

Chapter 5: Recursive and Recursively Enumerable Languages

In this chapter, we will study a universal programming language, which we will use to define the recursive and recursively enumerable languages. We will see that the context-free languages are a proper subset of the recursive languages, that the recursive languages are a proper subset of the recursively enumerable languages, and that there are languages that are not recursively enumerable.

Copyright © 2003–5 Alley Stoughton

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

The \LaTeX source of these slides, the associated book, and the distribution of the Forlan toolset are available on the WWW at <http://people.cis.ksu.edu/~stough/forlan/>.

Chapter 5: Introduction (Cont.)

Furthermore, we will learn that there are problems, like the halting problem (the problem of determining whether a program P halts when run on an input w), or the problem of determining if two grammars generate the same language, that can't be solved by programs.

Chapter 5: Introduction (Cont.)

Furthermore, we will learn that there are problems, like the halting problem (the problem of determining whether a program P halts when run on an input w), or the problem of determining if two grammars generate the same language, that can't be solved by programs.

Traditionally, one uses Turing machines for the universal programming language. Turing machines are finite automata that manipulate infinite tapes. Although Turing machines are very appealing in some ways, they are rather far-removed from conventional programming languages, and are hard to build and reason about.

Chapter 5: Introduction (Cont.)

Furthermore, we will learn that there are problems, like the halting problem (the problem of determining whether a program P halts when run on an input w), or the problem of determining if two grammars generate the same language, that can't be solved by programs.

Traditionally, one uses Turing machines for the universal programming language. Turing machines are finite automata that manipulate infinite tapes. Although Turing machines are very appealing in some ways, they are rather far-removed from conventional programming languages, and are hard to build and reason about.

Instead, we will work with a variant of the programming language Lisp. This programming language will have the same power as Turing machines, but it will be much easier to program in this language than with Turing machines.

Chapter 5: Introduction (Cont.)

Furthermore, we will learn that there are problems, like the halting problem (the problem of determining whether a program P halts when run on an input w), or the problem of determining if two grammars generate the same language, that can't be solved by programs.

Traditionally, one uses Turing machines for the universal programming language. Turing machines are finite automata that manipulate infinite tapes. Although Turing machines are very appealing in some ways, they are rather far-removed from conventional programming languages, and are hard to build and reason about.

Instead, we will work with a variant of the programming language Lisp. This programming language will have the same power as Turing machines, but it will be much easier to program in this language than with Turing machines.

An “implementation” of our language (or of Turing machines) on a real computer will run out of resources on some programs.

Section 5.1: Programs and Recursive and Recursively Enumerable Languages

We will work with a variant of the programming language Lisp, with the following properties and features:

- The language is statically scoped, dynamically typed, deterministic and functional.

Section 5.1: Programs and Recursive and Recursively Enumerable Languages

We will work with a variant of the programming language Lisp, with the following properties and features:

- The language is statically scoped, dynamically typed, deterministic and functional.
- The language's types are infinite precision integers, booleans, formal language symbols and arbitrary-length strings, arbitrary-length lists, and an error type (whose only value is **error**). There are the usual functions for going back and forth between integers and symbols.

Section 5.1: Programs and Recursive and Recursively Enumerable Languages

We will work with a variant of the programming language Lisp, with the following properties and features:

- The language is statically scoped, dynamically typed, deterministic and functional.
- The language's types are infinite precision integers, booleans, formal language symbols and arbitrary-length strings, arbitrary-length lists, and an error type (whose only value is **error**). There are the usual functions for going back and forth between integers and symbols.
- A program consists of one or more function definitions. The last function definition of a program is called its principal function. This function should take in some number of strings, and return a boolean.

(5.1) Programming Language Syntax

The set **Prog** of *programs* is a subset of **Syn**^{*}, where the alphabet **Syn** consists of:

- the digits 0–9;
- the letters a–z and A–Z; and
- the symbols `<space>`, `<newline>`, `<openPar>` and `<closPar>`.

When we present programs, however, we typically substitute a blank for `<space>`, a newline for `<newline>`, “(” for `<openPar>`, and “)” for `<closPar>`.

Little detail about our programming language will be given in the lecture slides. For example, here is a program that tests whether its argument string is nonempty:

```
(defun test (x) (not (equal (size x) 0)))
```

(5.1) Programming Language Semantics

Given an $n \in \mathbb{N}$, the function

$\text{run}_n \in \mathbf{Str} \times \mathbf{Str} \uparrow n \rightarrow \{\mathbf{true}, \mathbf{false}, \mathbf{error}, \mathbf{nonterm}\}$, where

$\mathbf{Str} \uparrow n$ consists of all n -tuples of strings, returns the following answer when called with arguments $(P, (x_1, \dots, x_n))$:

- If P is not a program, or the principal function of P doesn't have exactly n arguments, then it returns **error**.

Eventually, the book will contain a formal definition of the run_n functions.

(5.1) Programming Language Semantics

Given an $n \in \mathbb{N}$, the function

$\text{run}_n \in \mathbf{Str} \times \mathbf{Str} \uparrow n \rightarrow \{\mathbf{true}, \mathbf{false}, \mathbf{error}, \mathbf{nonterm}\}$, where $\mathbf{Str} \uparrow n$ consists of all n -tuples of strings, returns the following answer when called with arguments $(P, (x_1, \dots, x_n))$:

- If P is not a program, or the principal function of P doesn't have exactly n arguments, then it returns **error**.
- Otherwise, if running P 's principal function with arguments x_1, \dots, x_n results in the value **true** or **false** being returned, then it returns this value.

Eventually, the book will contain a formal definition of the run_n functions.

(5.1) Programming Language Semantics

Given an $n \in \mathbb{N}$, the function

$\text{run}_n \in \mathbf{Str} \times \mathbf{Str} \uparrow n \rightarrow \{\mathbf{true}, \mathbf{false}, \mathbf{error}, \mathbf{nonterm}\}$, where

$\mathbf{Str} \uparrow n$ consists of all n -tuples of strings, returns the following answer when called with arguments $(P, (x_1, \dots, x_n))$:

- If P is not a program, or the principal function of P doesn't have exactly n arguments, then it returns **error**.
- Otherwise, if running P 's principal function with arguments x_1, \dots, x_n results in the value **true** or **false** being returned, then it returns this value.
- Otherwise, if running P 's principal function with these arguments causes some other value to be returned, then it returns **error**.

Eventually, the book will contain a formal definition of the run_n functions.

(5.1) Programming Language Semantics

Given an $n \in \mathbb{N}$, the function

$\text{run}_n \in \mathbf{Str} \times \mathbf{Str} \uparrow n \rightarrow \{\mathbf{true}, \mathbf{false}, \mathbf{error}, \mathbf{nonterm}\}$, where $\mathbf{Str} \uparrow n$ consists of all n -tuples of strings, returns the following answer when called with arguments $(P, (x_1, \dots, x_n))$:

- If P is not a program, or the principal function of P doesn't have exactly n arguments, then it returns **error**.
- Otherwise, if running P 's principal function with arguments x_1, \dots, x_n results in the value **true** or **false** being returned, then it returns this value.
- Otherwise, if running P 's principal function with these arguments causes some other value to be returned, then it returns **error**.
- Otherwise, running P 's principal function with these arguments never terminates, and it returns **nonterm**.

Eventually, the book will contain a formal definition of the run_n functions.

(5.1) Checking Whether a String is a Program

It is possible to write a function in our programming language that checks whether a string is a valid program whose principal function has a given number of arguments.

This will involve writing a function for parsing programs, where parse trees will be represented using lists.

(5.1) Interpreters Written in our Language

With some effort, it is possible to write a function in our programming language that acts as an *interpreter*. It takes in a string w and a list of strings (x_1, \dots, x_n) . If w is not a program whose principal function has n arguments, then the interpreter returns **error**. Otherwise, it simulates the running of w with input (x_1, \dots, x_n) , returning what w returns, if w returns a boolean, and returning **error**, if w returns something else. Of course, w may also run forever, in which case the interpreter will also run forever.

(5.1) Interpreters (Cont.)

We can also write a function in our programming language that acts as an *incremental* interpreter. Like an ordinary interpreter, it takes in a string w and a list of strings (x_1, \dots, x_n) . If w is not a program whose principal function has n arguments, then the incremental interpreter returns **error**. Otherwise, it carries out a fixed number of steps of the running of w with input (x_1, \dots, x_n) . If w has returned a boolean by this point, then the incremental interpreter returns this boolean. If w has returned something else by this point, then the incremental interpreter returns **error**. But if w hasn't yet terminated, the incremental interpreter returns a function that, when called, will continue the incremental interpretation process.

(5.1) Total Programs and the Languages Accepted by Programs

Given $n \in \mathbb{N}$, we say that a program P is n -total iff, for all $x_1, \dots, x_n \in \mathbf{Str}$, $\mathbf{run}_n(P, (x_1, \dots, x_n)) \in \{\mathbf{true}, \mathbf{false}\}$.

(5.1) Total Programs and the Languages Accepted by Programs

Given $n \in \mathbb{N}$, we say that a program P is *n-total* iff, for all $x_1, \dots, x_n \in \mathbf{Str}$, $\mathbf{run}_n(P, (x_1, \dots, x_n)) \in \{\mathbf{true}, \mathbf{false}\}$.

A string w is *accepted by* a program P iff $\mathbf{run}_1(P, w) = \mathbf{true}$, i.e., iff running P with input w results in \mathbf{true} being returned. (We write the 1-tuple whose single component is w as w .)

(5.1) Total Programs and the Languages Accepted by Programs

Given $n \in \mathbb{N}$, we say that a program P is n -total iff, for all $x_1, \dots, x_n \in \mathbf{Str}$, $\mathbf{run}_n(P, (x_1, \dots, x_n)) \in \{\mathbf{true}, \mathbf{false}\}$.

A string w is *accepted by* a program P iff $\mathbf{run}_1(P, w) = \mathbf{true}$, i.e., iff running P with input w results in \mathbf{true} being returned. (We write the 1-tuple whose single component is w as w .)

The language *accepted by* a program P ($L(P)$) is

$$\{w \in \mathbf{Str} \mid w \text{ is accepted by } P\},$$

if this set of strings is a language (i.e., $\bigcup\{\mathbf{alphabet}(w) \mid w \in \mathbf{Str} \text{ and } w \text{ is accepted by } P\}$ is finite); otherwise $L(P)$ is undefined.

(5.1) Total Programs and the Languages Accepted by Programs

Given $n \in \mathbb{N}$, we say that a program P is n -total iff, for all $x_1, \dots, x_n \in \mathbf{Str}$, $\mathbf{run}_n(P, (x_1, \dots, x_n)) \in \{\mathbf{true}, \mathbf{false}\}$.

A string w is *accepted* by a program P iff $\mathbf{run}_1(P, w) = \mathbf{true}$, i.e., iff running P with input w results in \mathbf{true} being returned. (We write the 1-tuple whose single component is w as w .)

The language *accepted* by a program P ($L(P)$) is

$$\{w \in \mathbf{Str} \mid w \text{ is accepted by } P\},$$

if this set of strings is a language (i.e., $\bigcup\{\mathbf{alphabet}(w) \mid w \in \mathbf{Str} \text{ and } w \text{ is accepted by } P\}$ is finite); otherwise $L(P)$ is undefined.

For example, if P 's principal function takes in more than one argument, then $L(P) = \emptyset$, but if P 's principal function takes in a single argument but always returns \mathbf{true} , then $L(P)$ is undefined.

(5.1) Recursive and Recursively Enumerable Languages

We say that a language L is:

- *recursive* iff $L = L(P)$ for some 1-total program P ;
- *recursively enumerable* (r.e.) iff $L = L(P)$ for some program P .

(When we say $L = L(P)$, this means that $L(P)$ is defined, and L and $L(P)$ are equal.)

We define

$$\mathbf{RecLan} = \{ L \in \mathbf{Lan} \mid L \text{ is recursive} \},$$

$$\mathbf{RELan} = \{ L \in \mathbf{Lan} \mid L \text{ is recursively enumerable} \}.$$

(5.1) Recursive and Recursively Enumerable Languages

We say that a language L is:

- *recursive* iff $L = L(P)$ for some 1-total program P ;
- *recursively enumerable* (r.e.) iff $L = L(P)$ for some program P .

(When we say $L = L(P)$, this means that $L(P)$ is defined, and L and $L(P)$ are equal.)

We define

$$\mathbf{RecLan} = \{ L \in \mathbf{Lan} \mid L \text{ is recursive} \},$$

$$\mathbf{RELan} = \{ L \in \mathbf{Lan} \mid L \text{ is recursively enumerable} \}.$$

Hence $\mathbf{RecLan} \subseteq \mathbf{RELan}$. Because \mathbf{Prog} is countably infinite, we have that \mathbf{RecLan} and \mathbf{RELan} are countably infinite, so that $\mathbf{RELan} \subsetneq \mathbf{Lan}$. Later we will see that $\mathbf{RecLan} \subsetneq \mathbf{RELan}$.

(5.1) Recursive and Recursively Enumerable Languages (Cont.)

Proposition 5.1.1

For all $L \in \mathbf{Lan}$, L is recursive iff there is a program P such that, for all $w \in \mathbf{Str}$:

- if $w \in L$, then $\mathbf{run}_1(P, w) = \mathbf{true}$;
- if $w \notin L$, then $\mathbf{run}_1(P, w) = \mathbf{false}$.

(5.1) Recursive and Recursively Enumerable Languages (Cont.)

Proposition 5.1.1

For all $L \in \mathbf{Lan}$, L is recursive iff there is a program P such that, for all $w \in \mathbf{Str}$:

- if $w \in L$, then $\mathbf{run}_1(P, w) = \mathbf{true}$;
- if $w \notin L$, then $\mathbf{run}_1(P, w) = \mathbf{false}$.

Proof. (“only if” direction) Since L is recursive, $L = L(P)$ for some 1-total program P . Suppose $w \in \mathbf{Str}$. There are two cases to show.

Suppose $w \in L$. Since $L = L(P)$, we have that $\mathbf{run}_1(P, w) = \mathbf{true}$.

Suppose $w \notin L$. Since $L = L(P)$, we have that $\mathbf{run}_1(P, w) \neq \mathbf{true}$. But P is 1-total, and thus $\mathbf{run}_1(P, w) = \mathbf{false}$.

(5.1) Recursive and Recursively Enumerable Languages (Cont.)

Proof (cont.). (“if” direction) To see that P is 1-total, suppose $w \in \mathbf{Str}$. Since $w \in L$ or $w \notin L$, we have that $\mathbf{run}_1(P, w) \in \{\mathbf{true}, \mathbf{false}\}$.

Let $X = \{w \in \mathbf{Str} \mid w \text{ is accepted by } P\}$ (so far, we don’t know that X is a language). We will show that $L = X$.

Suppose $w \in L$. Then $\mathbf{run}_1(P, w) = \mathbf{true}$, so that $w \in X$.

Suppose $w \in X$, so that $\mathbf{run}_1(P, w) = \mathbf{true}$. If $w \notin L$, then $\mathbf{run}_1(P, w) = \mathbf{false}$ —contradiction. Thus $w \in L$.

Since $L = X$, we have that X is a language. Thus $L(P)$ is defined and is equal to X . Hence $L = L(P)$, finishing the proof that L is recursive. \square

(5.1) Recursive and Recursively Enumerable Languages (Cont.)

Proposition 5.1.2

For all $L \in \mathbf{Lan}$, L is recursively enumerable iff there is a program P such that, for all $w \in \mathbf{Str}$,

$$w \in L \text{ iff } \mathbf{run}_1(P, w) = \mathbf{true}.$$

(5.1) Recursive and Recursively Enumerable Languages (Cont.)

Proposition 5.1.2

For all $L \in \mathbf{Lan}$, L is recursively enumerable iff there is a program P such that, for all $w \in \mathbf{Str}$,

$$w \in L \text{ iff } \mathbf{run}_1(P, w) = \mathbf{true}.$$

Proof. (“only if” direction) Since L is recursively enumerable, $L = L(P)$ for some program P . Suppose $w \in \mathbf{Str}$.

Suppose $w \in L$. Since $L = L(P)$, we have that $\mathbf{run}_1(P, w) = \mathbf{true}$.

Suppose $\mathbf{run}_1(P, w) = \mathbf{true}$. Thus $w \in L(P) = L$.

(5.1) Recursive and Recursively Enumerable Languages (Cont.)

Proof (cont.). (“if” direction) Let $X = \{w \in \mathbf{Str} \mid w$ is accepted by $P\}$ (so far, we don’t know that X is a language). We will show that $L = X$.

Suppose $w \in L$. Then $\mathbf{run}_1(P, w) = \mathbf{true}$, so that $w \in X$. Suppose $w \in X$. Then $\mathbf{run}_1(P, w) = \mathbf{true}$, so that $w \in L$.

Since $L = X$, we have that X is a language. Thus $L(P)$ is defined and is equal to X . Hence $L = L(P)$, completing the proof that L is recursively enumerable. \square

(5.1) Relationship Between the Context-Free and the Recursive Languages

Theorem 5.1.3

The context-free languages are a proper subset of the recursive languages: $\text{CFLan} \subsetneq \text{RecLan}$.

Proof. To see that every context-free language is recursive,

To see that not every recursive language is context-free, let $L =$

□

(5.1) Relationship Between the Context-Free and the Recursive Languages

Theorem 5.1.3

The context-free languages are a proper subset of the recursive languages: $\text{CFLan} \subsetneq \text{RecLan}$.

Proof. To see that every context-free language is recursive, let L be a context-free language. Thus there is a grammar G such that $L = L(G)$. With some work, we can write and prove the correctness of a program P that implements our algorithm (see Section 4.3) for checking whether a string is generated by a grammar. Thus L is recursive.

To see that not every recursive language is context-free, let $L =$

□

(5.1) Relationship Between the Context-Free and the Recursive Languages

Theorem 5.1.3

The context-free languages are a proper subset of the recursive languages: $\mathbf{CFLan} \subsetneq \mathbf{RecLan}$.

Proof. To see that every context-free language is recursive, let L be a context-free language. Thus there is a grammar G such that $L = L(G)$. With some work, we can write and prove the correctness of a program P that implements our algorithm (see Section 4.3) for checking whether a string is generated by a grammar. Thus L is recursive.

To see that not every recursive language is context-free, let $L = \{0^n 1^n 2^n \mid n \in \mathbb{N}\}$. In Section 4.10, we learned that L is not context-free. And it is easy to write a program P that tests whether a string is in L . Thus L is recursive. \square