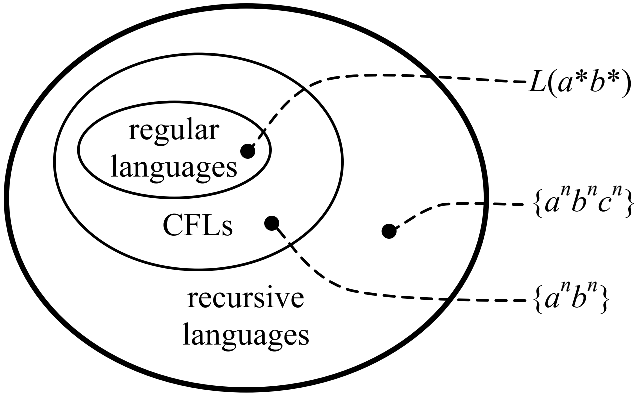
Chapter Eighteen: Uncomputability

Review: Computability

- A language is recursive if and only if it is L(*M*) of some *total* TM *M.*
- A function is (Turing) computable if and only if a *total* TM computes it.
- But we have:
  - For every language *L* we can define a corresponding function, such as $f(x) = 1$ if $x \in L$, 0 if $x \notin L$
  - For every function *f* we can define a corresponding language, such as $L = \{x\#y \mid y = f(x)\}$
- Therefore, *L* is recursive if and only if it is (Turing) computable.
- Church-Turing Thesis: *Anything an Algorithm can do a TM can do, and vice versa.*

The Church-Turing Thesis gives a definition of computability, like a border surrounding the algorithmically solvable problems.



Beyond that border is a wilderness of uncomputable problems. This is one of the great revelations of twentieth-century mathematics: the discovery of simple problems whose algorithmic solution would be very useful but is forever beyond us.

Outline

- 18.1 Decision and Recognition Methods
- 18.2 The Language L_u
- 18.3 The Halting Problems
- 18.4 Reductions Proving a Language Is Recursive
- 18.5 Reductions Proving a Language is Not Recursive
- 18.6 Rice's Theorem
- 18.7 Enumerators
- 18.8 Recursively Enumerable Languages
- 18.9 Languages That Are Not RE
- 18.10 Language Classifications Revisited
- 18.11 Grammars and Comnputability
- 18.12 Oracles
- 18.13 Mathematical Uncomputabilities

Switching To Java-Like Syntax

- In this chapter we switch from using Turing machines to using a Java-like syntax
- All the following ideas apply to any Turing-equivalent formalism
- Java-like syntax is easier to read than TMs
- It is just a different way of stating an algorithm and we know: for every algorithm we have a TM, and vice versa (Church-Turing Thesis)
- Note, this is not real Java; no limitations
- In particular, no bounds on the length of a string or the size of an integer

Decision Methods

- Total TMs correspond to *decision methods* in our Java-like notation
- A *decision method* takes a **`String`** parameter and returns a boolean value
- Another way of saying computable: it always returns, and does not run forever.
- Example, $\{ax \mid x \in \Sigma^*\}$:

```
boolean ax(String p) {
  return (p.length()>0 && p.charAt(0)=='a');
}
```

Decision Method Examples

- {}:

```
boolean emptySet(String p) {
  return false;
}
```

- Σ^*:

```
boolean sigmaStar(String p) {
  return true;
}
```

- As with TMs, the language accepted is *L*(**`m`**):
  - *L*(**`emptySet`**) = {}
  - *L*(**`sigmaStar`**) = Σ^*

Recursive Languages

- Previous definition: *L* is a recursive language if and only if it is *L*(*M*) for some total TM *M*
- New definition: *L* is a recursive language if and only if it is *L*(`m`) for some decision method `m`
- Recursive Language = (Turing) Decidable Language

Recognition Methods

- For methods that might run forever, a broader term
- A recognition method takes a **`String`** parameter and either returns a boolean value *or runs forever*
- A decision method is a special kind of recognition method, just as a total TM is a special kind of TM

Recursively Enumerable Languages

- Previous definition: *L* is a recursively enumerable language if and only if it is *L*(*M*) for some TM *M*
- New definition: *L* is a recursively enumerable language if and only if it is *L*(**`m`**) for some recognition method **`m`**
- Recursively Enumerable Language = (Turing) Recognizable Language

$\{a^nb^nc^n\}$ Recognition Method

```
boolean anbncn1(String p) {
  String as = "", bs = "", cs = "";
  while (true) {
    String s = as+bs+cs;
    if (p.equals(s)) return true;
    as += 'a'; bs += 'b'; cs += 'c';
  }
}
```

- Highly inefficient, but we don’t care about that
- We do care about termination; this recognition method loops forever if the string is not accepted
- It demonstrates only that $\{a^nb^nc^n\}$ is RE; we know it is recursive, so there is a decision method for it…

$\{a^nb^nc^n\}$ Decision Method

```
boolean anbncn2(String p) {
  String as = "", bs = "", cs = "";
  while (true) {
    String s = as+bs+cs;
    if (s.length()>p.length()) return false;
    else if (p.equals(s)) return true;
    as += 'a'; bs += 'b'; cs += 'c';
  }
}
```

- L(**`anbncn1`**) = L(**`anbncn2`**) = $\{a^nb^nc^n\}$
- But **`anbncn2`** is a *decision method*, showing that the language is recursive and not just RE

Universal Java Machine

- A universal TM performs a simulation to decide whether the given TM accepts the given string
- It is possible to implement the same kind of thing in Java; a **`run`** method like this:

```
/**
 * run(p, in) takes a String ‘p’ which is the text
 * of a recognition method, and a String ‘in’ which is
 * the input for that method.  We compile the method,
 * run it on the given parameter string, and return
 * whatever result it returns.  (If it does not
 * return, neither do we.)
 */
boolean run(String p, String in) {
  ... // don’t care about the details here

}
```

`run` Examples

- **`sigmaStar("abc")`** returns true, so the **`run`** in this fragment would return true:

```
String s = "boolean sigmaStar(String p) {return true;}";
run(s,"abc");
```

- **`ax("ba")`** returns false, so the **`run`** in this fragment would return false:

```
String s =
  "boolean ax(String p) {                          " +
  "  return (p.length()>0 && p.charAt(0)=='a'); " +
  "}                                               ";
run(s,"ba");
```

`run` Examples, Continued

- `anbncn1("abbc")` runs forever, so the `run` in this fragment would never return:

```
String s =
  "boolean anbncn1(String p) {               " +
  "  String as = \"\", bs = \"\", cs = \"\"; " +
  "  while (true) {                          " +
  "    String s = as+bs+cs;                  " +
  "    if (p.equals(s)) return true;         " +
  "    as += 'a'; bs += 'b'; cs += 'c';      " +
  "  }                                       " +
  "}                                         ";
run(s,"abbc");
```

☞ ‘run’ is a recognition method!

Outline

- 18.1 Decision and Recognition Methods
- 18.2 The Language L_u
- 18.3 The Halting Problems
- 18.4 Reductions Proving a Language Is Recursive
- 18.5 Reductions Proving a Language is Not Recursive
- 18.6 Rice's Theorem
- 18.7 Enumerators
- 18.8 Recursively Enumerable Languages
- 18.9 Languages That Are Not RE
- 18.10 Language Classifications Revisited
- 18.11 Grammars and Comnputability
- 18.12 Oracles
- 18.13 Mathematical Uncomputabilities

The Perils Of Infinite Computation

```
int j = 0;
for (int i = 0; i < 100; j++) {
  j += f(i);
}
```

- You run a program, and wait… and wait…
- You ask, “Is this stuck in an infinite loop, or is it just taking a long time?”
- No sure way for a person to answer such questions
- No sure way for a computer to find the answer for you…

The Language L_u

- $L_u = L($`run`$) = \{($`p`,`in`$) \mid$ `p` is a recognition method and `in` $\in L($`p`$)\}$
- (Remember *u* for *universal*)
- A corresponding language for universal TMs:
  $L_u = \{m\#x \mid m$ encodes a TM and x is a string it accepts$\}$
- We have a recognition method for it, so we know L_u is RE
- Is it recursive?

Is L_u Recursive?

- That is, is it possible to write a *decision* method with this specification:

```
/**
 * shortcut(p,in) returns true if run(p,in) would
 * return true, and returns false if run(p,in)
 * would return false or run forever.
 */
boolean shortcut(String p, String in) {
  ...
}
```

- Just like the **`run`** method, but does not run forever, even when **`run(p,in)`** would

Example

- For example, the **`shortcut`** in this fragment:

```
String x =
  "boolean anbncn1(String p) {               " +
  "  String as = \"\", bs = \"\", cs = \"\"; " +
  "  while (true) {                          " +
  "    String s = as+bs+cs;                  " +
  "    if (p.equals(s)) return true;         " +
  "    as += 'a'; bs += 'b'; cs += 'c';      " +
  "  }                                       " +
  "}                                         ";
shortcut(x,"abbc")
```

- It would return false, even though **`anbncn1("in")`** would run forever

Is This Possible?

- Presumably, **`shortcut`** would have to simulate the input program as **`run`** does
- But it would have to detect infinite loops
- Some are easy enough to detect:

  ```
  while(true) {}
  ```
- A program might even be clever enough to reason about the nontermination of **`anbncn1`**
- It would be very useful to have a debugging tool that could reliably alert you to infinite computations

The Bad News

- No such **`shortcut`** method exists and we can prove it!
- Our proof is by contradiction:
  - Assume by way of contradiction that L_u is recursive, so some implementation of **`shortcut`** exists
  - Then we could use it to implement this…

`nonSelfAccepting`

```
/**
 * nonSelfAccepting(p) returns false if run(p,p)
 * would return true, and returns true if run(p,p)
 * would return false or run forever.
 */
boolean nonSelfAccepting(String p) {
  return !shortcut(p,p);
}
```

- This determines what the given program would decide, given itself as input, then it returns the opposite
- So *L*(**`nonSelfAccepting`**) is the set of recognition methods that do not accept themselves

`nonSelfAccepting` Example

```
nonSelfAccepting(
  "boolean sigmaStar(String p) {return true;}"
);
```

- **`sigmaStar("boolean sigmaStar`**…**`")`** returns true: **`sigmaStar`** accepts everything, so it certainly accepts itself
- So it is self-accepting, and **`nonSelfAccepting`** returns false

`nonSelfAccepting` Example

```
nonSelfAccepting(
  "boolean ax(String p) {                         " +
  "  return (p.length()>0 && p.charAt(0)=='a');  " +
  "}                                              "
);
```

- **`ax("boolean ax...")`** returns false: **`ax`** accepts everything starting with **`a`**, but its own source code starts with **`b`**
- So it is not self-accepting, and **`nonSelfAccepting`** returns true

Back to the Proof

- We assumed by way of contradiction that **`shortcut`** could be implemented
- Using it, we showed an implementation of **`nonSelfAccepting`**
- Now comes the tricky part: what happens if we call **`nonSelfAccepting`**, giving it itself as input?
- We can easily arrange to do this:

Does **`nonSelfAccepting`** Accept Itself?

```
boolean nonSelfAccepting(String p) {
  return !shortcut(p,p);
};

String s = "boolean nonSelfAccepting(p) { " +
           "  return !shortcut(p,p);       " +
           "}                              ”;

nonSelfAccepting(s);
```

- Now consider:
  - shortcut(“nonSelfAccepting…”,”nonSelfAccepting…”) = true, but
  - nonSelfAccepting(“nonSelfAccepting…”) = false
  - Contradiction, not possible
- Or
  - shortcut(“nonSelfAccepting…”,”nonSelfAccepting…”) = false, but
  - nonSelfAccepting(“nonSelfAccepting…”) = true
  - Contradiction, not possible
- These are the only two outcomes because shortcut is a decision method by assumption.

Proof Summary

- We assumed by way of contradiction that **`shortcut`** could be implemented
- Using it, we showed an implementation of **`nonSelfAccepting`**
- We showed that applying **`nonSelfAccepting`** to itself results in a contradiction
- By contradiction, no program satisfying the specifications of **`shortcut`** exists
- In other words…

Theorem 18.2

L_u is not recursive.

- Our first example of a problem that is outside the borders of computability:
  - L_u is not *recursive*
  - The **`shortcut`** function is not *computable*
  - The machine-*M*-accepts-string-*x* property is not *decidable*
- This implies: No total TM can be a universal TM

Outline

- 18.1 Decision and Recognition Methods
- 18.2 The Language L_u
- 18.3 The Halting Problems
- 18.4 Reductions Proving a Language Is Recursive
- 18.5 Reductions Proving a Language is Not Recursive
- 18.6 Rice's Theorem
- 18.7 Enumerators
- 18.8 Recursively Enumerable Languages
- 18.9 Languages That Are Not RE
- 18.10 Language Classifications Revisited
- 18.11 Grammars and Comnputability
- 18.12 Oracles
- 18.13 Mathematical Uncomputabilities

Another Example

- Consider this recognition method:

```
/**
 * haltsRE(p,in) returns true if run(p,in) halts.
 * It just runs forever if run(p,in) runs forever.
 */
boolean haltsRE(String p, String in) {
  run(p,in);
  return true;
}
```

- It defines an RE language…

The Language L_h

- L_h = L(**`haltsRE`**) = {(**`p`**,**`in`**) | **`p`** is a recognition method that halts on **`in`**}
- (Remember *h* for *halting*)
- A corresponding language for universal TMs:
  $L_h = \{m\#x \mid m$ encodes a TM that halts on $x\}$
- We have a recognition method for it, so we know L_h is RE
- Is it recursive?

Is L_h Recursive?

- That is, is it possible to write a *decision* method with this specification:

```
/**
 * halts(p,in) returns true if run(p,in) halts, and
 * returns false if run(p,in) runs forever.
 */
boolean halts(String p, String in) {
  ...
}
```

- Just like the **`haltsRE`** method, but does not run forever, even when **`run(p,in)`** would

More Bad News

- From our results about L_u you might guess that L_h is not going to be recursive either
- Intuitively, the only way to tell what **`p`** will do when run on **`n`** is to simulate it
- If that runs forever, we won't get an answer
- But how do we know there isn't some other way of determining whether **`p`** halts, a way that doesn't involve actually running it?
- Proof is by contradiction: assume L_h is recursive, so an implementation of **`halts`** exists
- The we can use it to implement…

`narcissist`

```
/**
 * narcissist(p) returns true if run(p,p) would
 * run forever, and runs forever if run(p,p) would
 * halt.
 */
boolean narcissist(String p) {
  if (halts(p,p)) while(true) {}
  else return true;
}
```

- This halts (returning true) if and only if program `p` will contemplate itself forever
- So *L*(`narcissist`) is the set of recognition methods that run forever, given themselves as input
- Recall:

  -
    ```
    /**
     * halts(p,in) returns true if run(p,in) halts, and
     * returns false if run(p,in) runs forever.
     */
    ```

Back to the Proof

- We assumed by way of contradiction that **`halts`** could be implemented
- Using it, we showed an implementation of **`narcissist`**
- Now comes the tricky part: what happens if we call **`narcissist`**, giving it itself as input?
- We can easily arrange to do this:

Is `narcissist` a Narcissist?

```
narcissist(
  "boolean narcissist(p) {            " +
  "  if (halts(p,p)) while(true) {} " +
  "  else return true;                 " +
  "}                                   "
)
```

- Now consider:
  - halts(“narcissist…”,”narcissist…”) = true, but
  - narcissist(“narcissist…”) runs forever.
  - Contradiction
- Or
  - halts(“narcissist…”,”narcissist…”) = false , but
  - narcissist(“narcissist…”) halts and returns true.
  - Contradiction
- These are the only possible outcomes because halts is a decision method by assumption.

Proof Summary

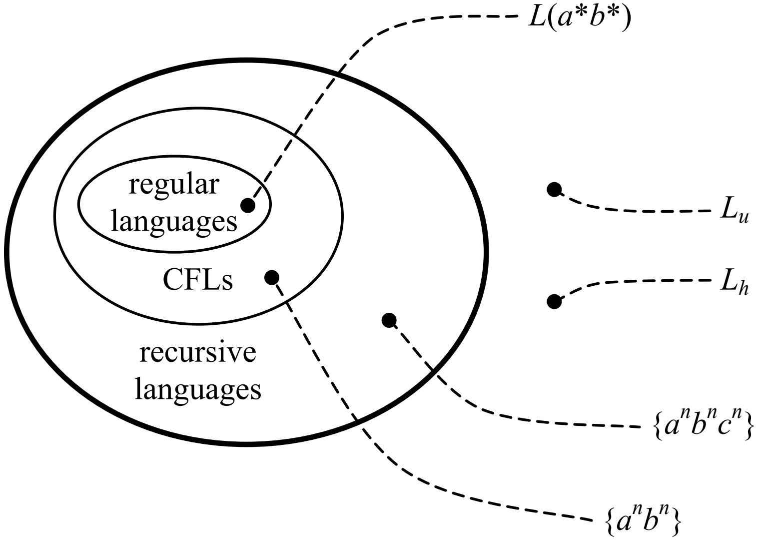
- We assumed by way of contradiction that `halts` could be implemented
- Using it, we showed an implementation of `narcissist`
- We showed that applying `narcissist` to itself results in a contradiction
- By contradiction, no program satisfying the specifications of `halts` exists
- In other words…

Theorem 18.3

L_h is not recursive.

- A classic undecidable problem: a *halting problem*
- Many variations:
  - Does a program halt on a given input?
  - Does it halt on any input?
  - Does it halt on every input?
- It would be nice to have a program that could check over your code and warn you about all possible infinite loops
- Unfortunately, it is impossible: the halting problem in all these variations, is undecidable

The Picture So Far



- The non-recursive languages don't stop there
- There are uncountably many languages beyond the computability border