Proving Properties of Fibonacci Representations via Automata Theory

by

Sonja Linghui Shan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

Author's Declaration

Statement of Contributions

This thesis is based, in part, on joint work with my supervisor Jeffrey Shallit [28]. Parts of Chapter 1, 2.1, 4, 5, 6, 7, 8, and 9 are based, in part, on my joint paper [28], and taken verbatim from that paper.

Abstract

In this work, we introduce a novel framework for mechanically testing the completeness and unambiguity of Fibonacci-based representations via automata theory. We call a representation (or a number system) complete and unambiguous when it provides one and only one representation for each number in the range covered by the representation. Many commonly used representations are complete and unambiguous: consider the familiar binary number system—each natural number has a unique representation up to leading zeros. Additionally, if a representation is complete, we describe an algorithm, of $O(\log n)$ complexity, to find a representation for any particular number n.

Since a number system is a set of valid representations (or strings), it can be seen as a formal language. If the language is regular, then we can test the membership of any given string with a finite automaton. This combined with Büchi's [8] theorem—stating that there exists a decision procedure that, given a first-order logical formula, using addition and indexing into an automatic sequence, will decide the truth or falsity of it—gives the theoretical foundation of our framework. Since each Fibonacci-based representation previously proposed in the literature actually forms a regular language, our framework can therefore provide a unified approach to significantly shorten and simplify their proofs of correctness. In addition to verification, our process can easily check whether a new proposed system is complete and unambiguous. This saves the need to have a long case-based or induction proof accompanying each new system. We propose several new systems, discovered from an exhaustive search of automata with a small number of states, and show a succinct proof of correctness for each following our framework.

Throughout the rest of this thesis, we demonstrate the versatility and efficacy of our framework by testing a diverse set of previously published and newly discovered representations. We consider representations covering different ranges of numbers: just natural numbers or all integers. We also consider representations using different alphabets and representations evaluated with different underlying sequences: Fibonacci numbers with positive or negative or positive and negative indices. For each representation, we analyze its definition to express it as a regular language and a finite automaton, then convert numbers in it to a form that can be processed by the software we use, and finally construct the first-order logical formulas asserting completeness and unambiguity. Different representations call for different proof processes; in particular, the conversion process needs to be customized to fit different alphabets, evaluation equations, and underlying sequences. We explain the different ways of making these adjustments as well as the tradeoffs we face; a larger intermediate automaton could simplify the construction of the first-order logical formulas but might also require a more complicated proof of correctness.

Acknowledgements

I would like to thank my supervisor, Jeffrey Shallit, for his guidance, patience, and encouragement.

I also thank Eric Blais and Barbara Csima for agreeing to read my thesis.

Finally, I would like to thank Arshia Fazeli for his support and the many particle-related jokes.

Dedication

To my mother.

Table of Contents

List of Figures

List of Tables

Chapter 1

Motivation and Preliminaries

Given an increasing sequence $(s_n)_{n\geq 0}$ of positive integers, a numeration system or a representation is a way of expressing natural numbers as a linear combination of the s_n. Many different numeration systems, such as representation in base k, or the more exotic systems based on the Fibonacci numbers, have been proposed. For example, recall that the Fibonacci numbers, sequence A000045 in the *On-Line Encyclopedia of Integer Sequences* (OEIS), are defined by the recurrence $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$ and the initial values $F_0 = 0$, $F_1 = 1$. Consider writing a non-negative integer n as a sum of distinct Fibonacci numbers F_i for $i \geq 2$. Some numbers, such as 12, have only one such representation ($12 = 8+3+1 = F_6+F_4+F_2$), while others have many: $8 = F_6 = F_5+F_4 = F_5+F_3+F_2$.

There are two very desirable characteristics of a numeration system. First, *completeness:* every natural number should have a representation. Second, *unambiguity:* no natural number should have two or more different representations. These two goals are typically achieved by restricting the types of representations that are considered valid within the system. If a system achieves both goals, we say it is *perfect.* For Fibonacci representations, various perfect systems have been proposed.

Among all possible perfect systems based on Fibonacci numbers, one is considered the canonical form: the *Zeckendorf* or *greedy* representation. This representation can be computed as follows: first, choose the largest index i such that $F_i \leq n$. Then the representation for n is F_i plus the (recursively-computed) representation for $n - F_i$. The representation for 0 is the empty sum of 0 Fibonacci numbers. A simple induction now shows that the greedy algorithm produces one and only one representation for every natural number, which is evidently unique.

This representation was originally noted by Zeckendorf, but was first published by

Lekkerkerker [16] and only later by Zeckendorf himself [31]. It was also anticipated, in much more general form, by Ostrowski [20].

An alternative (but equivalent) definition of Zeckendorf representation is to impose a condition that valid representations must obey. For example, the Zeckendorf condition requires that a representation be valid if and only if no two consecutive Fibonacci numbers appear in the sum.

It is convenient to express arbitrary sums of distinct Fibonacci numbers as strings of digits over a finite alphabet (in analogy with base-k representation). Let $x = a_1 \cdots a_t$ be a string (or word) made up of integer digits. We define its value as a Fibonacci representation as follows:

$$[x]_F := \sum_{1 \le i \le t} a_i F_{t+2-i}. \tag{1.1}$$

Note that these strings are in "most-significant-digit" first format. For example, $[2101]_F = 2F_5 + F_4 + F_2 = 14$.

It is also useful to define a (partial) inverse to $[x]_F$. By $(n)_F$ we mean the binary string x such that x is the Zeckendorf representation of n; alternatively, such that $[x]_F = n$ and x contains no occurrence of the block 11. In what follows, we adopt this string-based point of view almost exclusively. We can think of the condition "no occurrence of the block 11" as a *rule*, specifying which representations are valid, adopted precisely to guarantee both completeness and unambiguity.

In formal language theory, a language L is a (finite or infinite) collection of strings. A rule is then encoded by the language or set of strings that obey the rule. Completeness then becomes the assertion that for all n there exists a string $x \in L$ such that $[x]_F = n$, while unambiguity becomes the assertion that there do not exist distinct strings $x, y \in L$ such that $[x]_F = [y]_F$. [1]

Let us look at another example involving the Fibonacci numbers, one that can be seen as a mirror case to the Zeckendorf representation: the so-called *lazy* representation published by Brown in [5]. The computation of it differs from Zeckendorf only in their first steps: here we choose the largest index i such that $\sum_{j=2}^{i-1} F_j < n$.

Because of this, the representation is called "lazy" as it avoids including a Fibonacci number unless it has to. For example, the Brown representation $(10)_B = 1110$ avoids the inclusion of F_6, unlike the Zeckendorf one where $(10)_F = 10010$, since that can be compensated with the inclusion of F_5 and F_4. The consequence of this difference is that,

[1]We adopt the convention that two strings are considered to be the same if they differ only in the number of leading zeros. Thus, for example, $[100]_F = [0100]_F = 3$ are the same representation.

in this system, representations correspond (via Eq. (1.1)) to binary strings having no occurrence of the block 00 (where leading zeros are not even considered). Once again, this rule provides a numeration system that is both complete and unambiguous [5]. Table 1.1 gives both greedy (Zeckendorf) and lazy (Brown) representations for the first few natural numbers.

n	0	1	2	3	4	5	6	7	8	9	10	11
greedy	ϵ	1	10	100	101	1000	1001	1010	10000	10001	10010	10100
lazy	ϵ	1	10	11	101	110	111	1010	1011	1101	1110	1111

Table 1.1: Greedy and lazy Fibonacci representations.

The greedy and lazy representations are certainly not the only possible perfect numeration systems based on the Fibonacci numbers. In fact, there are *uncountably many* such systems! These result from making a choice, for all n having at least two different representations as sums of distinct Fibonacci numbers, about which particular representation is chosen to be valid. (By a result of Robbins [24], "most" numbers have more than one representation as a sum of distinct Fibonacci numbers.)

If we demand that the set of valid representation forms a regular language—that is, accepted by a finite automaton; see Chapter 4—there are still infinitely many different systems (although only countably many). For example, consider choosing the t'th largest possible representation for n in lexicographic order (if there are at least t), and otherwise the lexicographically first. It will follow from results below that, for each $t \geq 0$, this choice gives a regular language L_t of valid representations.

Some natural questions then arise: given a language L encoding the "rule" a representation must obey (such as no occurrence of the block 11, or no occurrence of the block 00), how can we determine if the corresponding set of Fibonacci representations is complete and unambiguous? And if it is complete, how can we efficiently find a representation for a given number n? Up to now, each new system proposed required a new proof, often a rather tedious case-based proof by induction. Here we provide a *general framework* for answering these questions "automatically", via an algorithm, in the case where the language of valid representations is regular.

These ideas are capable of generalization. For example, we can also consider representations for all integers $\mathbb{Z}$, instead of just the natural numbers $\mathbb{N}$. This can be achieved in two distinct ways:

- By allowing a larger digit set, say, $\{-1, 0, 1\}$;
- By using the negaFibonacci system invented by Bunder [9], based on the Fibonacci numbers of negative index F_{-n} for $n \geq 1$.

Once again, we would like a choice of valid representations that is complete and unambiguous.

In this work we show how to decide these properties, provided that the set of valid representations forms a regular language (which is indeed the case for all the proposed systems in the literature).

Now we give an outline of this thesis. In Chapter 2, we explain the basics of automata theory and first-order logic needed to understand the rest of the paper. In Chapter 3, we provide an introductory example of applying our framework on a lesser known base-2 representation. In Chapter 4 we elaborate on our framework, its theoretical foundation, and its complexity. In Chapter 5 we analyze the automata needed to translate arbitrary representations into representations that are recognized by the software used in our framework. In Chapter 6 we discuss how to test completeness and ambiguity for systems using digits 0 and 1 only. In Chapter 7 we discuss systems using digits $0, 1, -1$ only. In Chapter 8 we discuss representations for all integers, not just the natural numbers. In Chapter 9 we discuss new Fibonacci representations not previously studied in the literature. In Chapter 10 we discuss representations that make use of both the positively indexed and negatively indexed Fibonacci numbers. Finally, in Chapter 11, we conclude our work and present the open problem that remained for us.

We are interested in the Fibonacci representation because of its close connection to problems of number theory. One example is the Sturmian sequence called the Fibonacci word [4]; here the nth symbol of the sequence is exactly the least significant digit of the Fibonacci (Zeckendorf) representation of n. The Fibonacci word has many interesting properties, for example, a self-similar fractal curve can be created based on it [17].

Although not elaborated on in detail, our framework is applicable for number systems that are not Fibonacci-based. We provide an example of base-2 systems in Chapter 3. As long as a representation corresponds to a regular language and numbers in this representation can be converted to a form accepted by the software we use for computation, our framework can be applied to judge its completeness and unambiguity.

Chapter 2

Automata Theory and Logic

In this chapter, we recall some of the basics of automata theory and first-order logic.

2.1 Automata Theory

If Σ is a finite alphabet, then Σ^* denotes the set of all finite strings with symbols chosen from Σ. The empty string is denoted by ϵ.

A finite automaton M is a simple model of a computer. It takes inputs, which are finite strings over some finite alphabet Σ, and either accepts or rejects each string. At any moment, the automaton can be in one of a finite number of different states. Each input string is processed symbol-by-symbol, starting in an initial state, and reading each new symbol causes a transition to a new state, according to a transition table. Some states are distinguished as "final" or "accepting", and an input x is accepted iff processing x leads to a final state at the end. The set of all accepted strings is called the language of the automaton, and is denoted by $L(M)$.

More formally, a deterministic finite automaton (or DFA) is a quintuple $M = (Q, \Sigma, q_0, F, \delta)$ where

- Q is a finite nonempty set of states;
- Σ is the input alphabet;
- $q_0 \in Q$ is the starting state;

- $F \subseteq Q$ is the set of final states;
- and $\delta : Q \times \Sigma \to Q$ is the transition function.

The transition function δ is extended to the domain $Q \times \Sigma^*$ as follows: $\delta(q, x)$ is the state reached upon reading x, starting in state q. Then $L(M)$, the language of M, is defined to be $\{x \in \Sigma^* : \delta(q_0, x) \in F\}$.

It is often useful to display an automaton using a graphical format called a *transition diagram*. In such a diagram, states are represented by circles and accepting states are represented by double circles. Labeled arrows indicate the transitions, and a state with an arrow going in, but no origin, is the accepting state. Although the transition function $\delta(q, a)$ is required to be complete in a DFA (that is, it is well-defined for all states q and input symbols a), a common convention in transition diagrams is to omit transitions when they go to a "dead" state; that is, a state from which one can never reach an accepting state. For example, Figure 2.1 displays an automaton accepting those strings over $\{0, 1\}$ containing no two consecutive 1's.

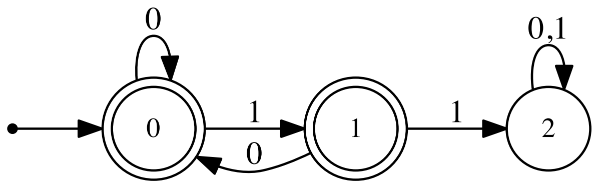


Figure 2.1: Automaton accepting strings containing no occurrence of 11.

There are basic algorithms in automata theory that can do operations on languages specified by DFA's. For example, given automata M_1 and M_2 for (respectively) languages L_1 and L_2, it is easy to algorithmically construct an automaton for the union or intersection of L_1 and L_2.

If L is the language of some DFA, we say that L is *regular*. A classic theorem of automata theory—Kleene's theorem—provides an alternate characterization of the regular languages, through regular expressions [13]. A regular expression denotes a language, and consists of subexpressions joined by various operations: concatenation, union, and

Kleene closure. The concatenation of two languages L_1 and L_2, L_1L_2, is defined to be $\{uv : u \in L_1, v \in L_2\}$. Set-theoretic union is denoted by the symbol $|$. The Kleene closure of a language L is the set of all finite strings obtained by concatenating 0 or more copies of strings of L, and is denoted by the symbol $*$. For example, the language of all strings containing no two consecutive 1's is denoted by the regular expression $(0|10)^*(1|\epsilon)$.

The following particular case of a theorem of Büchi [8] (as later corrected by Bruyère, Hansel, Michaux, and Villemaire [7]) is our principal tool in the paper.

Theorem 1. *There is a decision procedure that, given a first-order logical formula F involving natural numbers, comparisons, automata, and addition, and no free variables, will decide the truth or falsity of F. Furthermore, if F has free variables, the procedure constructs a DFA accepting those values of the free variables (in Fibonacci representation) that make F evaluate to* `TRUE`.

The specific case of the decision procedure for Fibonacci representation is discussed in detail by Mousavi, Schaeffer, and Shallit in [19, Theorem 2.2]. For the extension permitting quantification over integers instead of just natural numbers, see [29].

We should explain how automata can process pairs, triples, and generally k-tuples of inputs. This is done by replacing the input alphabet Σ with the alphabet $\overbrace{\Sigma \times \Sigma \times \cdots \times \Sigma}^{k \text{ times}}$. In other words, inputs are k-tuples of alphabet symbols. The i'th input then corresponds to the concatenation of the i'th components of all the k-tuples. Of course, this means that all k inputs have to have the same length; this is achieved by padding shorter inputs, if necessary, with leading zeros.

The decision procedure of Theorem 1 has been implemented in free software called `Walnut`, originally created by Hamoon Mousavi [18]; also see the book [27]. We recall some of the basics of `Walnut` syntax:

- `eval` evaluates a formula with no free variables and returns `TRUE` or `FALSE`; `def` defines an automaton for future use; `reg` defines a regular expression.
- In a regular expression, the period is an abbreviation for the entire alphabet.
- `&` is logical `AND`, `|` is logical `OR`, `=>` is logical implication, `<=>` is logical `IFF`, `~` denotes logical `NOT`.
- `A` denotes $\forall$ (for all); `E` denotes $\exists$ (there exists).

- `?msd_fib` tells `Walnut` to evaluate an arithmetic expression using Fibonacci representation.
- `?msd_neg_fib` tells `Walnut` to evaluate an arithmetic expression using negaFibonacci representation.
- `?msd_2` tells `Walnut` to evaluate an arithmetic expression using base-2 representation. This is also the default choice if no representation is specified in a formula.
- Negative numbers are placed in square brackets. For example, `[-1]` tells `Walnut` that to evaluate the number as -1.
- The order of the parameters of an automaton defined in `Walnut` with `def` follows alphabetical order. For example, the following is a definition statement from Chapter 3.

  ```
  def bbrEval_pos "?msd_2 Eu,v $bbrPos(s,u) & $bbrNeg(s,v) & z+v=u":
  ```

  Since there are two free variables in the formula, `s` and `z`, the automaton has two parameters. Therefore when we call `bbrEval_pos` with two arguments, the first argument will be considered as `s` in the formula by `Walnut` and, the second, `z`.

We use `Walnut` to do the computations needed to verify that a given system is complete and unambiguous.

For much more about `Walnut`, including a link to download it, visit
`https://cs.uwaterloo.ca/~shallit/walnut.html` .

2.2 First-Order Logic

In first-order logic, we quantify over individuals of the domain. This differentiates it from true/false propositions as well as from second-order logic where variables and quantifiers for sets of sets are allowed.

The terms and formulas of first-order logic consist of two kinds of symbols: logical symbols with fixed semantic meanings and non-logical symbols whose meanings are subject to interpretation. Logical symbols include the following.

- Connectives: $\neg$ for negation, $\wedge$ for conjunction, $\vee$ for disjunction, $\implies$ for implication, and $\iff$ for biconditional.
- The universal quantifier $\forall$ and the existential quantifier $\exists$.
- Punctuation symbols: the left and right parentheses and the comma.
- The free variable symbols and the bound variable symbols.

Non-logical symbols include:

- individual symbols such as the individual constant 0,
- relation symbols such as the equality symbol, and
- function symbols such as $+$.

Notice that there is not one single "first-order language" as we can choose the sets of individual, relation, and function symbols, and their meanings. An interpretation of a first-order language consists of a domain and a mapping of individual symbols to individuals in the domain, m-ary relation symbols to m-ary relations on the domain, and n-ary function symbols to n-ary functions on the domain.

For example, this thesis concerns the first-order languages using $\mathbb{N}$ or $\mathbb{Z}$ as a domain, with relations and functions being comparisons, addition, and automata representing Fibonacci-automatic sequences.

Chapter 3

Balanced Binary Representation

As an introduction to our approach, here we discuss a representation related to, but not as well known as, the familiar binary representation. We also prove its completeness and unambiguity.

We start by considering the binary number system where every non-negative integer can be uniquely represented as a sum of powers of 2, not considering leading zeros, as follows:

$$[x]_2 := \sum_{0\le i\le j} a_i 2^i, \tag{3.1}$$

where $a_i \in \{0,1\}$. If we expand the allowed alphabet to $a_i \in \{-1,0,1\}$ and we impose a rule stating that no two nonzero digits can appear consecutively in the representation, then we obtain what is call the *balanced binary representation* [25] or BBR (also considered by others including [23], [11], and [22] under different names). In Table 3.1, we give both the binary and balanced binary representations, $(n)_2$ and $(n)_{\mathrm{BBR}}$, for the first few natural numbers. We use $\bar{1}$ to denote the digit -1 in the representation.

n	0	1	2	3	4	5	6	7	8	9	10	11
$(n)_2$	ϵ	1	10	11	100	101	110	111	1000	1001	1010	1011
$(n)_{\mathrm{BBR}}$	ϵ	1	10	$10\bar{1}$	100	101	$10\bar{1}0$	$100\bar{1}$	1000	1001	1010	$10\bar{1}0\bar{1}$

Table 3.1: Examples of binary and balanced binary representations.

As a result of the expanded alphabet, not only the natural numbers but all integers can be expressed: in Table 3.2, we give the balanced binary representations $(n)_{\mathrm{BBR}}$ for the first few negative integers.

n	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11
$(n)_{\mathrm{BBR}}$	$\bar{1}$	$\bar{1}0$	$\bar{1}01$	$\bar{1}00$	$\bar{1}0\bar{1}$	$\bar{1}010$	$\bar{1}001$	$\bar{1}000$	$\bar{1}00\bar{1}$	$\bar{1}0\bar{1}0$	$\bar{1}0101$

Table 3.2: Examples of balanced binary representations.

Theorem 2. *The balanced binary representation is a complete and unambiguous numeration system for all integers.*

Proof. We first translate the rule disallowing consecutive nonzero digits into an automaton via the following regular expression and definition.

```
reg bbrExclude {-1,0,1} ".*(1|[-1])(1|[-1]).*":
def bbr "~$bbrExclude(s)":
```

The resultant automaton bbr is displayed in Figure 3.1; it accepts a string s iff s is over the alphabet $\{-1, 0, 1\}$ and has no consecutive nonzero digits.

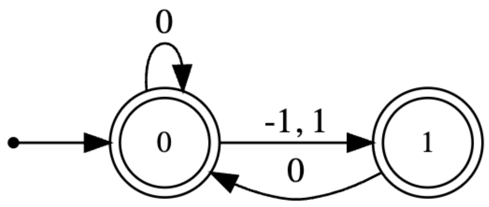


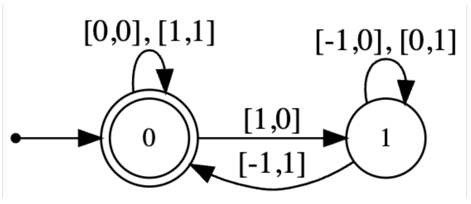
Figure 3.1: Automaton accepting valid balanced binary representations.

Then we consider how to make `Walnut` recognize the value of an arbitrary balanced binary representation: we convert it to the binary representation using the fact that the binary system is built into `Walnut` as the `msd_2` number system.

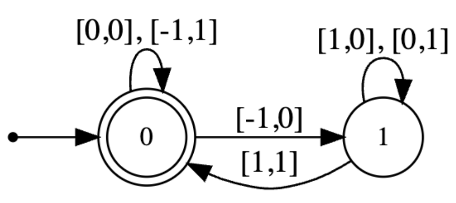
To convert an arbitrary BBR string, we first pull it apart into two binary strings using automata built with regular expressions. The automaton `bbrPos` takes two strings in parallel as input: a string w over the alphabet $\{-1, 0, 1\}$ and a string x over $\{0, 1\}$. And it accepts iff x has a 1 everywhere w has a 1. The automaton `bbrNeg` takes the same input, but it accepts iff x has a 1 everywhere w has a -1. Let s, u, and v be strings. If bbrPos accepts s and u and bbrNeg accepts s and v, then we know the value of s is the difference of the values of u and v. This is what we rely on to create the "conversion" automata displayed in Figure 3.2: bbrEval_pos for translating the positive integers and bbrEval_neg for the negative integers. Two are necessary because we use the msd_2 number system which assumes that the domain of objects is the natural numbers. Both automata work for the zero strings. The automaton bbrEval_pos takes two strings in parallel as input: a string s over the alphabet $\{-1, 0, 1\}$ and a number n in the binary representation. It accepts iff s, evaluated according to Eq. (3.1) with the expanded alphabet $a_i \in \{-1, 0, 1\}$, is equal to n. The automaton bbrEval_neg takes the same input, but it accepts iff s, evaluated in the same way, is equal to $-n$. We present the automata definitions as follows.

```
reg bbrPos {-1,0,1} msd_2 "([-1,0]|[0,0]|[1,1])*":
reg bbrNeg {-1,0,1} msd_2 "([-1,1]|[0,0]|[1,0])*":

def bbrEval_pos "?msd_2 Eu,v $bbrPos(s,u) & $bbrNeg(s,v) & z+v=u":
def bbrEval_neg "?msd_2 Eu,v $bbrPos(s,u) & $bbrNeg(s,v) & z+u=v":
```



(a) The DFA bbrEval_pos.



(b) The DFA bbrEval_neg.

Figure 3.2: Automata for conversion to the binary representation.

Now we prove completeness by testing two statements in Walnut. The statement bbrC_pos asserts that, for all non-negative integers n, there exists a string s such that s is a valid balanced binary representation and s evaluates to n. The statement bbrC_neg makes the same assertion except that s should evaluate to $-n$.

```
eval bbrC_pos "?msd_2 An Es $bbr(s) & $bbrEval_pos(s,n)":
# evaluates to TRUE, 17 ms

eval bbrC_neg "?msd_2 An Es $bbr(s) & $bbrEval_neg(s,n)":
# evaluates to TRUE, 2 ms
```

Before we can prove unambiguity, we need to create an automaton to test if two strings over the alphabet $\{-1, 0, 1\}$ are the same. We do so as follows.

```
reg same {-1,0,1} {-1,0,1} "([-1,-1]|[0,0]|[1,1])*":
```

To prove unambiguity, we again use two statements to cover all integers. Using `bbrU_pos`, we assert that, for any non-negative integer n, there do not exist two different valid balanced binary representations s and t such that both s and t evaluate to n. The statement `bbrU_neg` makes the same assertion except that s and t should evaluate to $-n$.

```
eval bbrU_pos "?msd_fib ~En,s,t $bbr(s) & $bbr(t) & (~$same(s,t))
& $bbrEval_pos(s,n) & $bbrEval_pos(t,n)":
# evaluates to TRUE, 7 ms

eval bbrU_neg "?msd_fib ~En,s,t $bbr(s) & $bbr(t) & (~$same(s,t))
& $bbrEval_neg(s,n) & $bbrEval_neg(t,n)":
# evaluates to TRUE, 5 ms
```

This completes our proof that the balanced binary representation is complete and unambiguous for all integers. □

We consider this example concerning the familiar binary number systems as an introduction to our framework which will be detailed in the next chapter. The rest of our work focuses on more exotic systems based on the Fibonacci numbers, as they yield more interesting variations.

Chapter 4

Our Framework

In this chapter, we introduce our approach and discuss its theoretical underpinning and complexity. We now state one of our main results.

Theorem 3. *There is an algorithm that, given rules that specify which representations are valid (in the form of a regular language L of all valid representations), will decide if the corresponding numeration system based on the Fibonacci numbers is complete and unambiguous for $\mathbb{N}$.*

Proof. Using Theorem 1, it suffices to express the properties of completeness and unambiguity as a first-order logic formula F. Once this is done, the decision algorithm can determine if F is true or false.

Completeness says every integer has a representation in L. We can express this as follows:

$$\forall n \ \exists x \ x \in L \ \wedge \ [x]_F = n, \tag{4.1}$$

Unambiguity says that no integer has two distinct representations in L. We can express this as follows:

$$\neg\exists x, y \in L \ (\neg \, \text{equal}(x, y)) \ \wedge \ [x]_F = [y]_F. \tag{4.2}$$

Here equal means that x and y are the same, up to leading zeros.

These two statements above include operations checking:

1. the membership of an arbitrary string in a language ($x \in L$)

2. whether a string, evaluated according to Eq. (1.1), corresponds to a certain number ($[x]_F = n$)

3. whether two strings are equal up to leading zeros ($\text{equal}(x, y)$)

4. whether two numbers are equal ($[x]_F = [y]_F$)

5. whether two numbers add correctly.

For 4 and 5, we rely on automata discussed in [19]. For 3, we can construct such automata in `Walnut` via regular expressions. For 2, we devote Chapter 5 to constructing automata for such purposes. For 1, if the language L is regular, the membership test can be expressed in a first-order language involving addition and automata. Therefore, completeness and unambiguity are decidable. □

Theorem 3 can be carried out mechanically with `Walnut` as follows.

- First, we define the language L of valid representations and construct a corresponding automaton via a regular expression. This is used to implement the expressions $x \in L$ in Formula 4.1 and $x, y \in L$ in Formula 4.2. The DFA `bbr` in Chapter 3 and `greedy` in Chapter 6.1 are examples of such automata.

- Since L may not be a numeration system built into `Walnut`, we need a "converter" automaton to translate between arbitrary representations and representations that are recognized and can be processed by `Walnut`. This is necessary for implementing the comparisons $[x]_F = n$ in Formula 4.1 and $[x]_F = [y]_F$ in Formula 4.2. An example of such "converter" automata are `bbrEval_pos` and `bbrEval_neg` we saw in Chapter 3. And we will see many more examples related to the Fibonacci-based numeration systems in Chapter 5, 7.2.1, 8, and 10.

- Lastly, we need to translate the first-order logical formulas asserting completeness and unambiguity of L into `Walnut` code. Examples include the `bbrC_pos` and `bbrU_neg` evaluation statements in Chapter 3 and `farDiffC` and `farDiffU` in Chapter 8.1. Our proofs are completed as these evaluation statements return `TRUE`.

4.1 Algorithm Complexity

Now we discuss the complexity of our algorithm.

Theorem 4. *Let S be an arbitrary numeration system. Let M be the automaton accepting the regular language of all valid representations in S. We can check the unambiguity of S in time polynomial in the number of states of M.*

Consider the evaluation statement for unambiguity; we use our work from Chapter 3 as an example:

```
eval bbrU_pos "?msd_fib ~En,s,t $bbr(s) & $bbr(t) & (~$same(s,t))
& $bbrEval_pos(s,n) & $bbrEval_pos(t,n)":
```

Assume the automaton representing the rule, such as `bbr`, has n states. Intersecting two copies of it with `same`, which has two states, and two copies of `bbrEval_pos`, each of which also has two states, gives an automaton of $8n^2$ states. As we will see in Chapter 5, the number of states of other kinds of "comparator" automata is fixed. Therefore the conclusion of this analysis holds for other representations instead of just the example here.

Theorem 5. *Let S be an arbitrary numeration system. There is an algorithm checking the completeness of S, running in exponential time in the number of states of the automaton representing the rules of S.*

Consider the evaluation statement for completeness from Chapter 3:

```
eval bbrC_pos "?msd_2 An Es $bbr(s) & $bbrEval_pos(s,n)":
```

The universal quantifier in $\forall n$ in Formula 4.1, implemented as `An` in the above evaluation statement, is evaluated via subset construction under our approach. Since the subset construction could, in principle, increase the size of an NFA with n states to a DFA with 2^n states, this provides an exponential upper bound on the complexity of the algorithm. It remains an open problem for us whether there exists a way to check the completeness of a numeration system in polynomial time.

4.2 Finding a Representation

Once we prove that a regular language L provides a system that is complete, we can find a representation in L for any particular number n efficiently in $O(\log n)$ time.

The first step is to represent n in the canonical form of a number system $(n)_C$, for example, $(n)_2$ for binary number systems or the Zeckendorf representation $(n)_F$ for Fibonacci-based number systems. This can be done using the greedy algorithm.

Let the converter automaton be M. We construct a new automaton from M using two intersections. The first intersection is with an automaton with a first component that belongs to L, while the second component is arbitrary. The second intersection is with an automaton where the first component is arbitrary, and the second is of the form $0^*(n)_C$. This gives a new automaton of $O(\log n)$ states, and it now suffices to find any accepting path (a path from the initial state to the final state). This can be done in linear time in the number of states using depth-first or breadth-first search. This gives us an $O(\log n)$ algorithm to find a representation. Thus we have proved:

Theorem 6. *Suppose L is a regular language. If L is complete, we can find a representation for an integer n in $O(\log n)$ time.*

Remark 7. Here we use the convention of the so-called "word RAM" model, where we assume that n fits in a single machine word, or more generally that we can perform basic operations on integers with $O(\log n)$ bits in unit time.

We will see more examples concerning Fibonacci-based number systems in later chapters. Here we illustrate with an example of finding the balanced binary representation of -27. We build a DFA:

```
def bbrN27 "?msd_2 $bbrEval_neg(s,x) & $bbr(s) & x=27":
```

The resultant DFA from this command of intersections is displayed in Figure 4.1. The accepted inputs are $[s = 0^n(-27)_{\text{BBR}},\ x = 0^n(27)_2]$ where $n \geq 0$. 0^n denotes n copies of 0 concatenated together. Thus finding any accepting path gives the valid balanced binary representation of -27 as -100101. Notice x is positive because the converter automaton we use is `bbrEval_neg` which evaluates s against the negated value of x; see Chapter 3 for its detailed definition.

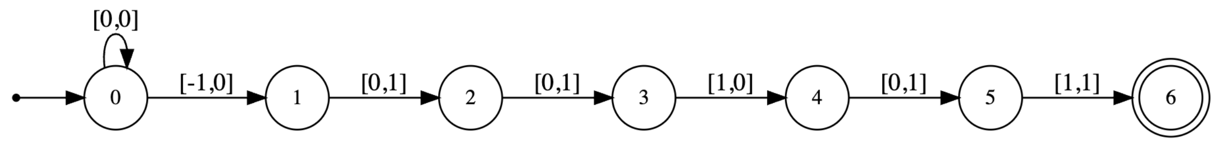


Figure 4.1: Automaton showing -27 in balanced binary representation.

Chapter 5

Converter Automata

As we discussed in Chapter 4, a crucial step in our proof is translating numbers in arbitrary representations into representations that are recognized and can be processed by `Walnut`. We build converter automata for this purpose. In this chapter we go into detail about these automata.

Two Fibonacci-based numeration systems are built into `Walnut`: Zeckendorf and negaFibonacci. To test an arbitrary Fibonacci representation for completeness and unambiguity, we need to "convert" it to either the Zeckendorf or negaFibonacci representation. To be more precise, the conversion is done via a comparison of numbers in two different representations. We start with focusing on the conversion to negaFibonacci because Zeckendorf can only be used to represent the natural numbers. Furthermore, the conversion to Zeckendorf requires a much simpler automaton and a significantly shorter proof of correctness.

5.1 Conversion to the NegaFibonacci Representation

We first introduce the negaFibonacci representation in more detail. Bunder [9] invented it as a numeration system representing all integers, instead of just the natural numbers. In this system, we write integers as a sum of distinct Fibonacci numbers with negative indices, subject to the condition that no two consecutive Fibonacci numbers can be used. Let $x = a_t \cdots a_1$ be a string (or word) over $\{0, 1\}$. We define its value as a negaFibonacci representation as follows:

$$[x]_{\mathrm{NF}} := \sum_{1 \le i \le t} a_i F_{-i}. \tag{5.1}$$

Note that these strings are in "most-significant-digit" first format. For example, $[1001]_{\text{NF}} = F_{-4} + F_{-1} = -3 + 1 = -2$. Since $F_{-n} = (-1)^{n+1} F_n$ for $n \in \mathbb{N}$, this is the same as enforcing the requirement in a Fibonacci representation $a_t F_t + \cdots + a_1 F_1$ with digits $a_i \in \{-1, 0, 1\}$,

(a) only the terms with odd indices are allowed to be positive,

(b) only the terms with even indices are allowed to be negative, and

(c) no two consecutive nonzero digits can appear.

This requirement guarantees that the representation is complete and unambiguous. We give two tables of negaFibonacci representations of integers: the first few natural numbers are given in Table 5.1 and the first few negative integers are given in Table 5.2. We use $\bar{1}$ to denote the digit -1 in the representation.

n	0	1	2	3	4	5	6	7	8	9	10
$(n)_{\text{NF}}$	ϵ	1	100	101	$100\bar{1}0$	10000	10001	10100	10101	$100\bar{1}0\bar{1}0$	$100\bar{1}000$

Table 5.1: Examples of negaFibonacci representations for non-negative integers.

n	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10
$(n)_{\text{NF}}$	$\bar{1}0$	$\bar{1}001$	$\bar{1}000$	$\bar{1}0\bar{1}0$	$\bar{1}00101$	$\bar{1}00100$	$\bar{1}00001$	$\bar{1}00000$	$\bar{1}000\bar{1}0$	$\bar{1}0\bar{1}001$

Table 5.2: Examples of negaFibonacci representations for negative integers.

For conversions to the negaFibonacci representation, we use a converter automaton of 53 states, `fsc`. We "guessed" its composition following the techniques described in [27, Chap. 5.6] using a variant of the Myhill-Nerode theorem. The automaton `fsc` takes two inputs in parallel:

1. a string s over the alphabet $\{-1, 0, 1\}$, and
2. a number n in negaFibonacci representation.

The automaton `fsc` accepts iff s, evaluated according to Eq. (1.1), is equal to n. Now we prove the correctness of our "guess".

Theorem 8. *The converter automaton* `fsc` *is correct.*

Proof. There are three parts to this proof. We need to show that:

1. for every string s over the alphabet $\{-1, 0, 1\}$, there exists a number n such that `fsc` accepts the pair (s, n),
2. for every string s, there exists only one number n such that the pair is accepted, and
3. every string s is accepted with the correct number n.

Before we can prove the first point, we need to consider that, when an automaton accepts multiple inputs in parallel, the shorter one is always padded to make both inputs have the same length. The same number in different representations can have different lengths; for example, recall the Brown representation of 9 from Chapter 1, $(9)_B = 1101$, has three fewer digits than $(9)_{\mathrm{NF}} = 100\bar{1}0\bar{1}0$. So for the string $(9)_B$ to be accepted with $(9)_{\mathrm{NF}}$, we need to pad $(9)_B$ with three zeros. We are not concerned about the case where the arbitrary representation string is longer than the negaFibonacci string because the negaFibonacci representation is built into `Walnut` and it is padded with zeros automatically as needed. Therefore we prove the first point, only for the strings that start with at least three zeros, in `Walnut` as follows.

```
reg three0s {-1, 0, 1} "000.*":
eval existn "As $three0s(s) => En $fsc(s,n)":
# evaluates to TRUE
# 2 ms
```

For the second point, we assert the following:

$$\neg\exists s, m, n \;\; \mathrm{fsc}(s, m) \wedge \mathrm{fsc}(s, n) \wedge m \neq n.$$

This translates into `Walnut` as follows.

```
eval onlyn "~Es,m,n $fsc(s,m) & $fsc(s,n) & m!=n":
# evaluates to TRUE
# 20 ms
```

For the third point, we proceed by induction on the number of nonzero terms in s. Let s_{k+1} be a string with $k+1$ nonzero terms. We assume s_k is s_{k+1} without its most significant nonzero term or the leftmost term. Let (s_{k+1}, m) and (s_k, n) be string-number pairs accepted by `fsc`. Let x be a number in the Fibonacci representation that records the absolute difference between m and n. Therefore $(x)_F$ contains only one 1 which is the difference between s_k and s_{k+1}. And $(x)_F$ shifted one position to the left equal either $(m-n)_{\mathrm{NF}}$ or $(n-m)_{\mathrm{NF}}$. To assert this relation, we build the automaton `fibToNeg`. It takes two arguments in parallel: a number u in Fibonacci and a number v in NegaFibonacci. It accepts iff $(u)_F$ has only one 1 and $(u)_F$ shifted one position to the left equal $(v)_{\mathrm{NF}}$.

```
reg fibToNeg msd_fib msd_neg_fib "[0,0]*[0,1][1,0][0,0]*":
```

Now, to prove that x records the correct difference, we have four cases to consider. Since m and n are expressed in NegaFibonacci and x is in Fibonacci, we need to differentiate based on if the leftmost term of s_{k+1} is a 1 or -1 and if the leftmost term is followed by an odd or even number of terms.

1. If the leftmost term of s_{k+1} is a 1 followed by an odd number of terms, we use the following regular expression to assert how s_{k+1}, s_k, and $(x)_F$ are related:

   ```
   reg diff1odd {-1, 0, 1} {-1, 0, 1} msd_fib
   "[0,0,0]*[1,0,1]([0,0,0]|[1,1,0]|[-1,-1,0])
   (([0,0,0]|[1,1,0]|[-1,-1,0])([0,0,0]|[1,1,0]|[-1,-1,0]))*":
   ```

   In this case, $(x)_F$ shifted one position to the left should equal $(m-n)_{\mathrm{NF}}$; we test this as follows.

   ```
   eval correct1odd "As,t ?msd_fib Ax ?msd_neg_fib Am,n
   ($diff1odd(s,t,x) & $fsc(s,m) & $fsc(t,n)) => $fibToNeg(x, m-n)":
   # evaluates to TRUE, 30 ms
   ```

2. If the leftmost term is a -1 followed by an odd number of terms, then the shifted $(x)_F$ should equal $(n-m)_{\mathrm{NF}}$. We assert how s_{k+1}, s_k, and $(x)_F$ are related in this case and test $(x)_F$ as follows.

```
reg diffN1odd {-1, 0, 1} {-1, 0, 1} msd_fib
"[0,0,0]*[-1,0,1]([0,0,0]|[1,1,0]|[-1,-1,0])
(([0,0,0]|[1,1,0]|[-1,-1,0])([0,0,0]|[1,1,0]|[-1,-1,0]))*":

eval correctN1odd "As,t ?msd_fib Ax ?msd_neg_fib Am,n
($diffN1odd(s,t,x) & $faut2(s,m) & $faut2(t,n)) => $fibToNeg(x, n-m)":
# evaluates to TRUE, 42 ms
```

We see that, to handle the different cases, we only need to change the regular expression asserting the relationship of s_{k+1}, s_k, and $(x)_F$ as well as the arguments to `fibToNeg`.

3. If the leftmost term is a 1 followed by an even number of terms, then the shifted $(x)_F$ should equal $(n-m)_{\text{NF}}$. We test this case as follows.

```
reg diff1even {-1, 0, 1} {-1, 0, 1} msd_fib
"[0,0,0]*[1,0,1](([0,0,0]|[1,1,0]|[-1,-1,0])
([0,0,0]|[1,1,0]|[-1,-1,0]))*":

eval correct1even "As,t ?msd_fib Ax ?msd_neg_fib Am,n
($diff1even(s,t,x) & $faut2(s,m) & $faut2(t,n)) => $fibToNeg(x, n-m)":
# evaluates to TRUE, 41 ms
```

4. If the leftmost term is a -1 followed by an even number of terms, then the shifted $(x)_F$ should equal $(m-n)_{\text{NF}}$. We test this case as follows.

```
reg diffN1even {-1, 0, 1} {-1, 0, 1} msd_fib
"[0,0,0]*[-1,0,1](([0,0,0]|[1,1,0]|[-1,-1,0])
([0,0,0]|[1,1,0]|[-1,-1,0]))*":

eval correctN1even "As,t ?msd_fib Ax ?msd_neg_fib Am,n
($diffN1even(s,t,x) & $faut2(s,m) & $faut2(t,n)) => $fibToNeg(x, m-n)":
# evaluates to TRUE, 44 ms
```

Finally, we establish the base case that strings of only zeros are accepted as 0 as follows.

```
reg only0s {-1, 0, 1} "0*":
eval baseCase "As $only0s(s) => $fsc(s,0)":
# evaluates to TRUE, 20 ms
```

This completes our induction proof. Therefore every s is accepted with the correct n and `fsc` is correct. □

5.2 Decomposition of `fsc`

To better comprehend the 53-state-automaton `fsc`, we try to build it piece by piece from smaller automata. Recall the two inputs to `fsc`: let t be a string over $\{-1, 0, 1\}$ and z be a number in the negaFibonacci representation. We consider how t can be converted to z as follows.

The first step is to shift t one position to the left since t is evaluated according to Eq. (1.1) whereas the negaFibonacci representation makes use of F_1. We build a "shifter" automaton called `lshift` and display it in Figure 5.1. The command `lshift(t,s)` guarantees the string s as t shifted one position to the left.

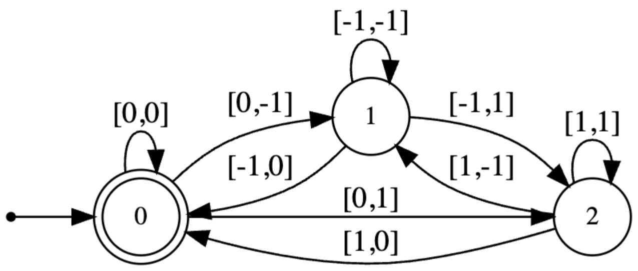


Figure 5.1: Left shifter automaton.

Now we consider how we can convert the shifted string s to fulfil the three requirements of the negaFibonacci representation:

1. only the terms with odd indices are allowed to be positive,
2. only the terms with even indices are allowed to be negative, and
3. no two consecutive nonzero digits can appear.

Consider the following strategy of pulling apart s into four valid negaFibonacci representations u, v, i and j.

1. We use the automaton `decompEven1`, built using a regular expression, and the command `decompEven1(s,u)` to obtain a string u. The string u has a 1 everywhere s has a 1 with an even index. However, the value of u, evaluated as a negaFibonacci representation, is the negated value of the 1's with even indices in s.

   ```
   reg decompEven1 {-1,0,1} msd_neg_fib "[0,0]*|([0,0]*
   ([-1,0]|[0,0]|[1,0])(([1,1]|[-1,0]|[0,0])([-1,0]|[0,0]|[1,0]))*)":
   ```

2. Similarly, we use `decompOddN1(s,v)` to ensure that the string v has a 1 everywhere s has a -1 with an odd index. Like u, the value of v needs to be negated for the same reason.

   ```
   reg decompOddN1 {-1,0,1} msd_neg_fib "[0,0]*|([0,0]*
   ([-1,1]|[0,0]|[1,0])(([-1,0]|[0,0]|[1,0])([-1,1]|[0,0]|[1,0]))*)":
   ```

3. Following the same idea, `decompCopyEven(s,i)` (`decompCopyOdd(s,j)`) ensures that the string i (j) has a 1 everywhere s has a -1 (1) with an even (odd) index. Notice the values of i and j do not need to be negated since they, evaluated as negaFibonacci representations, already represent the values of the digits they picked out from s.

   ```
   reg decompCopyEven {-1,0,1} msd_neg_fib "[0,0]*|([0,0]*
   ([-1,0]|[0,0]|[1,0])(([-1,1]|[0,0]|[1,0])([-1,0]|[0,0]|[1,0]))*)":

   reg decompCopyOdd {-1,0,1} msd_neg_fib "[0,0]*|([0,0]*
   ([1,1]|[-1,0]|[0,0])(([-1,0]|[0,0]|[1,0])([1,1]|[-1,0]|[0,0]))*)":
   ```

For the reason discussed above, now we consider how to negate the values of u and v. We build an equality checker automaton `nfequal` displayed in Figure 5.2. It takes two negaFibonacci representations m and n in parallel as input and accepts iff $[m]_{\mathrm{NF}} = -[n]_{\mathrm{NF}}$. As an example, consider the input $[1,0][0,1][0,0][0,1]$ to `nfequal`, whose first components spell out $m = 1000$ and whose second components spell out $n = 0101$. Given this input, the automaton visits, starting in state 0, successively, states 1, 3, 4, and 3 and accepts.

Therefore m should evaluate to the negated value of n. Indeed we have $[m]_{\mathrm{NF}} = F_{-4} = -3 = -(F_{-3} + F_{-1}) = -[n]_{\mathrm{NF}}$. The commands `nfequal(u,x)` and `nfequal(v,y)` ensure that x and y are the negated values of u and v, respectively.

```
def nfequal "?msd_neg_fib a=0-b":
```

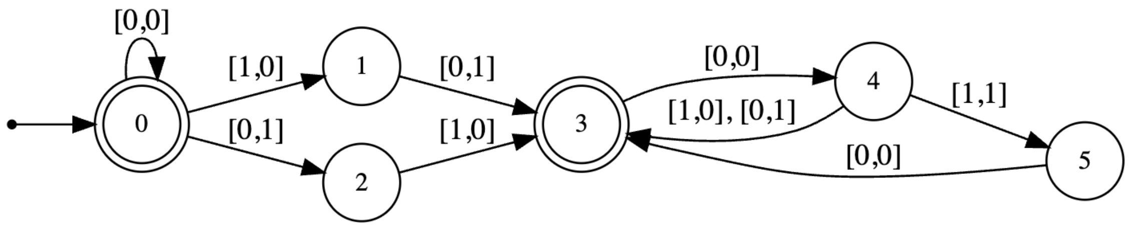


Figure 5.2: Equality checking automaton.

Finally, we construct z as the sum of x, y, i, and j. We code the entire process in the `Walnut` code below which creates a 55-state automaton `legofsc`. It takes the following in parallel as input:

1. a string s over the alphabet $\{-1, 0, 1\}$, and
2. a number n in negaFibonacci representation.

The automaton `legofsc` accepts only if s, evaluated according to Eq. (1.1), is equal to n. Recall that our goal is to have `legofsc` function the same as `fsc`. However, notice that `legofsc` has two more states than `fsc` does. The effect of this is that `fsc` accepts every pair of $[s, n]$ input `legofsc` accepts, but not the other way around. If a pair of $[s, n]$ input is accepted by `fsc` but not `legofsc`, then the s is nonempty and has at most one leading zero. We prove that, as long as the string input s in $[s, n]$ starts with at least two zeros, then `legofsc` and `fsc` are equivalent. Since representations are not differentiated by the number of leading zeros, for the purpose of this thesis, `legofsc` is equivalent to `fsc` and we have accomplished what we set out to do.

```
def legofsc "?msd_neg_fib Es,u,v,x,y,i,j $lshift(t,s)
& $decompEven1(s,u) & $decompOddN1(s,v) & $nfequal(u,x) & $nfequal(v,y)
& $decompCopyEven(s,i) & $decompCopyOdd(s,j) & z=x+y+i+j":
```

```
reg two0s {-1, 0, 1} "00.*":
eval fscEq "At,z $two0s(t) =>
($legofsc(t,z) => $fsc(t,z)) & ($fsc(t,z) => $legofsc(t,z))":
# evaluates to TRUE, 11 ms
```

5.3 Conversion to the Zeckendorf Representation

To convert from an arbitrary Fibonacci representation to the Zeckendorf representation, we observe the following:

Proposition 9. *We can convert a binary string x to a Zeckendorf representation y for the same number using the following algorithm: first append a 0 on the front, if necessary. Then scan the string from left to right, replacing each occurrence of "011" successively with "100".*

Proof. Each such replacement does not change the value of $[x]_F$ given how Fibonacci numbers are composed. The algorithm terminates because each replacement lowers the total number of 1's by 1. Finally, the algorithm cannot result in two consecutive 1's, because it introduces two consecutive 0's, only the second of which can later change to a 1. □

We implement this idea as a DFA `fcanon` that takes two inputs in parallel:

1. a string s over the alphabet $\{0, 1\}$, and
2. a number n in Zeckendorf representation.

The automaton `fcanon` accepts if s, evaluated according to Eq. (1.1), is equal to n. The automaton keeps track of $[x']_F - [y']_F$ for the prefix x' of x seen so far, and similarly for the prefix y' of y seen so far. Note that we assume that x and y have the same length, with the shorter of the two prefixed by leading zeros, if necessary. It is depicted in Figure 5.3. This automaton was given by Berstel [3] in a slightly different form. Also see [26].

As an example, consider the input $[0, 1][1, 0][1, 0][1, 1][0, 0]$ to `fcanon`, whose first components spell out $x = 01110$ and whose second components spell out $y = 10010$. Starting in state 0, the automaton visits, successively, states $1, 2, 0, 3, 0$, and hence accepts—as it should, since $[x]_F = [y]_F$.

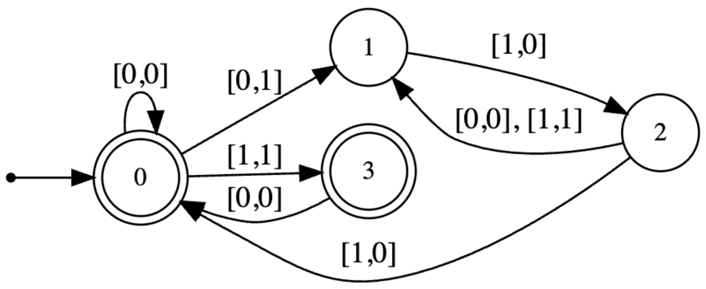


Figure 5.3: DFA `fcanon` for conversion to the Zeckendorf representation.

Chapter 6

Representation of Natural Numbers Using Digits 0 and 1 Only

In this chapter we consider representations of the natural numbers by Fibonacci numbers using digits 0 and 1 only.

6.1 Zeckendorf Representation: The Canonical Form

Recall that the definition of the Zeckendorf representation can be seen as a rule stating that a valid representation is over the alphabet $\{0, 1\}$ with no consecutive 1's. We write the following regular expression to create an automaton `greedy` accepting only the strings that can be valid Zeckendorf representations.

```
reg negExclude {-1,0,1} ".*[-1].*":
reg greedyExclude {-1,0,1} ".*11.*":
def greedy "~$greedyExclude(s) & ~$negExclude(s)":
```

We display `greedy` in Figure 6.1. Notice the DFA `greedy` is defined over the alphabet $\{-1, 0, 1\}$ even though any valid representation is over $\{0, 1\}$. This is because `greedy` needs to work with the converter automaton `fsc`, as defined in Chapter 5.1, to prove the completeness and unambiguity of the Zeckendorf representation, and `fsc` is defined over $\{-1, 0, 1\}$. By including the expression `~$negExclude(s)`, the DFA `greedy` is effectively defined over $\{0, 1\}$.

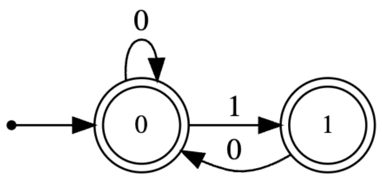


Figure 6.1: DFA for Zeckendorf's greedy representation.

We test the completeness and unambiguity of the Zeckendorf representation as follows.

```
eval zeckendorfC "?msd_neg_fib An n>=0 => Es $fsc(s,n) & $greedy(s)":
# evaluates to TRUE, 20 ms

eval zeckendorfU "~En,s,t $greedy(s) & $greedy(t) & (~$same(s,t))
& $fsc(s,n) & $fsc(t,n)":
# evaluates to TRUE, 13 ms
```

These `Walnut` commands are direct translations of Formula 4.1 and Formula 4.2. We add the `n>=0` restriction in the completeness evaluation statement because the `msd_neg_fib` number system is used and it assumes that the domain of objects is all integers; we choose the `msd_neg_fib` number system so that `n` can be properly processed by `fsc`. As both statements evaluate to `TRUE`, we prove that the Zeckendorf representation is complete and unambiguous.

6.2 Brown's "Lazy" Representation

Now we consider the Brown representation. Recall that its definition can be seen as a rule stating that any valid representation cannot have consecutive 0's.[1] We define the following DFA that accepts only the regular language made of strings abiding by that rule.

```
reg lazyExclude {-1,0,1} "0*1([-1]|0|1)*00([-1]|0|1)*":
def lazy "~$lazyExclude(s) & ~$negExclude(s)":
```

[1]This rule does not concern the leading zeros. In general, the rules / conditions imposed by representations do not concern the leading zeros.

We evaluate the completeness and unambiguity of the Brown representation using similar statements as what we have seen for Zeckendorf.

```
eval brownC "?msd_neg_fib An n>=0 => Es $fsc(s,n) & $lazy(s)":
# evaluates to TRUE, 9 ms

eval brownU "?msd_neg_fib ~En,s,t $lazy(s) & $lazy(t) & (~$same(s,t))
& $fsc(s,n) & $fsc(t,n)":
# evaluates to TRUE, 12 ms
```

As both statements evaluate to `TRUE`, we proved that the Brown representation is a perfect system. Note that this conclusion can be proved in another way using the converter automaton `fcanon` instead of `fsc`. To do that, we modify the definition of `lazy`, as the alphabet it is defined over needs to agree with the alphabet of `fcanon`. Note that both definitions of the DFA `lazy` give the same transition diagram shown in Figure 6.2. Then we check the completeness and unambiguity of the Brown representation using `fcanon` as follows.

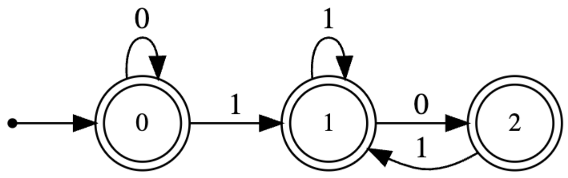


Figure 6.2: DFA for Brown's lazy representation.

```
reg lazyExclude {0,1} "0*1(0|1)*00(0|1)*":
def lazy "~$lazyExclude(s)":

reg equal {0,1} {0,1} "([0,0]|[1,1])*":
eval brownC "?msd_fib An Es $fcanon(s,n) & $lazy(s)":
eval brownU "?msd_fib ~En,s,t $lazy(s) & $lazy(t) & (~$equal(s,t))
& $fcanon(s,n) & $fcanon(t,n) ":
```

Again, both return `TRUE`. Here we use `equal` to check whether two binary strings over $\{0, 1\}$ are the same.

From comparing the two sets of definition and statements, we can tell that `fcanon` is a better fit for Brown's representation even though the advantage is not great due to the simplicity of the representation. As we will see in later sections, the advantage of choosing a suitable converter automaton becomes more pronounced when dealing with more complicated representations. Several factors impact this decision:

- One concerns whether the alphabet of the representation underlying the converter agree with that of the representation our proofs are about. For our example here, Brown's representation uses the same binary alphabet `fcanon` uses, and that consistency simplifies the definition of `lazy`.
- The other factor concerns whether the converter can accept the same range of numbers as the definition DFA. In our example above, the negaFibonacci representation underlying `fsc` can represent all integers, whereas Brown's representation is for only the natural numbers. That is why the completeness assertion can omit `n>=0` with the use of `fcanon`.
- One additional consideration is the level of complexity of the converter automata: `fsc` has 49 more states than `fcanon` and therefore requires a much longer and complicated proof of correctness.

With the proven completeness of Brown's representation, we give an example of finding the Brown representation for a particular number, say 17, to echo the discussion in Chapter 4.2. We define the following DFA.

```
def brown17 "?msd_fib $fcanon(s,x) & $lazy(s) & x=17":
```

The DFA `brown17` takes two parallel inputs: $[s = 0^*(17)_B,\ x = 0^*(17)_F]$ and is displayed in Figure 6.3. Following its transitions, we find our answer $(17)_B = 11101$.

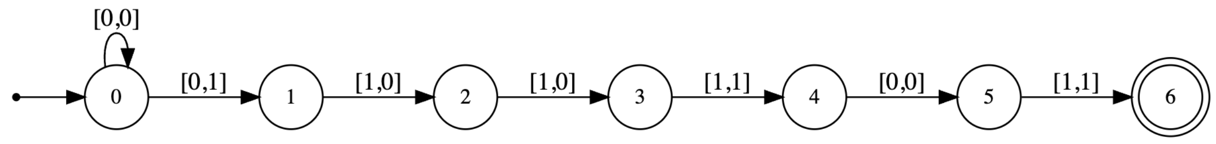


Figure 6.3: Automaton showing 17 in the Brown representation.

6.3 A New Representation: EPAS

In addition to shortening proofs about established systems, our approach can test new systems efficiently. Here we turn to a *new* system for representing natural numbers we call EPAS (even-prefix, alternating suffix). In this system, a representation is valid if it can be broken into two parts: a (possibly empty) prefix that has all its runs of 1's of even length, and a (possibly empty) suffix that looks like $101010\cdots$ (which can end in either 0 or 1). The first few valid representations $(n)_{\text{EPAS}}$ are given in Table 6.1. We display the automaton accepting only valid EPAS representations in Figure 6.4.

n	1	2	3	4	5	6	7	8	9	10	11	12
$(n)_{\text{EPAS}}$	1	10	11	101	110	111	1010	1100	1101	1110	1111	10101

Table 6.1: Examples of EPAS representations.

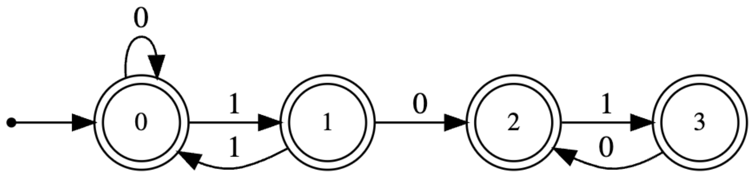


Figure 6.4: Automaton for the new system EPAS.

Theorem 10. *Let* $L = (0|11)^*(10)^*(1|\epsilon)$. *Then* L *is complete and unambiguous for Fibonacci representations.*

Proof. We use the following `Walnut` code:

```
reg epas {0,1} "(0|11)*(10)*(1|())":
eval epasC "?msd_fib An Ex $epas(x) & $fcanon(x,n)":
eval epasU "?msd_fib ~En,x,y $epas(x) & $epas(y) & (~$equal(x,y))
& $fcanon(x,n) & $fcanon(y,n)":
```

And both return `TRUE`. □

This example shows how simple it is to test a new proposed representation for completeness and unambiguity.

Chapter 7

Representation of Natural Numbers Using Digits -1, 0 and 1

We now turn to representations using digits -1, 0, and 1 in the Fibonacci system. Recently, Hajnal [12] described three Fibonacci representations using Eq. (1.1) to associate a string $s = e_t e_{t-1} \cdots e_2 \in \{-1, 0, 1\}^*$ with a natural number n: *alternating, even,* and *odd.* Using induction and case-based arguments, he proved that each of these three representations is complete and unambiguous.

Using automata, we can replace his rather long arguments with our general approach. We first describe each of his systems, and show that the set of valid representations for all natural numbers is a regular language.

7.1 Representation Definitions

7.1.1 Hajnal's Alternating Representation

The alternating representation requires a representation to fulfill four conditions:

1. the most significant nonzero term is positive,
2. two adjacent nonzero terms cannot be of the same sign,
3. two adjacent nonzero terms have at least one zero in between, and

4. if there are two or more nonzero terms, then there have to be at least two zeros between the last and the second-last nonzero terms.

We denote a number n in this representation as $(n)_{\text{ALT}}$ and give the first few valid representations in Table 7.1 where $\bar{1}$ is used for -1.

n	1	2	3	4	5	6	7	8	9	10	11
$(n)_{\text{ALT}}$	1	10	100	$100\bar{1}$	1000	$100\bar{1}0$	$1000\bar{1}$	10000	$10\bar{1}001$	$100\bar{1}00$	$1000\bar{1}0$

Table 7.1: Examples of alternating representations.

We use the following `Walnut` code to implement the four conditions of the alternating representation:

```
reg altInclude1 {-1,0,1} "(0*|0*1.*)":
reg altExclude1 {-1,0,1} ".*(10*1|[-1]0*[-1]).*":
reg altExclude2 {-1,0,1} ".*(1[-1]|[-1]1).*":
reg altInclude2 {-1,0,1} "(0*|0*10*|0*[-1]0*|.*(100+[-1]|[-1]00+1)0*)":
def alt "$altInclude1(s) & ~$altExclude1(s) & ~$altExclude2(s)
& $altInclude2(s)":
```

The result is an automaton of 12 states that checks whether an input over the alphabet $\{-1, 0, 1\}$ is alternating, and is illustrated in Figure 7.1.

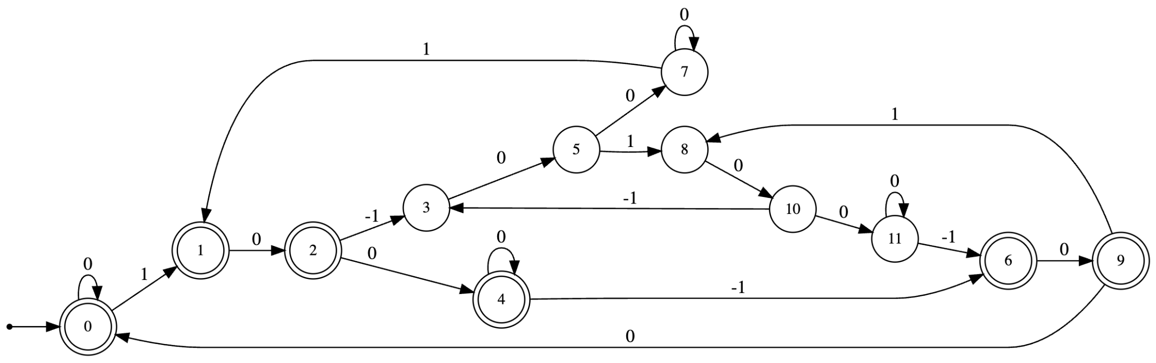


Figure 7.1: DFA for the alternating conditions.

7.1.2 Hajnal's Even Representation

The even representation requires three conditions:

1. the most significant nonzero term is positive,
2. only positions indexed with even numbers, such as e_2, can have nonzero terms, and
3. two adjacent nonzero terms cannot both be -1.

We denote a number n in this representation as $(n)_E$ and show valid representations in Table 7.2.

n	1	2	3	4	5	6	7	8	9	10	11
$(n)_E$	1	$10\bar{1}$	100	101	$10\bar{1}00$	$10\bar{1}01$	$1000\bar{1}$	10000	10001	$1010\bar{1}$	10100

Table 7.2: Examples of even representations.

```
reg evenInclude1 {-1,0,1} "(0*|0*1.*)":
reg evenInclude2 {-1,0,1} "0*|0*.(0.)*":
reg evenExclude {-1,0,1} ".*[-1]0*[-1].*":
def even "$evenInclude1(s) & $evenInclude2(s) & ~$evenExclude(s)":
```

This gives us a 5-state automaton to check the even condition, which is illustrated in Figure 7.2.

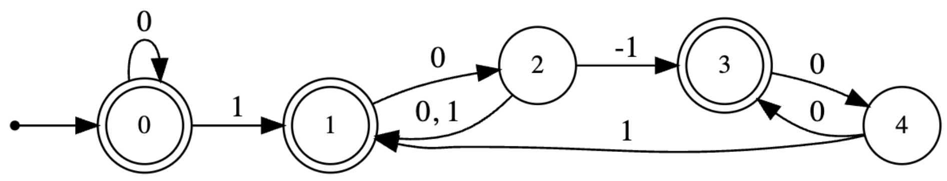


Figure 7.2: DFA for the even conditions.

7.1.3 Hajnal's Odd Representation

The odd representation adds an epsilon term to the sum in Eq. (1.1), therefore associating a string $e_t e_{t-1} \cdots e_2 \epsilon$, where $\epsilon \in \{-1, 0\}$, with a number n. The odd representation requires the string to meet three conditions:

1. the most significant nonzero term is positive,
2. only positions indexed with odd numbers (such as e_3) and the epsilon term are allowed to be nonzero, and
3. two adjacent nonzero terms cannot both be -1.

We denote a number n in this representation as $(n)_O$. We give in Table 7.3 the first few valid representations with the last digit in the representation being the ϵ term.

n	1	2	3	4	5	6	7	8	9	10
$(n)_O$	1	100	$10\bar{1}00$	$1000\bar{1}$	10000	$1010\bar{1}$	10100	$10\bar{1}0000$	$10\bar{1}010\bar{1}$	$10\bar{1}0100$

Table 7.3: Examples of odd representations.

We express the odd representation conditions in `Walnut` as follows. Notice we relax the third condition (required in [12]) slightly by limiting its application to only the string $e_t e_{t-1} \cdots e_2$ without the ϵ term.

```
reg oddInclude1 {-1,0,1} "(0*|0*1.*)":
reg oddInclude2 {-1,0,1} "0*|0*(.0)*":
reg oddExclude {-1,0,1} ".*[-1]0*[-1].*":
def odd "$oddInclude1(s) & $oddInclude2(s) & ~$oddExclude(s)":
```

This gives us a 5-state automaton to check the odd condition, which is illustrated in Figure 7.3.

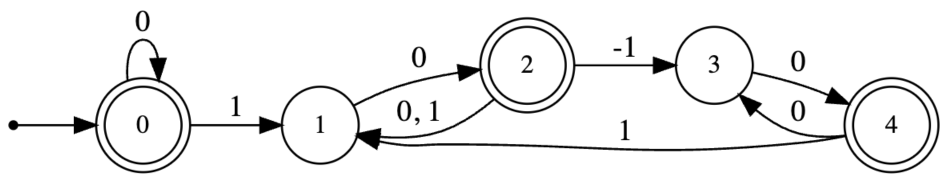


Figure 7.3: DFA for the odd conditions.

7.2 Proof of Completeness and Unambiguity

It now remains to use our technique to show that these representations proposed by Hajnal are all complete and unambiguous.

7.2.1 Proof Using `fcanon` as Converter

Our approach relies on the converter automaton `fcanon` with an added layer in order to process strings over $\{-1, 0, 1\}$. We construct such an automaton as follows. The idea is to use one automaton to "select" the positive digits of a representation, another one to "select" the negative digits, and then do an (implicit) subtraction to obtain the value of the representation.

```
reg posdigits {-1,0,1} {0,1} "([1,1]|[-1,0]|[0,0])*":
reg negdigits {-1,0,1} {0,1} "([-1,1]|[1,0]|[0,0])*":
def fcanon2 "?msd_fib Et,u,w,s $negdigits(x,t) & $posdigits(x,u)
& $fcanon(t,w) & $fcanon(u,s) & z+w=s":
```

This gives a 24-state automaton `fcanon2`, the analogue of `fcanon`, for doing the conversion.

Let us now check that the alternating representation of Hajnal is both complete and unambiguous.

```
eval altRepC "?msd_fib An Es $fcanon2(s,n) & $alt(s)":
# evaluates to TRUE, 4 ms
eval altRepU "?msd_fib ~En,s,t $alt(s) & $alt(t) & (~$same(s,t))
& $fcanon2(s,n) & $fcanon2(t,n)":
# evaluates to TRUE, 41 ms
```

Similarly, we can check the even and odd representations, as follows:

```
eval evenRepC "?msd_fib An Es $fcanon2(s,n) & $even(s)":
# evaluates to TRUE, 1 ms
eval evenRepU "?msd_fib ~En,s,t $even(s) & $even(t) & (~$same(s,t))
& $fcanon2(s,n) & $fcanon2(t,n)":
# evaluates to TRUE, 4 ms
eval oddRepC "?msd_fib An
(Es $fcanon2(s,n) & $odd(s)) | (Et $fcanon2(t,n+1) & $odd(t))":
# evaluates to TRUE, 7 ms
eval oddRepU "~En,s,t $odd(s) & $odd(t) & (~$same(s,t))
& $fcanon2(s,n) & $fcanon2(t,n)":
# evaluates to TRUE, 4 ms
```

This completes our proof that all three systems of Hajnal are complete and unambiguous.

Remark 11. We noticed, by testing the following, that this representation is also complete if $\epsilon \in \{1, 0\}$ instead of $\epsilon \in \{-1, 0\}$ as required in [12].

```
eval oddRepC1 "?msd_fib An
(Es $fcanon2(s,n) & $odd(s)) | (Et $fcanon2(t,n-1) & $odd(t))":
# evaluates to TRUE, 4 ms
```

7.2.2 Proof Using `fsc` as Converter

We can also check the completeness and unambiguity of the alternating representation as follows.

```
eval altRepC "?msd_neg_fib An n>=0 => Es $fsc(s,n) & $alt(s)":
# evaluates to TRUE, 9 ms

eval altRepU "?msd_neg_fib ~En,s,t $alt(s) & $alt(t) & (~$same(s,t))
& $fsc(s,n) & $fsc(t,n)":
# evaluates to TRUE, 83 ms
```

Notice that with `fsc` we do not need to construct an analogue like `fcanon2`, since the alternating representation (and the other two Hajnal representations) and the negaFibonacci representation use the same alphabet. On the other hand, we need to bound the `n` in the completeness evaluation statement to `n>=0` due to the number system in use, as we explained in Chapter 6.1. We use the following statements to evaluate the even and odd representations with `fsc`.

```
eval evenRepC "?msd_neg_fib An n>=0 => Es $fsc(s,n) & $even(s)":
# evaluates to TRUE, 7 ms
eval evenRepU "?msd_neg_fib ~En,s,t $even(s) & $even(t) & (~$same(s,t))
& $fsc(s,n) & $fsc(t,n)":
# evaluates to TRUE, 10 ms

eval oddRepC "?msd_neg_fib An n>=0 =>
(Es $fsc(s,n) & $odd(s)) | (Et $fsc(t,n+1) & $odd(t))":
# evaluates to TRUE, 12 ms
eval oddRepU "?msd_neg_fib ~En,s,t $odd(s) & $odd(t) & (~$same(s,t))
& $fsc(s,n) & $fsc(t,n)":
# evaluates to TRUE, 6 ms
```

7.3 Extending the Representations

Now we consider an interesting property of the alternating representation—it is easy to negate a number. All we need to do is to change the sign of each nonzero digit in a representation. For example, $(6)_{\text{ALT}} = 100\bar{1}0$ and the string $\bar{1}0010$ would evaluate to -6 according to Eq. (1.1). Notice the string $\bar{1}0010$ follows all but the first conditions in the definition of the alternating representation: the most significant nonzero term is positive. This condition is also present in the definitions for the even and odd representations. Therefore we are interested in the effect of omitting this condition from all three definitions. We show that the modified representations become complete and unambiguous for *all integers* instead of just the natural numbers. We modify the representation definition automata as follows.

```
def altInt "~$altExclude1(s) & ~$altExclude2(s) & $altInclude2(s)":
def evenInt "$evenInclude2(s) & ~$evenExclude(s)":
```

```
def oddInt "$oddInclude2(s) & ~$oddExclude(s)":
```

We prove that these three modified representations are complete for all integers as follows. Note that we no longer need the `n>=0` qualification.

```
eval altIntC "?msd_neg_fib An Es $fsc(s,n) & $altInt(s)":
# evaluates to TRUE, 3 ms
eval evenIntC "?msd_neg_fib An Es $fsc(s,n) & $evenInt(s)":
# evaluates to TRUE, 1 ms
eval oddIntC "?msd_neg_fib An Es
($fsc(s,n) & $oddInt(s)) | (Et $fsc(t,n+1) & $oddInt(t))":
# evaluates to TRUE, 5 ms
eval oddIntC1 "?msd_neg_fib An Es
($fsc(s,n) & $oddInt(s)) | (Et $fsc(t,n-1) & $oddInt(t))":
# evaluates to TRUE, 5 ms
```

Recall Remark 11, here we notice the same thing: the modified odd representation is complete with either $\epsilon \in \{-1, 0\}$ (the condition in [12]) or $\epsilon \in \{1, 0\}$. This is intriguing because negation via simply changing the signs of nonzero terms would require the alphabet of ϵ to be $\{-1, 0, 1\}$. For example, the negation of $(6)_O = 1010\bar{1}$, with the last digit $\epsilon = -1$, is $\bar{1}0\bar{1}01$ where the last digit shows $\epsilon = 1$. Therefore our intuition gained from the alternating representation example does not apply. Out of interest, we construct a DFA, displayed in Figure 7.4, which reveals the modified odd representation of -6 as $\bar{1}010100$ where $\epsilon = 0$.

```
def oddIntN6 "?msd_neg_fib $fsc(s,x) & $oddInt(s) & x+6=0":
```

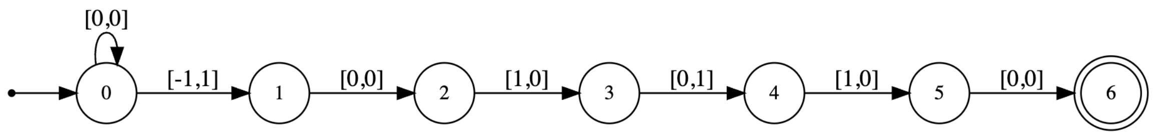


Figure 7.4: Automaton showing -6 in the modified odd representation.

Now we prove the unambiguity of the modified representations.

```
eval altIntU "?msd_neg_fib ~En,s,t $altInt(s) & $altInt(t)
& (~$same(s,t)) & $fsc(s,n) & $fsc(t,n)":
# evaluates to TRUE, 36 ms
eval evenIntU "?msd_neg_fib ~En,s,t $evenInt(s) & $evenInt(t)
& (~$same(s,t)) & $fsc(s,n) & $fsc(t,n)":
# evaluates to TRUE, 6 ms
eval oddIntU "?msd_neg_fib ~En,s,t $oddInt(s) & $oddInt(t) & (~$same(s,t))
& $fsc(s,n) & $fsc(t,n)":
# evaluates to TRUE, 5 ms
```

As all evaluations return `TRUE`, all three modified representations are complete and unambiguous for all integers.

Chapter 8

Representation for All Integers

In this chapter, we investigate two different ways to represent *all* integers (not just the natural numbers) using Fibonacci representations.

8.1 Alpert Representation: Using Digits -1, 0, and 1

Alpert [1] described a *far-difference representation* for Fibonacci numbers that writes *every integer* with a Fibonacci numeration system using the digits $-1, 0, 1$. In Alpert's system, the far-difference representation requires the string to have

1. at least three zeros between any two nonzero terms of the same sign, and
2. at least two zeros between any two nonzero terms of different signs.

We use $(n)_A$ to denote a number n in this representation; see examples in Table 8.1.

n	1	2	3	4	5	6	7	8	9	10	11
$(n)_A$	1	10	100	$100\bar{1}$	1000	$100\bar{1}0$	$1000\bar{1}$	10000	10001	$100\bar{1}00$	$1000\bar{1}0$

Table 8.1: Examples of Alpert's far-difference representations.

One nice feature of Alpert's system is that it is very easy to negate an integer: all we have to do is change the sign of each digit. This is reminiscent of our discussion in Chapter 7.3. We express the far-difference representation conditions in `Walnut` as follows.

```
reg exclude1 {-1,0,1} ".*([-1][-1]|[-1]0[-1]|[-1]00[-1]|11|101|1001).*":
reg exclude2 {-1,0,1} ".*([-1]1|1[-1]|10[-1]|[-1]01).*":
def alpert "~$exclude1(s) & ~$exclude2(s)":
```

This gives a 7-state automaton that checks the Alpert conditions, as illustrated in Figure 8.1.

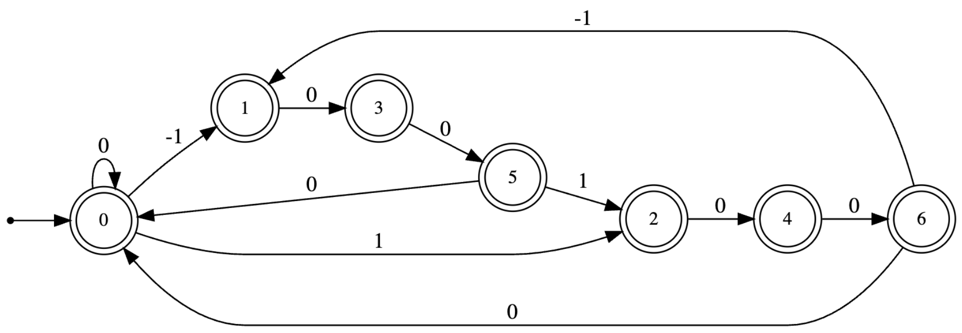


Figure 8.1: DFA for Alpert's far-difference representation.

As in the last sections, we check completeness and ambiguity with two different comparator automata. Using fcanon, we first construct fcanon2 as discussed in Chapter 7.2.1. Additionally, since we have to check positive and negative integers separately, we need an automaton fcanon2_neg that takes a string x over the alphabet $\{-1, 0, 1\}$ and a natural number $n \geq 0$ in parallel as inputs and accepts if and only if $[x]_F = -n$.

```
def fcanon2_neg "?msd_fib Et,u,w,s $negdigits(x,t) & $posdigits(x,u)
& $fcanon(t,w) & $fcanon(u,s) & z+s=w":
```

We can then prove the completeness and unambiguity of this system as follows.

```
eval farDiffC_pos "?msd_fib An Es $fcanon2(s,n) & $alpert(s)":
eval farDiffC_neg "?msd_fib An Es $fcanon2_neg(s,n) & $alpert(s)":
# both evaluate to TRUE, 3 ms

eval farDiffU_pos "?msd_fib ~En,s,t $alpert(s) & $alpert(t)
& (~$same(s,t)) & $fcanon2(s,n) & $fcanon2(t,n)":
```

```
eval farDiffU_neg "?msd_fib ~En,s,t $alpert(s) & $alpert(t)
& (~$same(s,t)) & $fcanon2_neg(s,n) & $fcanon2_neg(t,n)":
# both evaluate to TRUE, 9 ms
```

This concludes our proof, using `fcanon`, of the completeness and unambiguity of Alpert's conditions.

With the use of `fsc`, our proof statements in `Walnut` are significantly more straightforward. As discussed in Chapter 6.2, this is because both Alpert's and the negaFibonacci representations use the same alphabet and can represent the same range of numbers.

```
eval farDiffC "?msd_neg_fib An Es $fsc(s,n) & $alpert(s)":
eval farDiffU "?msd_neg_fib ~En,s,t $alpert(s) & $alpert(t)
& (~$same(s,t)) & $fsc(s,n) & $fsc(t,n)":
# both evaluate to TRUE, 17 ms
```

Thus we have easily verified the correctness of Alpert's conditions.

8.2 Bunder Representation: Using Negatively Indexed Fibonacci Numbers

Now we consider the completeness and unambiguity of Bunder's negaFibonacci system. Since this is the system underlying the converter automaton `fsc`, we can only use `fcanon` as the converter automaton for our proof of correctness. Recall that Bunder's system can be seen as a rule stating that (a) only the terms with odd indices are allowed to be positive and only the terms with even indices are allowed to be negative and (b) no two consecutive nonzero digits can appear. These conditions can be coded in regular expression as follows:

```
reg bunderEx1 {-1,0,1} ".*1.(..)*":
reg bunderEx2 {-1,0,1} ".*[-1](..)*":
reg bunderEx3 {-1,0,1} ".*((1[-1])|([-1]1)).*":
def bunder "~$bunderEx1(x) & ~$bunderEx2(x) & ~$bunderEx3(x)":
```

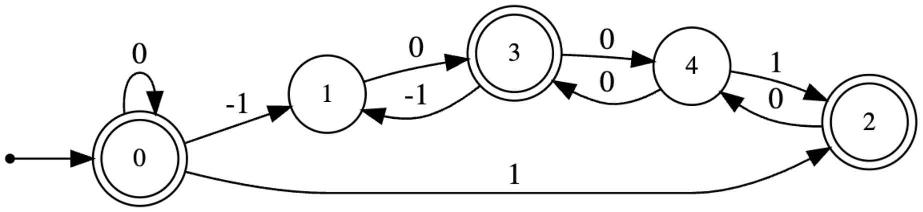


Figure 8.2: DFA for Bunder's negaFibonacci system.

which gives the automaton `bunder` in Figure 8.2.

Now we can check completeness and unambiguity using `fcanon` much as we did for Alpert's system, but there is a new wrinkle: negaFibonacci representations have an extra digit at the end, corresponding to the term a_1F_1, that must be taken care of. To do this we introduce a "shifter" automaton that shifts a representation one position to the right, and we use another automaton `lastbit`, constructed in regular expression below, to determine if the last bit of a representation is 1 or 0. The shifter is called `rshift` and is displayed in Figure 8.3. Notice that it is very similar to `lshift` discussed in Chapter 5.2.

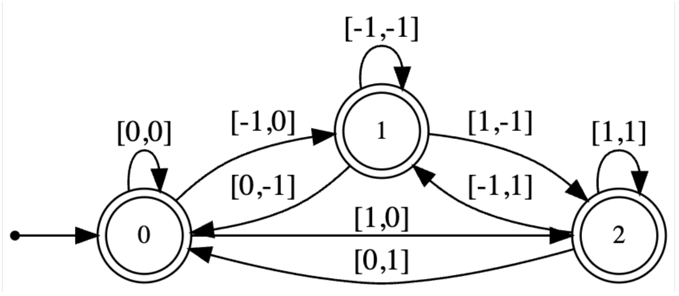


Figure 8.3: Right shifter automaton.

Then Bunder's representation can be verified to be complete and unambiguous, as follows:

```
reg lastbit {-1,0,1} {0,1} "([0,0]|[1,0]|[-1,0])*([1,1]|[0,0])":
def fcanon3 "?msd_fib Et,u,m $rshift(x,t) & $lastbit(x,u)
& $fcanon2(t,m) & z=m+u":
def fcanon3_neg "?msd_fib Et,u,m $rshift(x,t) & $lastbit(x,u)
& $fcanon2_neg(t,m) & z=m-u":
```

```
eval bunderC_pos "?msd_fib An Es $fcanon3(s,n) & $bunder(s)":
eval bunderC_neg "?msd_fib An Es $fcanon3_neg(s,n) & $bunder(s)":
# both evaluate to TRUE, 3 ms

eval bunderU_pos "?msd_fib ~En,s,t $bunder(s) & $bunder(t) & (~$same(s,t))
& $fcanon3(s,n) & $fcanon3(t,n)":
eval bunderU_neg "?msd_fib ~En,s,t $bunder(s) & $bunder(t) & (~$same(s,t))
& $fcanon3_neg(s,n) & $fcanon3_neg(t,n)":
# both evaluate to TRUE, 12 ms
```

Thus we have verified the correctness of Bunder's conditions.

Chapter 9

New Representations

In this chapter we look at complete and unambiguous Fibonacci representations that were not studied previously in the literature.

9.1 Maximum Dictionary Order Representation

Here we consider an entirely new Fibonacci representation based on dictionary order. We first introduce how strings are compared in dictionary order. Let $s = s_1 s_2 \cdots s_m$ and $t = t_1 t_2 \cdots t_n$ where $m \leq n$ be two strings. Let i such that $1 \leq i \leq m$ be the first position where $s_i \neq t_i$. If $s_i < t_i$, then $s < t$ in dictionary order; otherwise $s > t$. For example, $1\underline{0}11 < 1\underline{1}00$, but $10\underline{1}1 > 10\underline{0}1$. If there is no such position i, then either $s = t$ or s is a proper prefix of t. In this latter case we say $s < t$. For example, $110 = 110$ and $110 < 1100$.

Consider a representation of natural numbers by always choosing the *largest string representation in dictionary order* for every number. Since every number has a Fibonacci-based representation, the representation is complete. Since we choose only one Fibonacci-based representation for each number, the representation is unambiguous. We use $(n)_D$ to denote a number n in the maximum dictionary order representation. Representations of the first few numbers are given in Table 9.1.

We now show that

Theorem 12. *The set of largest Fibonacci representations in dictionary order forms a regular language.*

n	1	2	3	4	5	6	7	8	9	10	11
$(n)_D$	1	10	11	101	110	111	1010	1100	1101	1110	1111

Table 9.1: Examples of maximum dictionary order representations.

Proof. The idea is to construct a comparator DFA `dGreater` that can take two representations in parallel and decide if one is greater than the other, in dictionary order.

In order to take two representations in parallel, they would have to be the same length, and therefore the shorter one would have to be padded with leading zeros to make it the same length as the longer one. In this case, it is not hard to see that no automaton can do the needed comparison.

However, in our case, we can take advantage of the following fact: two Fibonacci representations for the same number cannot be of wildly different lengths.

Lemma 13. *The lengths of two Fibonacci-based representation strings for the same natural number differ by one at most (not counting leading zeros).*

Proof. Let s and t be two Fibonacci representations for a natural number m. Without loss of generality, assume that s is longer. Suppose the leading 1 digit of s corresponds to F_i. If s and t differ in length by more than one, then t is a sum of some F_j's where $j \leq i-2$. Now a classic identity on Fibonacci numbers states that $\sum_{0\leq j\leq n} F_j = F_{n+2} - 1$. Using this relation, we conclude that $\sum_{j=2}^{i-2} F_j = F_i - 2 < F_i$. Therefore s and t do not represent the same number. $\square$

Using this fact, it is indeed possible to compare two strings in dictionary order with a comparator automaton.

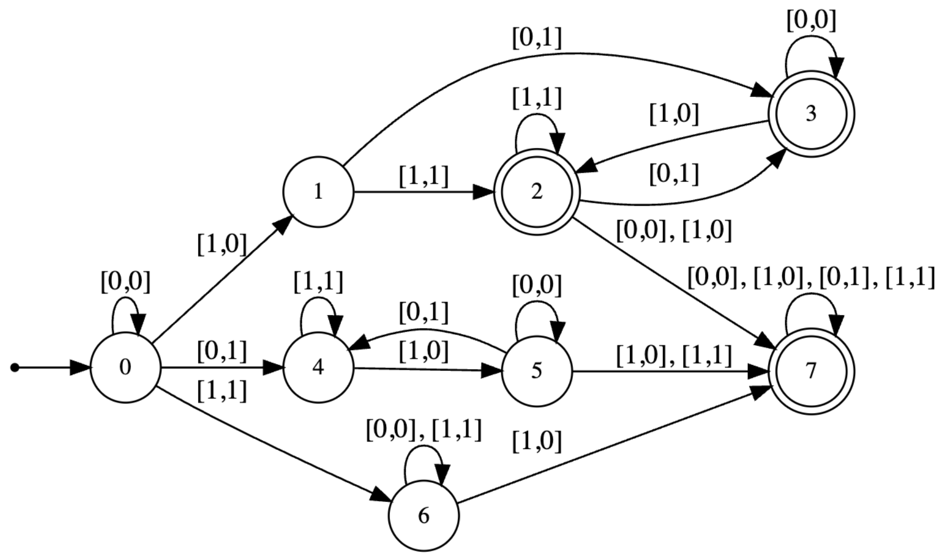


Figure 9.1: DFA for comparing strings in dictionary order.

It is shown in Fig. 9.1 and takes two inputs in parallel, s' and t'. Let s and t be s' and t' without leading zeros. The DFA `dGreater` accepts if and only if s is greater than t in dictionary order. We have three cases to consider: $|s| > |t|$, $|s| < |t|$, and $|s| = |t|$. We now discuss how the 8 states of `dGreater` relate to these 3 cases.

- State 0 is the initial state.
- State 1 is reached if $|s| > |t|$; that is, if s' starts with 1 and t' starts with 01.
- State 2 is reached when $|s| > |t|$, s ends in 1, and based on the inputs so far, t is a proper prefix of s therefore $s > t$.
- State 3 is reached when $|s| > |t|$, s ends in 0, and based on the inputs so far, t is a proper prefix of s therefore $s > t$.
- State 4 is reached when $|s| < |t|$ and t ends in 1, and based on the inputs so far, s is a proper prefix of t therefore $s < t$.
- State 5 is reached when $|s| < |t|$ and t ends in 0, and based on the inputs so far, s is a proper prefix of t therefore $s < t$.
- State 6 is reached when $|s| = |t|$ and, based on the inputs so far, we have $s = t$.

- State 7 is one of the accepting states. It is reached when we can identify a position i such that $s_i > t_i$. Additional symbols read, starting from this state, cannot change the comparison result.

It is now easy to verify that the transitions maintain the invariants corresponding to each state, and we leave this to the reader. □

Using the comparator automaton, we can build a DFA `dictOrder` that finds the maximum dictionary order representation for each natural number. We implement it in `Walnut` as follows.

```
def dictOrder "$fcanon(s,x) &
(At $fcanon(t,x) => ($dGreater(s,t)|$equal(s,t)))":
```

The automaton `dictOrder` takes two inputs in parallel: a number x in Zeckendorf representation and a string $s \in \{0,1\}^*$; and it only accepts if, out of all Fibonacci-based representations of x, the string s is the greatest based on dictionary order. It has 7 states and is depicted in Figure 9.2.

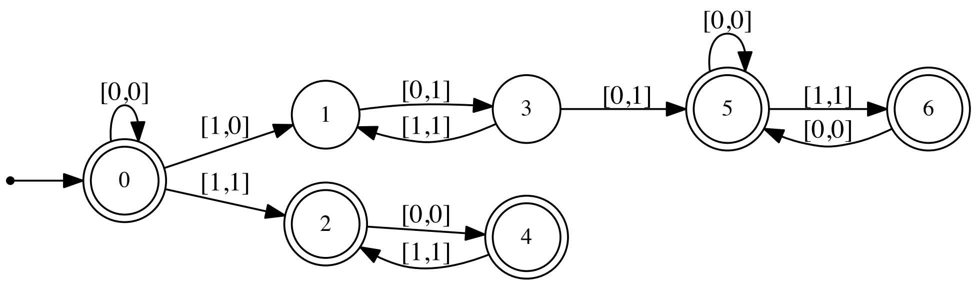


Figure 9.2: DFA for converting to dictionary order representation.

We show an example of using `dictOrder` to find the representation of a particular number: the following `Walnut` statement constructs a DFA displayed in Figure 9.3 and it shows $(18)_D = 11110$.

```
def dict18 "?msd_fib $dictOrder(s,x) & x=18":
```

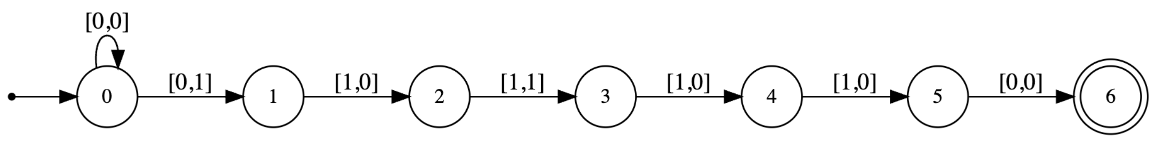


Figure 9.3: DFA showing 18 in maximum dictionary order representation.

9.2 Finding New Perfect Systems of Small Complexity via Exhaustive Search

We see that a Fibonacci-based representation of natural numbers can be represented by a language over the binary alphabet $\{0, 1\}$. If the language is regular, we can express it with a DFA and test its completeness and unambiguity in `Walnut`. For example, the Zeckendorf representation can be expressed as a 3-state DFA (counting in the sink state) and the Brown one, a 4-state DFA. Therefore we were curious about whether there exist other DFAs with a small number of states that can qualify as complete and unambiguous representations. We conducted an exhaustive search to find such automata and found a surprising number of them. If we allow up to 7 states, we found more than 28 new complete and unambiguous representations.[1] We present two interesting examples out of the seven new 6-state representations.

Theorem 14. *Let* $L = 0^*(\epsilon|1|10(\epsilon|0|1)1^*(01^+)^*(\epsilon|0))$*. Then* L *is complete and unambiguous.*

Proof. We use the following `Walnut` code:

```
reg one0sq {0,1} "0*(()|1|10(()|0|1)1*(01+)*(()|0))":
eval one0sqTestC "?msd_fib An Ex $one0sq(x) & $fcanon(x,n)":
eval one0sqTestU "?msd_fib ~En,x,y $one0sq(x) & $one0sq(y) & (~$equal(x,y))
& $fcanon(x,n) & $fcanon(y,n)":
```

Both returned `TRUE`. Here `one0sq` tests membership in L. □

We display the DFA `one0sq` accepting the language L in Figure 9.4. Notice this representation allows 1000 at the very beginning but no other consecutive 0's are allowed. This

[1]There could be more as the heuristics we used to trim our search tree can sometimes exclude eligible representations if, for two numbers m, n where $m < n$, the representation of m is longer than that of n.

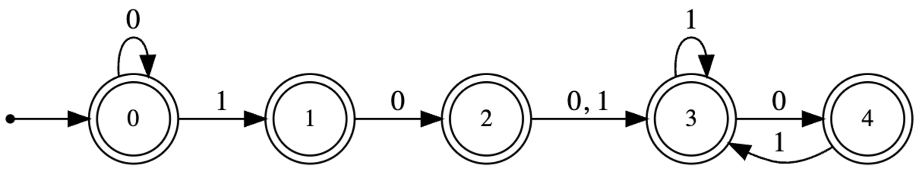


Figure 9.4: The DFA `one0sq`.

restriction on 00 blocks is very similar to Brown's lazy representation. In fact, Brown's representation can be expressed, in the form of a regular expression, as

$$0^*(\epsilon|11^*(01^+)^*(\epsilon|0)) = 0^*(\epsilon|1\cancel{|10(\epsilon|0|1)}1^*(01^+)^*(\epsilon|0)).$$

We can imagine that a new representation could be generated for allowing a block of 00 after the second 1, or the third, or after both the first and third 1, or the first and fourth, etc. This offers another construction of infinitely many perfect representations.

Theorem 15. *Let L be the language accepted by the DFA* `az`*. Then L is complete and unambiguous.*

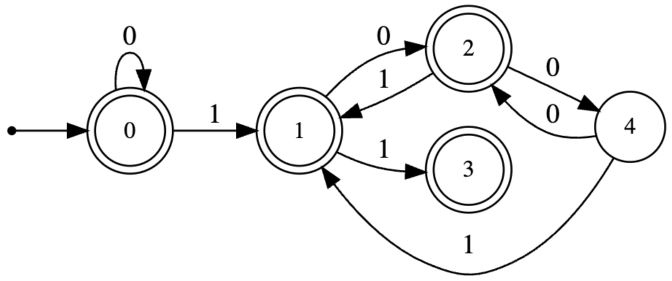


Figure 9.5: The DFA `az`.

Proof. We use the following `Walnut` code:

```
eval azTestC "?msd_fib An Ex $az(x) & $fcanon(x,n)":
eval azTestU "?msd_fib ~En,x,y $az(x) & $az(y) & (~$equal(x,y))
& $fcanon(x,n) & $fcanon(y,n)":
```

Both returned `TRUE`. Here `az` tests membership in L. □

The strings in L can end with a single 1 or the block 11 or an odd number of 0's, but not an even number of 0's. Additionally, the strings cannot contain the block "11" anywhere but the end. This restriction on "11" is reminiscent of the Zeckendorf representation.

Chapter 10

Representation Using Fibonacci Numbers With Positive and Negative Indices

In this chapter, we apply our approach to test representations that employ a sequence of Fibonacci numbers with both positive and negative indices.

10.1 Park Representation

Park et al.[21] showed that, given any integers r and n with $n \geq 2$, there exists a complete representation for natural numbers using Fibonacci numbers whose indices are not congruent to r modulo n. For arbitrary r and n, we do not believe that our approach is applicable. However, if we fix an r and an n, then we have a particular representation and we can use our framework to prove the completeness of it.

We focus on the case where $r = 0$ and $n = 3$, that is, the representation using only the Fibonacci numbers whose indices are not multiples of 3. We call it the Park representation for our discussion here.

Let $x = a_u \cdots a_1$ and $y = a_v \cdots a_{-1}$ be two strings of integer digits. Let $k \in \mathbb{Z}$ such that $v \leq k \leq u$, each $a_k \in \{0, 1\}$, and $a_k = 0$ when k is a multiple of 3. We define the value of x and y as a Park representation as follows:

$$[x + y]_P = [x]_{P^+} + [y]_{P^-} = \sum_{1 \leq i \leq u} a_i F_i \; + \sum_{v \leq j \leq -1} a_j F_j \tag{10.1}$$

We give a valid Park representation for each of the first few natural numbers in Table 10.1. We put a dot in place of a_0. We write x as is, in "most-significant-digit" first format, before the dot; we write y in reverse, in "least-significant-digit" first format, after the dot. Note that the Park representation is complete but not unambiguous, unlike the representations discussed above. Therefore the representations listed in Table 10.1 may not be the only valid ones.

n	0	1	2	3	4	5	6	7	8	9
$(n)_P$	$1.0\bar{1}$	.1	1.1	11.1	1010.	1011.	1011.1	$11000.0\bar{1}$	11000.	11000.1

Table 10.1: Examples of the Park representations.

Each Park representation has two parts: one associated with the positively indexed Fibonacci numbers, the string x, and one with the negatively indexed Fibonacci numbers, the string y. We handle the two parts separately because x and y are constricted by different sets of conditions.

For the positively indexed part of the Park representation, the only condition we want to encode is that a 0 must appear for positions whose indices are multiples of 3. We do so as follows.

```
reg parkPos {0,1} "0*(()|1(0..)*|1(.0.)*.)":
```

Since the string is over $\{0,1\}$, it is natural to use `fcanon` as the converter automaton, but there is one issue: Park's representation makes use of the term F_1, and therefore the string x has an extra digit a_1 at the end. We need to shift the string one position to the right before it can be processed by `fcanon`. We use the shifter automaton `rshiftBin` displayed in Figure 10.1; this is the binary version of the shifter in Figure 8.3. We also need an automaton `lastbitBin`, constructed in regular expression below, to determine whether a_1 is 0 or 1. Then we can construct the converter `fcanonParkPos` for the positively indexed part of the Park representation. The automaton `fcanonParkPos` takes a string x over the alphabet $\{0,1\}$ and a number z in the Fibonacci representation in parallel as input and accepts iff $[x]_{P+} = z$.

```
reg lastbitBin {0,1} {0,1} "([0,0]|[1,0])*([1,1]|[0,0])":
def fcanonParkPos "?msd_fib Et,u,m $rshiftBin(x,t) & $lastbitBin(x,u)
& $fcanon(t,m) & z=m+u":
```

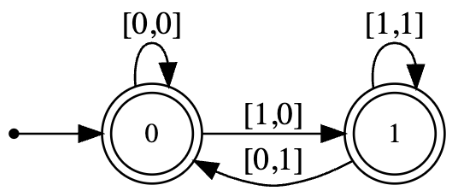


Figure 10.1: Binary right shifter automaton.

For the negatively indexed part of the Park representation, it can be seen as a string adhering to the following set of conditions:

1. only the terms with odd indices are allowed to be positive,
2. only the terms with even indices are allowed to be negative, and
3. 0 must appear where the index is a multiple of 3.

We represent these conditions as follows.

```
reg parkNegEx1 {-1,0,1} ".*1.(..)*":
reg parkNegEx2 {-1,0,1} ".*[-1](..)*":
reg parkNegIn {-1,0,1} "0*(()|([-1]|1)(0..)*|([-1]|1)(.0.)*.)":
def parkNeg "~$parkNegEx1(x) & ~$parkNegEx2(x) & $parkNegIn(x)":
```

Notice we can use the converter `fcanon3` developed in Chapter 8.2 to evaluate the negatively indexed part of the Park representation. With this, we can define the following converter automaton for Park representations.

```
def fcanonPark "?msd_fib Eu,v $fcanonParkPos(x,u) & $fcanon3(y,v)
& u+v=z":
```

This gives an 81-state automaton `fcanonPark` that takes the following in parallel as input:

1. a string x over the alphabet $\{0, 1\}$,
2. a string y over $\{-1, 0, 1\}$, and

3. a natural number z in the Fibonacci representation.

The automaton `fcanonPark` accepts if and only if $[x+y]_P = z$. Now we are equipped to prove the completeness of the Park representation via the following evaluation.

```
eval parkC "?msd_fib An Es,t $fcanonPark(s,t,n) & $parkPos(s)
& $parkNeg(t)":
# evaluates to TRUE, 36 ms
```

Thus we have proved that the Park representation is complete.

10.2 Anderson Representation: Completeness for Pairs of Numbers

In this section we examine one more proof of completeness that can be simplified with our approach, however the completeness here is for pairs of numbers instead of individual numbers as we have seen so far. Anderson [2] used the *Extended Fibonacci Zeckendorf* (EZ) representation in which integers are expressed as sums of non-consecutive Fibonacci numbers without restriction on the signs of the subscripts. For example, let $m \geq 0$, $n > 0$, and a be integers, and let

$$a = \sum_{0 \leq i \leq m} c_i F_i \; + \sum_{-n \leq j \leq -1} c_j F_j, \tag{10.2}$$

then $(a)_{\text{EZ}} = c_m c_{m-1} \cdots c_1 c_0 | c_{-1} \cdots c_{-n+1} c_{-n}$ where $|$ is placed after the coefficient of F_0. Now let b be an integer and $\phi = \frac{1+\sqrt{5}}{2}$. Anderson proved that, if $b\phi + a > 0$, there exists an EZ representation of b such that

$$b = \sum_{0 \leq i \leq m} c_i F_{i+1} \; + \sum_{-n \leq j \leq -1} c_j F_{j+1} \tag{10.3}$$

or $(b)_{\text{EZ}} = c_m c_{m-1} \cdots c_1 c_0 c_{-1} | c_{-2} \cdots c_{-n+1} c_{-n}$. To clearly present how a and b are related in Anderson's theorem, we show how the coefficients correspond to the Fibonacci numbers in Table 10.2.

To prove Anderson's theorem, our first step is to assert conditions about a and b with automata. Let s, t, u, v be strings such that $(a)_{\text{EZ}} = s|t^R$ and $(b)_{\text{EZ}} = u|v^R$. Note that w^R is the reversal of a string or word w. We consider the following aspects.

	F_{m+1}	F_m	$\cdots$	F_2	F_1	F_0	F_{-1}	F_{-2}	$\cdots$	F_{-n+1}	F_{-n}
$(a)_{\text{EZ}}$	0	c_m	$\cdots$	c_2	c_1	c_0	c_{-1}	c_{-2}	$\cdots$	c_{-n+1}	c_{-n}
$(b)_{\text{EZ}}$	c_m	c_{m-1}	$\cdots$	c_1	c_0	c_{-1}	c_{-2}	c_{-3}	$\cdots$	c_{-n}	0

Table 10.2: Example of a pair of numbers following Anderson's theorem.

- The strings s and u should follow the rule of the Zeckendorf representation and we use the automaton `greedy` defined below to test them. This is the binary alphabet version of the one defined in Chapter 6.1.

```
reg greedyExclude {0,1} ".*11.*":
def greedy "~$greedyExclude(s)":
```

- The strings t and v should follow the Bunder conditions; we can test them using `bunder` defined in the Chapter 8.2. Note that the Bunder conditions here combined with the Zeckendorf conditions above guarantee that no consecutive Fibonacci numbers are used in $(a)_{\text{EZ}}$ or $(b)_{\text{EZ}}$.
- The string s is u shifted one position to the right and we can check that using the automaton `rshiftBin` defined in Chapter 10.1.
- The string v is t "shifted" one position to the right; however, there is a complication caused by the Bunder representation format. Suppose a coefficient c is 1, when it is the coefficient for F_{-1} or any other Fibonacci numbers with an odd negative index, c is expressed as 1 in the Bunder representation. But when c is the coefficient for Fibonacci numbers with an even negative index, it is expressed as -1. Therefore we build the shifter `rshiftPGA` displayed in Figure 10.2 to handle this particular right shift.
- The last digit in u should be the same as the last digit in t. We verify this with the use of `lastbit` defined in Chapter 8.2 and `lastbitBin` defined in Chapter 10.1.

With these considerations, we build the automaton `pgaPair`; it takes strings $s, t, z \in \{0,1\}^*$ and $u, v \in \{-1, 0, 1\}^*$ in parallel and accepts if and only if $s|t^R$ and $u|v^R$ are valid EZ representations and they relate as described in Anderson's theorem.

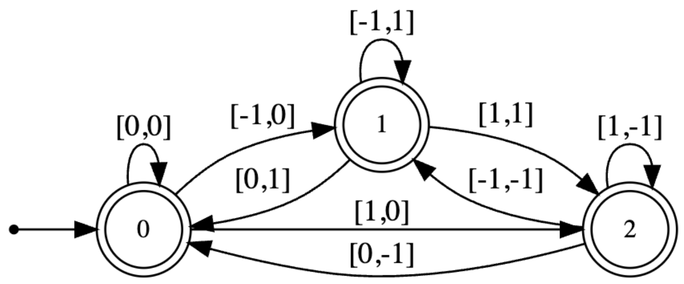


Figure 10.2: The automaton `rshiftPGA`.

```
def pgaPair "$greedy(s) & $greedy(u) & $bunder(t) & $bunder(v)
& $rshiftBin(u,s) & $rshiftPGA(t,v) & $lastbitBin(u,z) & $lastbit(t,z) ":
```

Now we consider the converters we need to evaluate EZ representations. We again use $(a)_{\mathrm{EZ}} = s|t^R$ as an example. The string s can be handled by `fcanon` once it is shifted two positions to the right; we can accomplish this via `rshiftBin` and catches whether F_1 should be included via `lastbitBin`. The following gives a 7-state automaton `fcanonF0` which is an analogue of `fcanon`.

```
def fcanonF0 "?msd_fib Em,t,u,y $rshiftBin(x,y) & $rshiftBin(y,t)
& $lastbitBin(y,u) & $fcanon(t,m) & z=m+u":
```

We can evaluate t using `fcanon3` and `fcanon3_neg` which are devised in Chapter 8.2. With these, we are equipped to build converter automata for $(a)_{\mathrm{EZ}}$. If $a \geq 0$, then we have two cases to consider depending on whether t evaluates to a positive number; each case is evaluated differently and they combine to form a converter for $(a)_{\mathrm{EZ}}$ as follows.

```
def fcanonPGA1 "?msd_fib Ex,y $fcanonF0(s,x) & $fcanon3(t,y) & z=x+y":
def fcanonPGA2 "?msd_fib Ex,y $fcanonF0(s,x) & $fcanon3_neg(t,y) & z+y=x":
def fcanonPGA_pos "$fcanonPGA1(s,t,z) | $fcanonPGA2(s,t,z)":
```

If $a < 0$, then t must evaluate to a negative number and we build a converter for $(a)_{\mathrm{EZ}}$ as follows.

```
def fcanonPGA_neg "?msd_fib Ex,y $fcanonF0(s,x) & $fcanon3_neg(t,y)
& z+x=y":
```

The automata `fcanonPGA_pos` and `fcanonPGA_neg` both take

1. a string $s \in \{0,1\}^*$,
2. a string $t \in \{-1,0,1\}^*$, and
3. a number z in the Fibonacci representation in parallel as input.

The automaton `fcanonPGA_pos` accepts iff s and t evaluate to z. On the other hand, `fcanonPGA_neg` accepts iff s and t evaluate to $-z$.

One last thing to consider is calculating $b\phi$ in `Walnut`. We use the approach presented in [27, Chap. 10.11]. The `Walnut` code below gives the 7-state automaton `phin` displayed in Figure 10.3. It takes two numbers b and c, both in the Fibonacci representation, in parallel and accepts iff $c = b\phi$.

```
reg lshiftBin {0,1} {0,1} "([0,0]|[0,1][1,1]*[1,0])*":
def phin "?msd_fib (c=0 & b=0) | Ex $lshiftBin(b-1,x) & c=x+1":
```

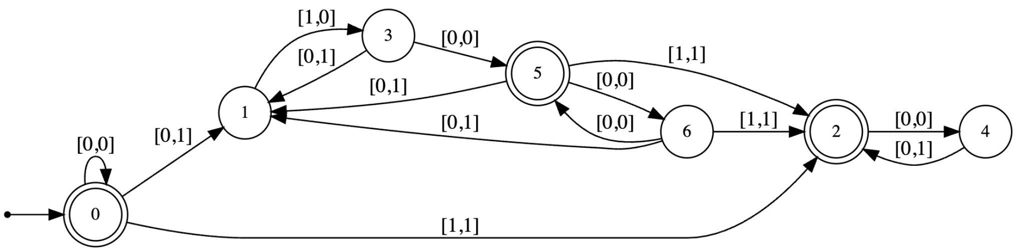


Figure 10.3: The automaton `phin`.

Now we prove Anderson's theorem. Since Anderson's condition is $b\phi + a > 0$, we have three cases to consider. If $a, b > 0$, we evaluate as follows.

```
eval anderson1 "?msd_fib Aa,b Es,t,u,v,z $pgaPair(s,t,u,v,z)
& $fcanonPGA_pos(s,t,a) & $fcanonPGA_pos(u,v,b)":
# evaluates to TRUE, 31 ms
```

If $a \geq 0$, $b \leq 0$, and $b\phi + a > 0$, we evaluate as follows.

```
eval anderson2 "?msd_fib Aa,b,c (a>=0 & $phin(b,c) & a>=c+1) => Es,t,u,v,z
$pgaPair(s,t,u,v,z) & $fcanonPGA_pos(s,t,a) & $fcanonPGA_neg(u,v,b)":
# evaluates to TRUE, 38 ms
```

If $a \leq 0$, $b \geq 0$, and $b\phi + a > 0$, we evaluate as follows.

```
eval anderson3 "?msd_fib Aa,b,c (b>=0 & $phin(b,c) & c>=a+1) => Es,t,u,v,z
$pgaPair(s,t,u,v,z) & $fcanonPGA_neg(s,t,a) & $fcanonPGA_pos(u,v,b)":
# evaluates to TRUE, 32 ms
```

As all statements evaluate to `TRUE`, we verified Anderson's theorem.

We show an example of finding the set of qualifying coefficients given a pair of a and b. Say $a = 18$ and $b = -5$. We can see that $b\phi + a > 0$. Since $a \geq 0$ and $b \leq 0$, we modify the evaluation statement `anderson2` into the following definition.

```
def anderson18N5 "?msd_fib $pgaPair(s,t,u,v,z) & $fcanonPGA_pos(s,t,a)
& $fcanonPGA_neg(u,v,b) & a=18 & b=5":
```

The automaton `anderson18N5` is displayed in Figure 10.4 and the order of the arguments is a, b, s, t, u, v, z. This reveals $(18)_{\mathrm{EZ}} = 1010|0010001$ and $(-5)_{\mathrm{EZ}} = 10100|0\bar{1}000\bar{1}$ as qualifying EZ representations.

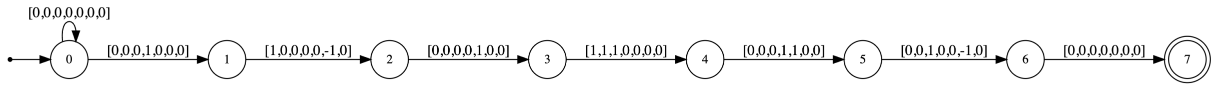


Figure 10.4: Automaton showing valid EZ representations for 18 and -5.

Chapter 11

Final Remarks

In the chapters above, we present a framework for mechanically testing, via automata theory, the completeness and unambiguity of Fibonacci-based representations, also called numeration systems. We illustrate its effectiveness and versatility by applying it to a diverse set of previously published and newly discovered representations. Additionally, if a representation is proved to be complete, we describe an algorithm, of $O(\log n)$ complexity, to find a representation for any particular number n.

11.1 Open Problem

As discussed in Chapter 4.1, given an automatic numeration system, our framework presents an algorithm checking its completeness, running in exponential time in the number of states of the automaton depicting the system. Therefore a natural question is to ask whether there exists a way of assessing the completeness of a given automatic numeration system in polynomial time. This problem remains open for us.

References

[1] H. Alpert. Differences of multiple Fibonacci numbers. *INTEGERS*, 9(#A57), 2009. (electronic).

[2] P. G. Anderson. Extended Fibonacci Zeckendorf theory. In *Proceedings of the Sixteenth International Conference on Fibonacci Numbers and Their Applications*, pages 15–21. The Fibonacci Association, 2014.

[3] J. Berstel. An exercise on Fibonacci representations. *RAIRO Inform. Théor. App.*, 35:491–498, 2001.

[4] J. Berstel, A. Lauve, C. Reutenauer, and F. V. Saliola. *Combinatorics on words: Christoffel words and repetitions in words*, volume 27 of *CRM Monograph Series.* American Mathematical Society, 2009.

[5] J. L. Brown, Jr. A new characterization of the Fibonacci numbers. *Fibonacci Quart.*, 3(1):1–8, 1965.

[6] J. L. Brown, Jr. Unique representation of integers as sums of distinct Lucas numbers. *Fibonacci Quart.*, 7:243–252, 1969.

[7] V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and p-recognizable sets of integers. *Bull. Belgian Math. Soc.*, 1:191–238, 1994. Corrigendum, *Bull. Belg. Math. Soc.* **1** (1994), 577.

[8] J. R. Büchi. Weak secord-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960. Reprinted in S. Mac Lane and D. Siefkes, eds., *The Collected Works of J. Richard Büchi*, Springer-Verlag, 1990, pp. 398–424.

[9] M. W. Bunder. Zeckendorf representations using negative Fibonacci numbers. *Fibonacci Quart.*, 30:111–115, 1992.

[10] L. Carlitz, R. Scoville, and V.E. Hoggatt, Jr. Fibonacci representations of higher order. *Fibonacci Quart.*, 10:43–69, 94, 1972.

[11] U. Güntzer and M. Paul. Jump interpolation search trees and symmetric binary numbers. *Inform. Process. Lett.*, 26(4):193–204, 1987.

[12] P. Hajnal. A short note on numeration systems with negative digits allowed. *Bull. Inst. Combin. Appl.*, 97:54–66, 2023.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[14] A. F. Horadam. Zeckendorf representations of positive and negative integers by Pell numbers. In G. E. Bergum, A. N. Philippou, and A. F. Horadam, editors, *Applications of Fibonacci Numbers*, volume 5, pages 305–316. Kluwer, 1993.

[15] S. Labbé and J. Lepšová. A numeration system for Fibonacci-like Wang shifts. In T. Lecroq and S. Puzynina, editors, *WORDS 2021*, volume 12847 of *Lecture Notes in Computer Science*, pages 104–116. Springer-Verlag, 2021.

[16] C. G. Lekkerkerker. Voorstelling van natuurlijke getallen door een som van getallen van Fibonacci. *Simon Stevin*, 29:190–195, 1952.

[17] A. Monnerot-Dumaine. The Fibonacci word fractal. 2009. hal-00367972.

[18] H. Mousavi. Automatic theorem proving in `Walnut`. Arxiv preprint arXiv:1603.06017 [cs.FL], available at http://arxiv.org/abs/1603.06017, 2016.

[19] H. Mousavi, L. Schaeffer, and J. Shallit. Decision algorithms for Fibonacci-automatic words, I: basic results. *RAIRO Inform. Théor. App.*, 50:39–66, 2016.

[20] A. Ostrowski. Bemerkungen zur Theorie der Diophantischen Approximationen. *Abh. Math. Sem. Hamburg*, 1:77–98,250–251, 1922. Reprinted in *Collected Mathematical Papers*, Vol. 3, pp. 57–80.

[21] H. Park, B. Cho, D. Cho, Y. D. Cho, and J. Park. Representation of integers as sums of Fibonacci and Lucas numbers. *Symmetry*, 12(10):1625, 2020.

[22] H. Prodinger. On binary representations of integers with digits −1, 0, 1. *INTEGERS*, 0(#A08), 2000. (electronic).

[23] G. W. Reitwiesner. Binary arithmetic. In *Advances in Computers*, volume 1, pages 231–308. Elsevier, 1960.

[24] N. Robbins. Fibonacci partitions. *Fibonacci Quart.*, 34:306–313, 1996.

[25] J. Shallit. A primer on balanced binary representations. `https://cs.uwaterloo.ca/~shallit/Papers/bbr.pdf`, 1992.

[26] J. Shallit. Robbins and Ardila meet Berstel. *Inform. Process. Lett.*, 167:106081, 2021.

[27] J. Shallit. *The Logical Approach to Automatic Sequences: Exploring Combinatorics on Words with* `Walnut`, volume 482 of *London Math. Soc. Lecture Notes Series*. Cambridge University Press, 2022.

[28] J. Shallit and S. L. Shan. A general approach to proving properties of Fibonacci representations via automata theory. In *Electronic Proceedings in Theoretical Computer Science*, volume 386, pages 228–242. Open Publishing Association, 2023.

[29] J. Shallit, S. L. Shan, and K. H. Yang. Automatic sequences in negative bases and proofs of some conjectures of shevelev. *RAIRO-Theor. Inf. Appl.*, 57:4, 2023.

[30] N. J. A. Sloane et al. The on-line encyclopedia of integer sequences, 2022. Available at `https://oeis.org`.

[31] E. Zeckendorf. Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas. *Bull. Soc. Roy. Liège*, 41:179–182, 1972.