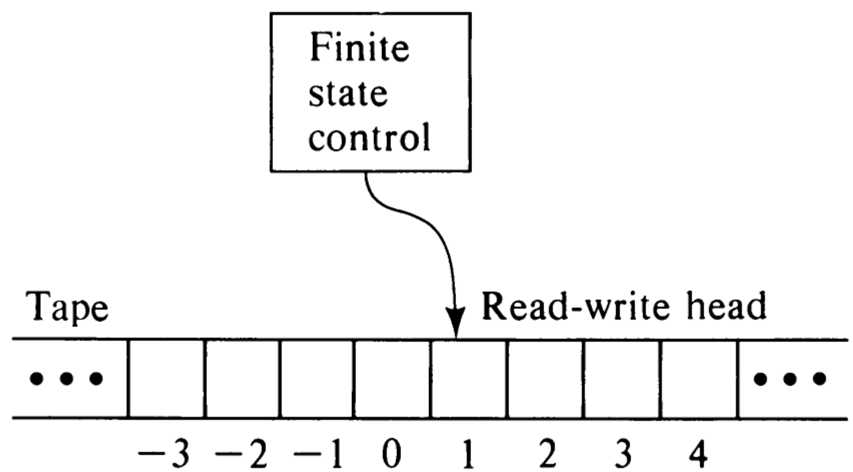
Undecidable Problems

Recall

- Turing Machines
- Maximal Power
- General-Purpose Machines
- Encoding



Everything Is An Integer

- Data types have become very important as a programming tool
- At another level, there is only one type, which you may think of as integers or strings
- Key point: strings that are programs are just another way to think about the same one data type

Example: Text

- Strings of ASCII or Unicode characters can be thought of as binary strings, with 8 or 16 bits/character
- Binary strings can be thought of as integers
- It makes sense to talk about “the *i*-th string”

Example: Images

- Represent an image in (say) GIF
- The GIF file is an ASCII string
- Convert string to binary
- Convert binary string to integer
- Now we have a notion of “the i-th image”

Example: Proofs

- A formal proof is a sequence of logical expressions, each of which follows from the ones before it
- Encode mathematical expressions of any kind in Unicode
- Convert expression to a binary string and then an integer
- It is a sequence of expressions; we just need a way to *separate* them

Example: Programs

- Programs are just another kind of data
- Represent a program in ASCII
- Convert to a binary string, then to an integer

Abstract Machines

- Finite Automata
  - Cannot solve problems that involve unrestricted counting
- Pushdown Automata
  - Cannot handle any problem where information needs to be reused in more than one place

Turing Machines

- The most advanced device we have seen
- Seems to have everything that we need:
  - Unlimited storage that can be accessed in any order
  - Arbitrary loops
  - Conditionals
  - Subroutines

How Much Further Can We Push?

- To get more increasingly powerful systems
- Perhaps not indefinitely: our attempts to make Turing machines more powerful by adding features did not get us anywhere
- There may be a hard limit on computational power
  - What are computers and programming languages fundamentally capable of doing?
  - Is there anything that they can’t do?
  - Are there any impossible programs?

Universal Systems Can Perform Algorithms

- The practical purpose of a computing machine is to perform *algorithms*
- An algorithm is a list of instructions describing some process for turning an input value into an output value
- Those instructions need to fulfill:
  - Finiteness, simplicity, termination, and correctness

An Important Question

- Can *any* algorithm be turned into instructions suitable for execution by a machine?
- Abstract intuitive ideas and concrete and logical implementation of an algorithm in a computational system are different
- Could there be *an* algorithm so large, complex, and unusual that its essence cannot be captured by an unthinking mechanical process? -- This is a philosophical rather than scientific question

Church-Turing Thesis

- The idea any algorithm can be performed by a machine–specifically a deterministic Turing machine—is called *Church-Turing Thesis*
- It is a conjecture *not* a proven fact
  - It certainly looks so
  - Historical hints
- It has enough evidence in its favor to be generally accepted as true

Implications

- Turing machines, despite their simplicity, have all the power required to perform *any computation* that can in principle be carried out by a person following simple instructions
- Further, it is just not possible to do any better: any real-world computer or programming language can only ever do as much as a Turing machine can do, and no more

Universality

- Universality is a powerful idea
- General-purpose computers are universal
  - A Turing machine that is capable of simulating any other Turing machine
- Inconvenient consequence:
  - Any system that is powerful enough to be universal will *inevitably* allow us to construct computations that loop forever without halting

Decidability

- Turing machines have a lot of power and flexibility
  - Execute arbitrary programs encoded as data
  - Perform any algorithms we can think of
  - Run for an unlimited amount of time
  - Calculate their own descriptions
- These machines have turned out to be representative of universal systems in general

Decidability

- Is there anything that Turing machines—and therefore real-world computers and programming languages—cannot do?
- Decision problems: a decision problem is any question with a yes or no answer, "is 2 less than 3?" "does regular expression (a(|b))* match the string abaab?"

Decidability

- A decision problem is *decidable* (or *computable*) if there is *an* algorithm that is guaranteed to solve it in a *finite* amount of time for *any* possible input
- The Church-Turing thesis claims that every algorithm *can* be performed by a Turing machine
- For a problem to be decidable, we have to be able to design a Turing machine that always produces the correct answer and always halts if we let it run for long enough

Decidability

- Is there *always* a clever way to sneak around a problem and find a way to implement a machine, or a program, that is guaranteed to solve it in a finite amount of time?
- No, unfortunately not
- There are many (actually *infinitely* many) decision problems that are undecidable

The Halting Problem

- It asks whether the execution of a particular Turing machine with a particular initial tape will ever halt
- In more practical terms: given a string containing the source code of a *program* and another string of *data* for that program to read from standard input, will running that *program* ultimately result in an answer or just an infinite loop?

Difficulty

- It is hard to predict what a program will do without actually running it
- A *halting-detection* algorithm must find a way to produce a definitive answer in a *finite* amount of time just by analyzing the *text* of the program, not by simply running it and waiting
- That is actually no good either: if the program does not halt, it will run forever and we will not get an answer, no matter how long we wait

Fundamentally Impossible

- Let us pretend halting problem is decidable
- It is possible to write a full implementation of halts(program, input), which always comes back with a true or false answer for every program and input, and the answer always correctly predicts whether program would halt if it was run with input on standard input

Fundamentally Impossible

- We then will be able to construct a new method halts_on_itself(program) that calls halts to determine what a program does when run with its own source code as input

```
def halts_on_itself(program)
        halts(program,program)
end
```

- halts_on_itself always finish and return a Boolean value: true if program halts when run with itself as input, false if it loops forever

Fundamentally Impossible

- Based on halts and halts_on_itself, we now can develop a program called do_the_opposite:

```
program = $stdin.read
if halts_on_itself (program)
   while true
            # do nothing
   end
end
```

What does halts_on_itself return if we call it with do_the_opposite's source code as its argument?

Fundamentally Impossible

- halts_on_itself must return either true or false when given the source of do_the_opposite as an argument
- If it returns true to indicate a halting program, then do_the_opposite will loop forever, which means halts_on_itself was wrong about what would happen
- If it returns false, that'll make do_the_opposite immediately halt, again contradicting halts_on_itself's prediction

Decidability

- Recall definition of decidability – a decision problem is decidable if there is an algorithm that is guaranteed to solve it in a finite amount of time for any possible input
- We have shown it is impossible to write a program that completely solves the halting problem
- Church-Turing thesis says that every algorithm can be performed by a Turing machine
- There is no Turing machine for solving halting problem, there is no algorithm either; in other words, the halting problem is undecidable

Other Undecidable Problem

- Can we just not build “do the opposite” program and eliminate this depressing situation and disregard it as an academic curiosity and go on with our lives?
- Unfortunately, it is not that simple. The structure of the as-shown undecidability proof points to something larger and more general
- Any nontrivial property (a claim about what a program does, not how it does it) of program behavior is undecidable – Rice’s theorem

Depressing Implications

- Undecidability is inconvenient and the halting problem is disappointing
- We want the unrestricted power of a universal programming language
- We want to write programs that produce a result without getting stuck in an infinite loop
- We can’t have everything

Depressing Implications

- Not only is the question “does this program halt?” undecidable, but so is “*does this program do what I want it to do?*”
- We really wanna this, why?
- It might be mechanically checkable for individual programs, not in general ☹
- We will never be able to *completely* trust machine to do the job for us

Depressing Implications

- It would save a lot of time and money if we could use an *automated* system to check every program for compliance, but thanks to undecidability, it’s not possible to build a system that does the job *accurately*
- We have no choice but to hire human beings to manually test those programs by running them, disassembling them, and (with the help of OS, for example) profiling their dynamic behaviors ← Is this a good news for programmers?

Fundamental Problems

- We do not have the power to look into the future and see what will happen when a program is executed
- The only general way to find out what a program does is to *run it for real*
- Some programs are simple enough to have behavior that is straightforwardly predictable, a universal language will always permit programs whose behavior cannot be predicted just by analyzing their source code

Fundamental Problems

- When we do decide to run a program, there is no reliable way to know how long it will take to finish
- The only general solution is to run it and wait, but since we know that programs in a universal language can loop forever without halting, there will always be some programs for which no finite amount of waiting is long enough

Why Does This Happen?

- Underlying problem: in general, program behavior is too powerful/complex to be accurately predicted
- Any system with enough power to be self-referential cannot correctly answer every question about itself

Why Does This Happen?

- In the case of universal programming languages, there is *no more powerful system* for us to upgrade to
- The Church-Turing thesis tells us that any usable algorithm we invent for making predictions about the behavior of programs *can itself be performed by a program*, so we are stuck with the capabilities of universal systems

What Can Be done?
Coping with Uncomputability

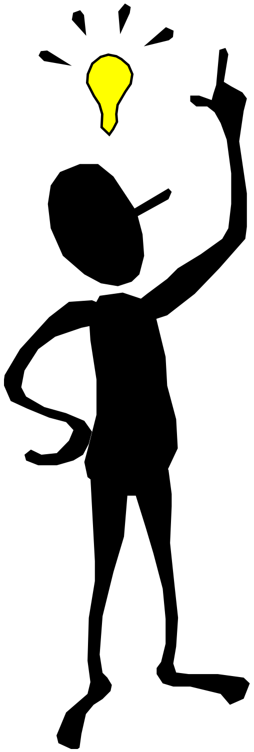
- The whole point of writing a program is to get a computer to do something useful
- Denial is tempting but might be overreaction
- Program analysis is not impossible, it is that we cannot write a nontrivial analyzer that will *always* halt and produce the right answer

What Can Be done? Coping with Uncomputability

- In spite of undecidability:
  - Ask undecidable questions, but give up if an answer can’t be found
  - Ask several small questions whose answers, when taken together, provide empirical evidence for the answer to a larger question
  - Ask decidable questions by being conservative where necessary
  - Approximate a program by converting it into something simpler, then ask decidable questions about the approximation

Next

- Computational Complexity



Thanks ! ☺

Questions ?