stored in .gv files. Not only can the software Graphviz open the files with this extension, but it can also convert them to many different file formats. For example, suppose you have a file named automaton.gv. To convert it to automaton.jpeg type the following in the terminal:

```
$dot -Tjpg automaton.gv -o automaton.jpeg
```

See Graphviz to learn how to convert .gv files to other file types.

References

[1] J. R. Büchi. "On a Decision Method in Restricted Second Order Arithmetic". In: *Logic, Methodology and Philosophy of ScienceProceeding of the 1960 International Congress*. Studies in Logic and the Foundations of Mathematics 44 (1966). Ed. by Patrick Suppes Ernest Nagel and Alfred Tarski, pp. 1–11.

[2] Anders Møller. *dk.brics.automaton – Finite-State Automata and Regular Expressions for Java*. http://www.brics.dk/automaton/. 2010.

[14] Lexicographic ordering on symbols is assumed to be $\cdots < -2 < -1 < 0 < 1 < 2 < \cdots$.

[3] D. Goč, H. Mousavi, L. Schaeffer, and J. Shallit. “A New Approach to the Paperfolding Sequences”. In: *Lecture Notes in Computer Science* 9136 (2015). Ed. by A. Beckmann, V. Mitrana, and M. Soskova. CiE 2015, pp. 34–43.

[4] H. Mousavi and J. Shallit. “Mechanical Proofs of Properties of the Tribonacci Word”. In: *Lecture Notes in Computer Science* 9304 (2015). Ed. by F. Manea and D. Nowotka. WORDS 2015, pp. 170–190.

[5] J.-P. Allouche and J. Shallit. *Automatic Sequences: Theory, Applications, Generalizations*. Vol. 1. CUP, 2003.

[6] E. Charlier, N. Rampersad, and J. Shallit. “Enumeration and Decidable Properties of Automatic Sequences”. In: *International Journal of Foundations of Computer Science* (2012), pp. 1035–1066.

[7] D. Goc. “Automatic Sequences and Decidable Properties: Implementation and Applications”. Master’s thesis. University of Waterloo, 2013.

[8] H. Mousavi, L. Schaeffer, and J. Shallit. “Decision Algorithms for Fibonacci-Automatic Words, I: Basic Results”. RAIRO Inform. Theorique to appear. 2016.

[9] C. F. Du, H. Mousavi, L. Rowland, L. Schaeffer, and J. Shallit. “Decision Algorithms for Fibonacci-Automatic Words, II: Related Sequences and Avoidability”. submitted. 2016.

[10] C. F. Du, H. Mousavi, L. Schaeffer, and J. Shallit. “Decision Algorithms for Fibonacci-Automatic Words, III: Enumeration and Abelian Properties”. submitted. 2016.

[11] L. Schaeffer and J. Shallit. “Trapezoidal, and Balanced Words in Automatic Sequences”. preprint. 2015.

[12] L. Schaeffer. “Deciding Properties of Automatic Sequences”. Master’s thesis. University of Waterloo, 2013.

[13] J. Shallit. “Decidability and Enumeration for Automatic Sequences: a survey”. In: *Lecture Notes in Computer Science* 7913 (2013). Ed. by A. A. Bulatov and A. M. Shur, pp. 49–63.