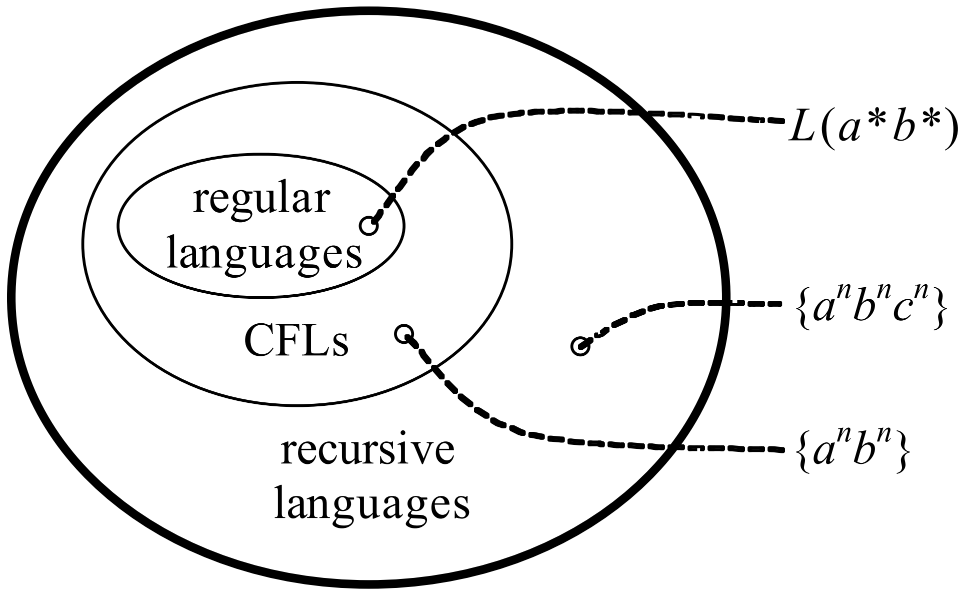
Chapter Eighteen: Uncomputability

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
- 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
- (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`**
- These definitions are equivalent because Java is Turing-equivalent

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

$\{a^n b^n c^n\}$ 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 $\{a^n b^n c^n\}$ is RE; we know it is recursive, so there is a decision method for it…

$\{a^n b^n c^n\}$ 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**) = $\{a^n b^n c^n\}$
- But **anbncn2** is a *decision method*, showing that the language is recursive and not just RE

RE Languages

- Previous definition: *L* is a recursively enumerable (RE) language if and only if it is *L*(*M*) for some TM *M*
- New definition: *L* is an RE language if and only if it is *L*(**`m`**) for some recognition method **`m`**
- These definitions are equivalent because Java is Turing-equivalent

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) {
  ...
}
```

`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");
```

Relaxing the Definitions

- **`run`** takes two **`String`** parameters, so it doesn't quite fit our definition of a recognition method
- We could make it fit by redefining it using a single delimited input: **`run(p+'#'+in)`** instead of **`run(p,in)`**
- That's the kind of trick we used to get multiple inputs into a Turing machine: recall *linearAdd*(101#1)
- Instead, we'll relax our definitions, allowing recognition and decision methods to take more than one **`String`** parameter
- So **`run`** is a recognition (but not a decision) 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

- $L($**`run`**$) =$
  $\{($**`p`**,**`in`**$) \mid$ **`p`** is a recognition method and **`in`** $\in L($**`p`**$)\}$
- A corresponding language for universal TMs:
  $\{m\#x \mid m$ encodes a TM and x is a string it accepts$\}$
- In either case, we'll call the language L_u
- (Remember *u* for *universal*)
- 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
- Tricky to prove such things; it is not enough to say we tried really hard but couldn't do 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?

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

- All possible results are contradictory:
  - If it accepts itself, that means **`shortcut`** determined it was not self-accepting
  - If it rejects itself, that means **`shortcut`** determined it was self-accepting
  - But it must return something, because **`shortcut`** is a decision method

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*
- No total TM can be a universal TM
- Verifies our earlier claim that total TMs are weaker than general TMs

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 Power of Self-Reference

- *This sentence is false*
- Easy to do in English
  - A sentence can refer to itself as “this sentence”
- Fairly easy to do with computational procedures:
  - A method can receive its source as a parameter
  - A TM can get a string encoding of itself
- Not a big stretch for modern programmers
- Self-reference is the key trick in our proof that L_u is not recursive

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(**`haltsRE`**) =
  {(**`p`**,**`in`**) | **`p`** is a recognition method that halts on **`in`**}
- A corresponding language for universal TMs:
  {$m\#x$ | m encodes a TM that halts on x}
- In either case, we'll call the language L_h
- (Remember *h* for *halting*)
- 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

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;              " +
  "}                                "
)
```

- All possible results are contradictory:
  - If it runs forever, that means **`halts`** determined it would halt
  - If it halts, that means **`halts`** determined it would run forever

Proof Summary

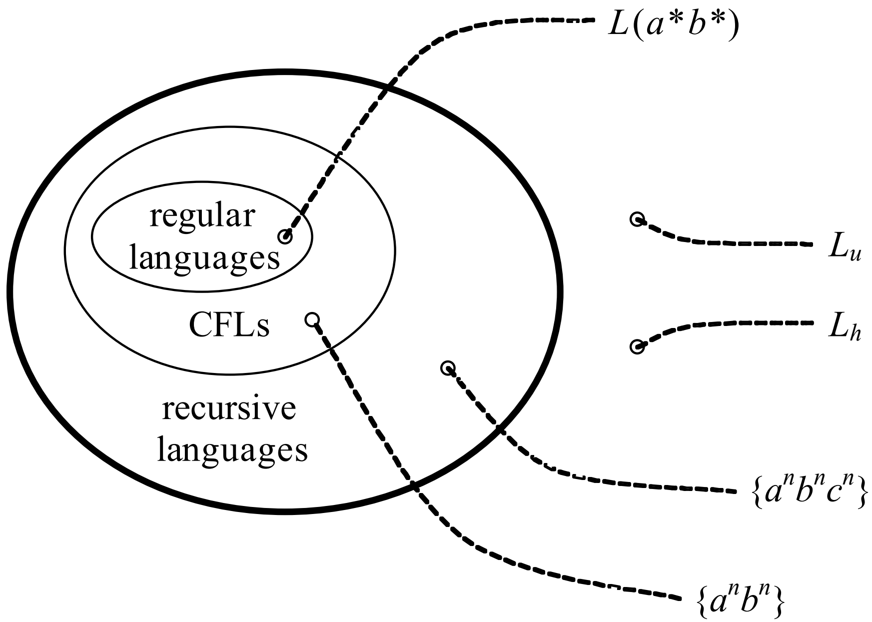
- We assumed by way of contradiction that `halts` could be implemented
- Using it, we showed an implementation of `narcissist`
- We showed that applying `narcissist` to itself results in a contradiction
- By contradiction, no program satisfying the specifications of `halts` exists
- In other words…

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

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

Planning A Trip

- You formulate a plan:
  1. I will drive my car to the airport
  2. I will fly to my friend's airport
  3. My friend will pick me up
- Steps 1 and 3 are clearly possible, so that just leaves step 2
- You have reduced an original problem *A* (making a trip from house to house) to another problem *B* (finding a flight from airport to airport)
- If you can get a flight, you can make the trip

What The Reduction Shows

- Reducing *A* to *B* shows that *A* is no harder than *B*
- It does not rule out the possibility that *A* is easier than *B*: there might be other ways to solve it
- For example, if you and your friend are in the same city, your plan will work, but is not optimal

Algorithmic Reductions

- Given problem *A*, a *reduction* is a solution of this form:
  1. Convert the instance of problem *A* into an instance of problem *B*
  2. Solve that instance of problem *B*
  3. Convert the solution of the instance of problem *B* back into a solution of the original instance of problem *A*
- If steps 1 and 3 are no harder than step 2, we can conclude that problem *A* is no harder than problem *B*

1. (Still, *A* might be easier than *B*; there might be an easier, completely different algorithm)

Reductions Proving a Language Is Recursive

- Given a language L_1, we can use a reduction to prove it is recursive:
  1. Given a string x_1 to be tested for membership in L_1, convert it into another string x_2 to be tested for membership in L_2
  2. Decide whether $x_2 \in L_2$
  3. Convert that decision about x_2 back into a decision about x_1
- If steps 1 and 3 are computable—if those conversions can be computed effectively, without infinite looping—and if L_2 is already known to be recursive, this proves that L_1 is recursive too

Example

```
boolean decideL1(String x1) {
  String x2="";
  for (int i = 0; i < x1.length(); i++) {
    char ith = x1.charAt(i);
    if (ith=='d') x2+='c';
    else x2+=ith;
  }
  boolean b = anbncn2(x2);
  return !b;
}
```

Step 1

Step 2

Step 3

$L_1 = \{x \in \{a,b,d\}^* \mid x \notin \{a^n b^n d^n\}\}$ by reduction to $L_2 = \{a^n b^n c^n\}$

Example

```
boolean anbn(String x1) {
  String x2=x1;
  for (int i = 0; i < x1.length()/2; i++)      Step 1
    x2+='c';
  boolean b = anbncn2(x2);                     Step 2
  return b;                                    Step 3
}
```

$L_1 = \{a^n b^n\}$ by reduction to $L_2 = \{a^n b^n c^n\}$

(Obviously, there's a more efficient way!)

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 Other Direction

- A reduction from *A* to *B* shows that *A* is no harder than *B*
- Equivalently: *B is no easier than A*
- Useful to show a language L_1 is not recursive
- Reduce *from* a nonrecursive language L_2 *to* the language L_1
- Then you can conclude L_1 is not recursive either, since it is no easier than L_2

Example: L_e

- L_e = {**`p`** | **`p`** is a recognition method that never returns true}
- In other words, L_e is the set of recognition methods **`p`** for which L(**`p`**) = {}
- (Remember *e* for empty)
- We will show that L_e is not recursive
- Proof is by reduction from L_h (a language we already know is nonrecursive) to L_e

Theorem 18.5.1

$\boxed{L_e \text{ is not recursive.}}$

- Proof is by reduction from the halting problem
- Assume by way of contradiction that L_e is recursive
- Then there is a decision method **`empty`** for it
- We can write a decision method **`halts`**…

```
boolean halts(String p, String x) {
  String x2 =
    "boolean f(String z) {          " +
    "  run(\""+p+"\",\""+x+"\");    " +
    "  return true;                 " +
    "}                              ";
  boolean b = empty(x2);
  return !b;
}
```

- **`x2`** is the source for a recognition method **`f`**
- **`f`** ignores parameter **`z`**, runs **`p`** on **`x`**, then returns true
- If **`p`** runs forever on **`x`**, $L($**`f`**$) = \{\}$; if not, $L($**`f`**$) = \Sigma^*$
- Thus, **`x2`** $\in L_e$ if and only if **`p`** runs forever on **`x`**
- So if **`empty`** is a decision method for L_e, **`halts`** is a decision method for L_h
- That's a contradiction: L_h is not recursive

Theorem 18.5.1, Summary

L_e is not recursive.

- Proof is by reduction from the halting problem
- Assume by way of contradiction that L_e is recursive
- Then there is a decision method **`empty`** for it
- We can write a method **`halts`**, as on the previous slide, that is a decision method for L_h
- That's a contradiction: L_h is not recursive
- By contradiction, L_e is not recursive

Example: L_r

- L_r = {**`p`** | **`p`** is a recognition method and L(**`p`**) is regular}
- For example, this string is in L_r, because Σ^* is regular:

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

- But our previous decision method **`anbn`** is not in L_r, because $\{a^n b^n\}$ is not regular
- (Remember *r* for regular)
- We will show that L_r is not recursive
- Proof is by reduction from L_h (a language we already know is nonrecursive) to L_r

Theorem 18.5.2

$\boxed{L_r \text{ is not recursive.}}$

- Proof is by reduction from the halting problem
- Assume by way of contradiction that L_r is recursive
- Then there is a decision method **`regular`** for it
- We can write a decision method **`halts`**…

```
boolean halts(String p, String x) {
  String x2 =
    "boolean f(String z) {          " +
    "  run(\""+p+"\",\""+x+"\");    " +
    "  return anbn(z);              " +
    "}                              ";
  boolean b = regular(x2);
  return !b;
}
```

- **`x2`** is the source for a recognition method **`f`**
- **`f`** runs **`p`** on **`x`**, returns true if and only if **`z`** $\in \{a^n b^n\}$
- If **`p`** runs forever on **`x`**, $L($**`f`**$) = \{\}$; if not, $L($**`f`**$) = \{a^n b^n\}$
- Thus, **`x2`** $\in L_r$ if and only if **`p`** runs forever on **`x`**
- So if **`regular`** is a decision method for L_r, **`halts`** is a decision method for L_h

Theorem 18.5.2, Summary

$\boxed{L_r \text{ is not recursive.}}$

- Proof is by reduction from the halting problem
- Assume by way of contradiction that L_r is recursive
- Then there is a decision method **`recursive`** for it
- We can write a method **`halts`**, as on the previous slide, that is a decision method for L_h
- That's a contradiction: L_h is not recursive
- By contradiction, L_r is not recursive

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

Theorem 18.6: Rice's Theorem

For all nontrivial properties α, the language

$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ has property $\alpha\}$

is not recursive.

- To put it another way: *all nontrivial properties of the RE languages are undecidable*
- Some examples of languages covered by the Rice's Theorem…

Rice's Theorem Examples

L_e = $\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ is empty$\}$
L_r = $\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ is regular$\}$
$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ is context free$\}$
$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ is recursive$\}$
$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $|L(\mathbf{p})| = 1\}$
$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $|L(\mathbf{p})| \geq 100\}$
$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $hello \in L(\mathbf{p})\}$
$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p}) = \Sigma^*\}$

What "Nontrivial" Means

- A property is *trivial* if no RE languages have it, or if all RE languages have it
- Rice's theorem does not apply to trivial properties such as these:

$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ is RE$\}$
$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p}) \supset \Sigma^*\}$

Proving Rice's Theorem

For all nontrivial properties α, the language
$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ has property $\alpha\}$
is not recursive.

- Proof is by reduction from the halting problem
- Given any nontrivial property α of the RE languages, define $A = \{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ has property $\alpha\}$
- Assume by way of contradiction that A is recursive
- Then there is a decision method **`falpha`** for it
- We can use it to write a decision method **`halts`**
- Two cases to consider: either {} has property α or it doesn't

```
boolean halts(String p, String x) {
  String x2 =
    "boolean f(String z) {          " +
    "  run(\""+p+"\",\""+x+"\");    " +
    "  return fy(z);                " +
    "}                              ";
  boolean b = falpha(x2);
  return !b;
}
```

- Case 1: {} has property α
- Because α is nontrivial, some RE language *Y* does not have it
- **`x2`** is the source for a recognition method **`f`**
- **`f`** runs **`p`** on **`x`**, then returns true if and only if **`z`** $\in Y$
- If **`p`** runs forever on **`x`**, L(**`f`**) = {}; if not, L(**`f`**) = Y
- Thus, **`x2`** $\in A$ if and only if **`p`** runs forever on **`x`**
- So if **`falpha`** is a decision method for A, **`halts`** is a decision method for L_h

```
boolean halts(String p, String x) {
  String x2 =
    "boolean f(String z) {            " +
    "  run(\""+p+"\",\""+x+"\");    " +
    "  return fy(z);                  " +
    "}                                ";
  boolean b = falpha(x2);
  return b;
}
```

- Case 2: {} does not have property α
- Because α is nontrivial, some RE language Y does have it
- **x2** is the source for a recognition method **f**
- **f** runs **p** on **x**, then returns true if and only if **z** $\in Y$
- If **p** runs forever on **x**, $L($**f**$) = \{\}$; if not, $L($**f**$) = Y$
- Thus, **x2** $\in A$ if and only if **p** halts on **x**
- So if **falpha** is a decision method for A, **halts** is a decision method for L_h

Proving Rice's Theorem

For all nontrivial properties α, the language
$\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ has property $\alpha\}$
is not recursive.

- Proof is by reduction from the halting problem
- Given any nontrivial property α of the RE languages, define $A = \{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ has property $\alpha\}$
- Assume by way of contradiction that A is recursive
- Then there is a decision method **`falpha`** for it
- Two cases to consider: either {} has property α or it doesn't
- Either way, we can write a method **`halts`**, as on the previous slides, that is a decision method for L_h
- That's a contradiction: L_h is not recursive
- By contradiction, A is not recursive

Using Rice's Theorem

- Easy to use, when it applies
- Example:

  $\{\mathtt{p} \mid \mathtt{p}$ is a recognition method and $|L(\mathtt{p})| = 1\}$
- To prove this is not recursive:
  - The language is of the form covered by Rice's theorem
  - The property in question, $|L(\mathtt{p})| = 1$, is nontrivial: some RE languages have one element and others don't

Guidance: Nonrecursive

- Sets of programs (or TMs, etc.) defined in terms of their runtime behavior are usually not recursive
- Of course, when Rice's theorem applies, such a language is definitely not recursive
- And such languages are usually not recursive, even if we can't use use Rice's theorem:
  - {**`p`** | **`p`** is a method that prints **`"hello world"`**}
  - {**`p`** | **`p`** is a method that never gets an uncaught exception}
  - {**`p`** | **`p`** is a method that produces no output}

Guidance: Recursive

- Sets of programs (or TMs, etc.) defined in terms of their *syntax* are usually recursive:
  - {**`p`** | **`p`** contains the statement **`while(true){}`**}
  - {*m* | *m* encodes a TM *M* with 10 states}

Caution

- This is just guidance: it is possible to construct exceptions either way
- For example: {(**`p`**,**`x`**) | **`p`** is a method that executes at least 10 statements when run with input **`x`**}
- Just start simulating **`p`** on **`x`** and count the number of statements executed:
  - If **`p`** returns before you get to 10, say no
  - If **`p`** gets to 10, say yes
- Either way, we get an answer; no infinite loops
- Although defined in terms of runtime behavior, this language is recursive

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

TMs That Enumerate

- We have treated TMs as recognition machines
- Alan Turing's original concept (1936) treated them as *enumerators*: they take no input, but simply generate a sequence of strings on an output tape
- Another way of defining languages:
  - $L(M) = \{x \mid$ for some i, x is the ith string in M's output$\}$
- Like all TMs, enumerators may run forever
- They must, if the language they enumerate is infinite
- They may, even if the language is finite

Enumerator Objects

- An *enumerator class* is a class with an instance method **`next`** that takes no input and returns a string (or runs forever)
- An enumerator object may preserve state across calls of **`next`**
- So **`next`** may (and generally does) return a different string every time it is called
- For an enumerator class **`C`**, *L*(**`C`**) is the set of strings returned by an infinite sequence of calls to the **`next`** method of an object of class **`C`**

L(**AStar**) = $\{a\}^*$

```
class AStar {
  int n = 0;

  String next() {
    String s = "";
    for (int i = 0; i < n; i++) s += 'a';
    n++;
    return s;
  }
}
```

- This enumerates in order of length
- Enumerators don't have to do that

L(**TwinPrimes**)

```
class TwinPrimes {
  int i = 1;

  String next() {
    while (true) {
      i++;
      if (isPrime(i) && isPrime(i+2))
        return i + "," + (i+2);
    }
  }
}
```

- Enumerates twin primes: `"3,5"`, `"5,7"`, `"11,13"`, …
- It is not known whether *L*(**`TwinPrimes`**) is infinite
- If not, there is a largest pair, and a call to **`next`** after that largest pair has been returned will run forever

An Enumerator Problem

- Make an enumerator class for the set of all pairs of natural numbers, $\{(j,k) \mid j \geq 0, k \geq 0\}$
- (As always, we'll use decimal strings)
- This is a bit trickier…

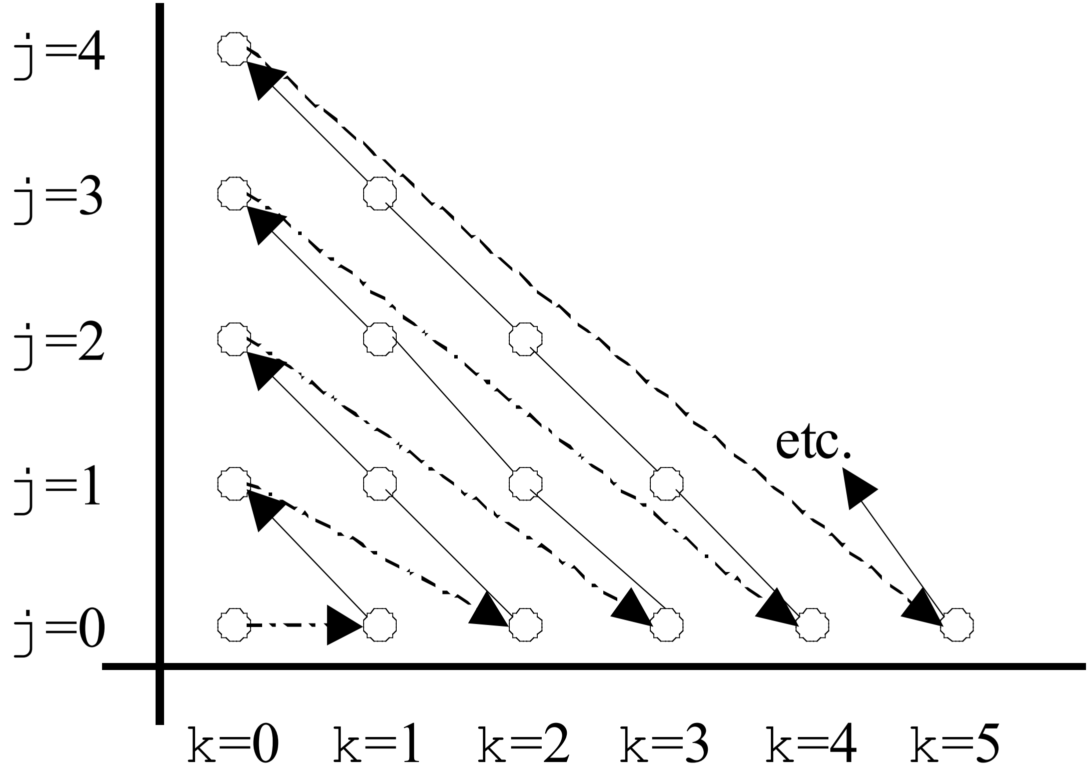
`NatPairs` Failures

```
class BadNatPairs1 {
  int k = 0;

  String next() {
    return "(0," + k++ + ")";
  }
}
```

$\{(j,k) \mid j = 0, k \geq 0\}$

```
class BadNatPairs2 {
  int j = 0;
  int k = 0;

  String next() {
    return "(" + j++ + "," + k++ + ")";
  }
}
```

$\{(j,k) \mid j = k, k \geq 0\}$




```
class NatPairs {
  int n = 0;
  int j = 0;

  String next() {
    String s = "(" + j + "," + (n-j) + ")";
    if (j<n) j++;
    else {j=0; n++;}
    return s;
  }
}
```

An Easier Enumerator Problem

- Make a class **`SigmaStar`** that enumerates Σ^*
- For example, if $\Sigma = \{a,b\}$, a **`SigmaStar`** object might produce `"", "a", "b", "aa", "ab",” ba", "bb", "aaa",`…
- Exact order does not matter here
- Not difficult … left as an exercise

Numbering A Language

- We can number the strings in a language by the order in which they are enumerated
- For example, the *i*th string from **`SigmaStar`**:

```
String sigmaStarIth(int i) {
  SigmaStar e = new SigmaStar();
  String s = "";
  for (int j = 0; j<=i; j++) s = e.next();
  return s;
}
```

- Not necessarily one-to-one, but for every $s \in \Sigma^*$ there is at least one i such that **`sigmaStarIth`**(i) = s

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

Theorem 18.8

A language is RE if and only if it is $L(M)$ for some enumeration machine M.

- Our definition of RE used our interpretation of TMs as recognition machines
- So the theorem says there is a recognition machine for L if and only if there is an enumeration machine for L
- To show it, we will give two constructions:
  - Given an enumerator class, construct a recognition method
  - Given a recognition method, construct an enumerator class

Enumerator To Recognizer

```
boolean aRecognize(String s) {
  AEnumerate e = new AEnumerate();
  while (true)
    if (s.equals(e.next())) return true;
}
```

- A recognition (not decision) method
- **`aRecognize(s)`** returns true if and only if **`AEnumerate`** eventually produces **`s`**
- So *L*(**`aRecognize`**) = *L*(**`AEnumerate`**)

A More Difficult Direction

```
class BadAEnumerate {
  SigmaStar e = new SigmaStar();

  String next() {
    while (true) {
      String s = e.next();
      if (aRecognize(s)) return s;
    }
  }
}
```

- Only works if **`aRecognize`** is a decision method
- If **`aRecognize`** runs forever on one of the strings generated by **`SigmaStar`**, **`next`** will get stuck
- We need a trick…

`runLimited`

- A time-limited version of `run`
- Recall that `run(p,in)` runs recognition method `p` on input `in` and returns the result
- `runWithTimeLimit(p,in,j)` returns true if and only if `p` returns true for `in` within `j` steps of the simulation
- This can be total, because it can return false as soon as the `j`th step has passed

Recognizer To Enumerator

```
class AEnumerate {
  NatPairs e = new NatPairs();

  String next() {
    while (true) {
      int (j,k) = e.next();
      String s = sigmaStarIth(j);
      if (runWithTimeLimit(aRecognize,s,k)) return s;
    }
  }
}
```

- $s \in L$(**`aRecognize`**) if and only if s is the jth string in Σ^* and is accepted within k steps, for some pair (j,k)
- So L(**`aRecognize`**) = L(**`AEnumerate`**)

Theorem 18.8, Summary

A language is RE if and only if it is $L(M)$ for some enumeration machine M.

- Our definition of RE used our interpretation of TMs as recognition machines
- So the theorem says there is a recognition machine for L if and only if there is an enumeration machine for L
- We showed it using two constructions:
  - Given an enumerator class, construct a recognition method
  - Given a recognition method, construct an enumerator class
- The name “recursively enumerable” makes more sense in this light!

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

Languages That Are Not RE

- We've seen examples of nonrecursive languages like L_h and L_u
- Although not recursive, they are still RE: they can be defined using recognition methods (but not using decision methods)
- Are there languages that are not even RE?
- Yes, and they are easy to find…

Theorem 18.9

If a language is RE but not recursive, its complement is not RE.

- Proof is by contradiction
- Let *L* be any language that is RE but not recursive
- Assume by way of contradiction that the complement of *L* is also RE
- Then both *L* and its complement have recognition methods; call them **`lrec`** and **`lbar`**
- We can use them to implement a *decision* method for *L*…

Theorem 18.9, Continued

If a language is RE but not recursive,
its complement is not RE.

```
boolean ldec(String s) {
  for (int j = 1; ; j++) {
    if (runLimited(lrec,s,j)) return true;
    if (runLimited(lbar,s,j)) return false;
  }
}
```

- For some `j`, one of the two `runLimited` calls must return true
- So this is a decision method for *L*
- This is a contradiction; *L* is not recursive
- By contradiction, the complement of *L* is not RE

Closure Properties

- So the RE languages are not closed for complement
- But the recursive languages are
- Given a decision method **`ldec`** for *L*, we can construct a decision method for *L*'s complement:

```
boolean lbar(String s) {return !ldec(s);}
```

- That approach does not work for nonrecursive RE languages
- If the recognition method **`lrec(s)`** runs forever, **`!lrec(s)`** will too

Examples

- L_h and L_u are RE but not recursive
- By Theorem 18.9, their complements are not RE:

$\overline{L_u}$ = {(**`p`**,**`s`**) | **`p`** is *not* a recognition method that returns true for **`s`**}

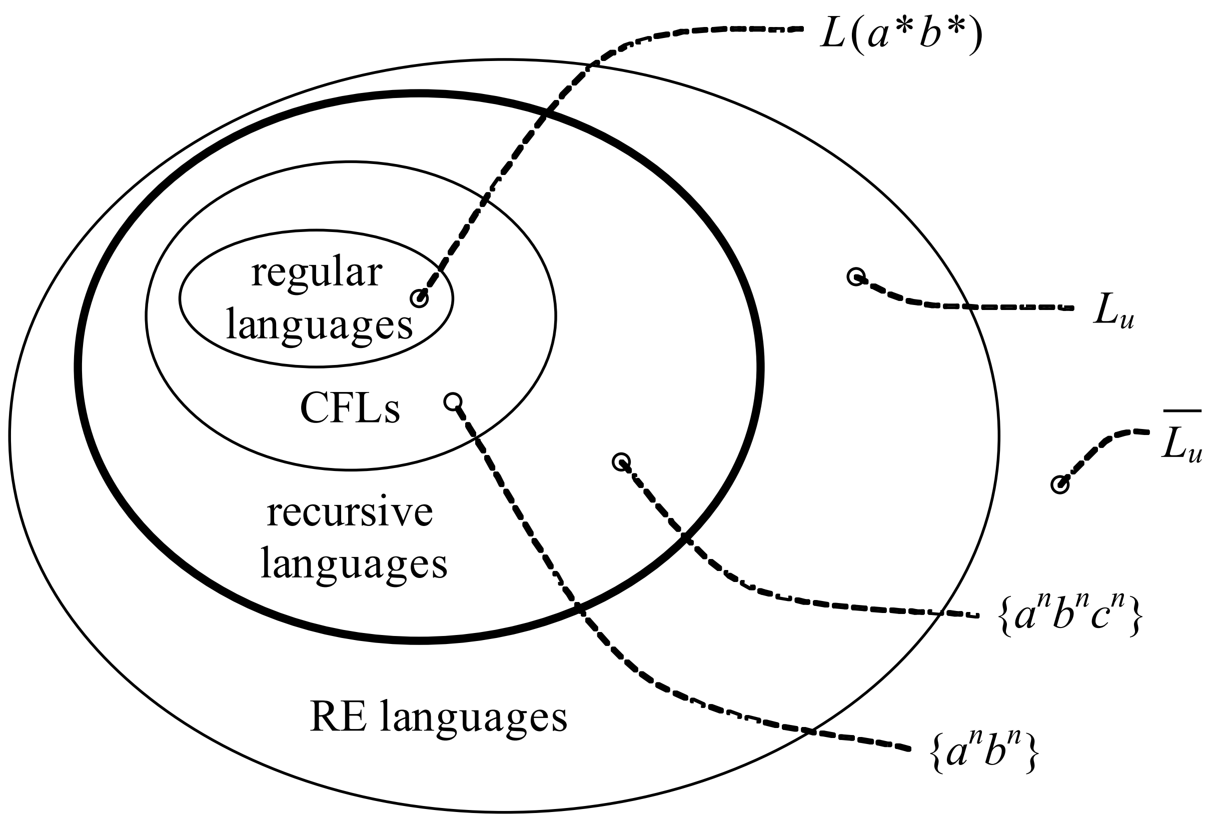
$\overline{L_h}$ = {(**`p`**,**`s`**) | **`p`** is *not* a recognition method that halts given **`s`**}

- These languages cannot be defined as $L(M)$ for any TM M, or with any Turing-equivalent formalism

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 Big Picture



Recursive

- When a language is *recursive*, there is an effective computational procedure that can definitely categorize all strings
  - Given a positive example it will decide yes
  - Given a negative example it will decide no
- A language that is *recursive*, a property that is *decidable*, a function that is *computable*
- All these terms refer to total-TM-style computations, computations that always halt

RE But Not Recursive

- There is a computational procedure that can effectively categorize positive examples:
  - Given a positive example it will decide yes
  - Given a negative example it may decide no, or may run forever
- A property like this is called *semi-decidable*
- Like the property of $(\mathbf{p},\mathbf{s}) \in L_h$
  - If **p** halts on **s**, a simulation can answer yes
  - If not, neither simulation nor any other approach can always answer with a definite no

Not RE

- There is no computational procedure for categorizing strings that gives a definite yes answer on all positive examples
- Consider $(\mathbf{p},\mathbf{s}) \in \overline{L_h}$
- One kind of positive example would be a recognition method **`p`** that runs forever on **`s`**
- But there is no algorithm to identify such pairs
- Obviously, you can't simulate **`p`** on **`s`**, see if it runs forever, and then say yes

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

General Grammars

- We defined grammars using general productions of the form $x \rightarrow y$:
  - x and y can be any strings, $x \neq y$
- But our examples have all been context free:
  - Right-hand side x is a single nonterminal symbol
- You can define more languages if you use productions that are not context free

Example: $a^n b^n c^n$

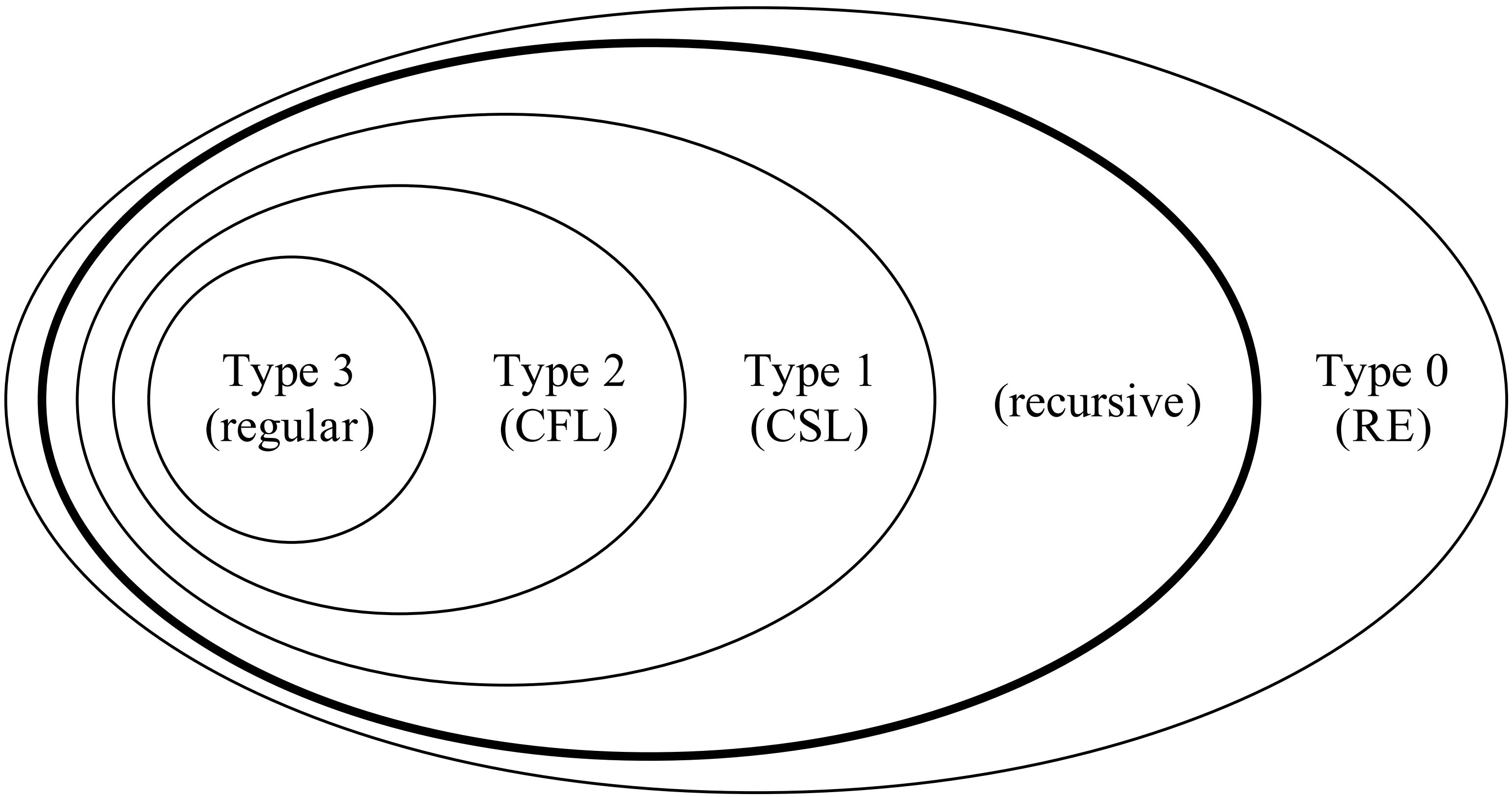
$$
\boxed{\begin{array}{l} S \rightarrow aBSc \mid abc \mid \varepsilon \\ Ba \rightarrow aB \\ Bb \rightarrow bb \end{array}}
$$

- Here are some derivations for this grammar:
  - $S \Rightarrow \varepsilon$
  - $S \Rightarrow abc$
  - $S \Rightarrow aBSc \Rightarrow aBabcc \Rightarrow aaBbcc \Rightarrow aabbcc$
  - $S \Rightarrow aBSc \Rightarrow aBaBScc \Rightarrow aBaBabccc \Rightarrow aaBBabccc \Rightarrow aaBaBbccc$ $\Rightarrow aaaBBbccc \Rightarrow aaabbbccc$
- The language generated is $a^n b^n c^n$: recursive but not context-free

Chomsky Hierarchy

- Noam Chomsky, late 1950s
- Four classifications for grammars, determined by the syntax of productions:
  - Type 0 (unrestricted): all forms allowed
  - Type 1 (context sensitive): form $xAz \to xyz$, where $y \neq \varepsilon$; $S \to \varepsilon$ is also allowed, if S does not appear on the right-hand side of any production
  - Type 2 (context free)
  - Type 3 (right linear)

Remarkable Correspondence



The CSLs

- Context-sensitive languages
  - A superset of the CFLs, a subset of the regular languages
  - A *large* subset: there are languages that are recursive but not context-sensitive, but they're hard to find
- Another way to define them: nondeterministic linear-bounded automata (NLBA)
  - Start with the NDTM model
  - Add the restriction that writing on **B** is not permitted
  - In effect, this limits the NDTM to that part of the tape occupied by the input
  - *L* is accepted by some NLBA if and only if *L* is a CSL

Uncomputability And CFGs

- We saw Rice's theorem:

> For all nontrivial properties α, the language
> $\{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p})$ has property $\alpha\}$
> is not recursive.

- There's nothing as categorical for CFGs
- But there are a number of interesting properties α for which
  $\{G \mid G$ is a CFG and $L(G)$ has property $\alpha\}$
  is not recursive

Examples

- These languages are not recursive:
  - $\{G \mid G \text{ is a CFG and } L(G) = \Sigma^*\}$
  - $\{G \mid G \text{ is a CFG and } \overline{L(G)} \text{ is a CFL}\}$
- Similarly, these questions are undecidable:
  - Do two given CFGs generate the same language?
  - Is the intersection of the languages defined by two given CFGs a CFL?

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

L_e and L_f

- Two languages:
  - $L_e = \{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p}) = \{\}\}$
  - $L_f = \{\mathbf{p} \mid \mathbf{p}$ is a recognition method and $L(\mathbf{p}) = \Sigma^*\}$
- Neither is recursive (by Rice's theorem)
- In fact, neither is RE
- Yet there is a sense in which one is harder to recognize than the other…

Reduction From Halting

- We saw that L_h is not recursive:
  - {(**`p`**,**`in`**) | **`p`** is a recognition method that halts on **`in`**}
- We showed that L_e is not recursive by reduction from L_h:
  - If there were a way to decide L_e, we could use that to decide L_h
  - Conclusion: L_e must not be recursive
- So no decision method for L_e is possible
- But if we did have some other way of deciding L_e, we could use that to decide L_h as well

Oracle Machines

- TMs with such impossible powers are called oracle machines
- Just like ordinary TMs, but augmented with an oracle: a one-step way of checking membership in a particular language
- Giving a TM an oracle for a nonrecursive language like L_e increases its power
- Given an oracle for L_e, both L_e and L_h are recursive
- With a different construction, you can show that given an oracle for L_h, both L_e and L_h are recursive

Levels Of Impossibility

- An oracle for L_h doesn't end uncomputability
- It can decide the halting problem, for ordinary TMs, but not for TMs with L_h oracles
- That requires a more powerful oracle, whose addition make the halting problem harder, requiring a still stronger oracle, and so on…
- An infinite hierarchy of oracles

L_e and L_f Revisited

- Two languages:
  - L_e = {**`p`** | **`p`** is a recognition method and L(**`p`**) = {}}
  - L_f = {**`p`** | **`p`** is a recognition method and L(**`p`**) = Σ^*}
- Neither is recursive (by Rice's theorem)
- In fact, neither is RE
- L_f is harder to recognize than L_e in this sense:
  - An oracle for L_h makes L_e recursive
  - An oracle for L_h does not make L_f recursive; that requires one of the more powerful oracles

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

Uncomputability In Other Domains

- All our nonrecursive languages have been languages of programs
- Of course, they're interesting to programmers
- Uncomputability turns up in many other domains
- Especially at the foundations of mathematics…

Formalist View Of Mathematics

- One view: math is a structure of theorems
  - Each built from simpler theorems by mechanically following rules of logic
  - At the bottom are axioms, are accepted as true because they are simple and self-evident
- If you think of mathematics that way, then:
  - It is important for the axioms to be *consistent,* meaning that they lead to no false theorems
  - And it is important for them to be *complete*, meaning that all true theorems can be proved

David Hilbert, 1862-1943

- One of the most influential mathematicians in modern history
- Issued a list of 23 open problems at a conference in Paris in 1900
- They guided mathematical research for the century, as he intended
- A solution to any problem on the list has brought fame to the mathematician who solved it
- Most are now “solved”, in a sense

Formalist Goals

- Goals:
  - Prove the foundational axioms are consistent (#2 on the list)
  - Show that they are complete
  - Give an exact procedure to decide the truth of any given assertion
- Hilbert believed that finite proof or disproof was always possible for well-formed mathematical conjectures
- He (and most other mathematicians) believed that these goals were almost within reach

Kurt Gödel, 1906-1978

- Showed how to express "this assertion has no proof" in number theory: a formal mathematical language of simple assertions about natural numbers
  - Such self-reference is easy to do with English, and not hard with computer programs, but very hard in number theory
  - If false, it has a proof: that's a proof of something false, so the axioms are not consistent
  - If true, it has no proof: that's a truth that can't be proved, so the axioms are not complete
- His first incompleteness theorem: no axiomatic system containing number theory can be both consistent and complete

Formalist Goals, Revisited

- Gödel (1929-1931)
  - No axiomatic system containing number theory can be both consistent and complete
  - No consistent system containing number theory can prove its own consistency
- Turing, Church (1936)
  - There can be no algorithm for deciding provability

More Undecidabilities

- Since then, many other mathematical problems have been found to be uncomputable
- Example: solving Diophantine equations
  - Polynomial equations, such as $x^2 + y^2 = z^2$, restricted to integer variables and constants
  - Find a general algorithm for these: Hilbert's tenth problem
  - Matiyasevich "solved" this one in 1970, showing that it has no solution
  - For every TM M there is a Diophantine equation with one variable x which has a solution exactly where $x \in L(M)$
- As always, close ties between computer science and the foundations of mathematics