16.070

Introduction to Computers & Programming

Theory of computation: More on CFG, CNF, Turing machines, complexity

Prof. Kristina Lundqvist
Dept. of Aero/Astro, MIT

So far …

- Two different, though equivalent, methods of describing languages: **finite automata** and **regular expressions**.
  - Many, but not all languages can be described in that way, e.g., **not** the following: $\{0^n 1^n \mid n \geq 0\}$
- **Context free grammars**, a more powerful method of describing languages.
  - Can describe features that have a recursive structure.
  - An important application of CFGs occurs in the specification and compilation of programming languages.
  - **Push down automata**, a class of machines recognizing the context-free languages.

A context free grammar

- A 4-tuple (**V**, S, **R**, **S**)
  - **V** is a finite set called the ***variables***
  - S is a finite set, disjoint from **V**, called the ***terminals***
  - **R** is a finite set of ***rules***, with each rule being a variable and a string variables and terminals
  - **S** ∈ **V** is the ***start variable***
- If *u*, *v*, and *w* are strings of variables and terminals, and A → *w* is a rule of the grammar, we say that *uAv* ***yields*** *uwv*, written $uAv \Rightarrow uwv$.

  Write $u \Rightarrow v$ if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

  $$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$$

Example

- Grammar **G** = ({S}, {a, b}, R, S).
- The set of rules, R, is: S → aSb | SS | ε
- This grammar generates strings such as *abab*, *aaabbb*, and *aababb*.
- L(G) is the language of all strings of properly nested parentheses

(if you think of *a* as being a “(“ and *b* being a “)”

Chomsky Normal Form

- **Chomsky Normal Form** (CNF) is a simple and useful form of a CFG
- A CFG is in CNF if every rule is of the form

$$A \rightarrow BC$$
$$A \rightarrow a$$

- Where ***a*** is any terminal and *A*, *B*, *C* are any variables except *B* and *C* may not be the start variable
  - There are two and only two variables on the right hand side of the rule

Theorem

- Any context free language may be generated by a context free grammar in Chomsky Normal Form

- To show how this is possible we must be able to **convert any CFG into CNF**
  1. Eliminate all e rules of the form **A → e**
     1. Exception: *S* → *e* is permitted where *S* is the start variable
  2. Eliminate all unit rules of the form **A → B**
  3. Convert any remaining rules into the form **A → BC**

Proof

- First **add a new start symbols** S_0 and the rule $S_0 \rightarrow S$ where S was the original start symbol
  - This guarantees the new start symbol is not on the RHS of any rule

- **Remove all** e **rules**.
  - Remove a rule $A \rightarrow \varepsilon$ where A is not the start symbol. For each occurrence of A on the RHS of a rule, add a new rule with that occurrence of A deleted
  - Ex:

    $R \rightarrow uAv$ becomes $R \rightarrow uv$
  - This must be done for each occurrence of A, e.g.:

    $R \rightarrow uAvAw$ becomes $R \rightarrow uvAw \mid uAvw \mid uvw$

Repeat until all ε rules are removed, not including the start

Proof

- Next **remove all unit rules** of the form $A \rightarrow B$
  - Whenever a rule $B \rightarrow u$ appears, add the rule $A \rightarrow u$.
  - u may be a string of variables and terminals
  - Repeat until all unit rules are eliminated
- Convert all remaining rules into the form with **two variables on the right**
  - The rule $A \rightarrow u_1u_2u_3 \ldots u_k$ becomes
  - $A \rightarrow u_1A_1 \quad A_1 \rightarrow u_2A_2 \quad \ldots \quad A_{k-2} \rightarrow u_{k-1}u_k$
  - Where the A_i's are new variables. u may be a variable or a terminal (and in fact a terminal must be converted to a variable since CNF does not allow a mixture of variables and terminals on the right hand side)

Example

- Convert the following grammar into CNF

  $S \rightarrow ASA \mid aB$

  $A \rightarrow B \mid S$

  $B \rightarrow b \mid \varepsilon$

First add a new start symbol S_0:

$S_0 \rightarrow S$

$S \rightarrow ASA \mid aB$

$A \rightarrow B \mid S$

$B \rightarrow b \mid \varepsilon$

Example

- Next remove the epsilon transition from rule B

  $S_0 \rightarrow S$

  $S \rightarrow ASA \mid aB \mid$ **a**

  $A \rightarrow B \mid S \mid e$

  $B \rightarrow b \mid e$

- We must repeat this for rule A:

  $S_0 \rightarrow S$

  $S \rightarrow ASA \mid aB \mid a \mid$ **AS | SA | S**

  $A \rightarrow B \mid S \mid e$

  $B \rightarrow b$

Example

- Next remove unit rules, starting with $S_0 \rightarrow S$ and $S \rightarrow S$ can also be removed

  $S_0 \rightarrow$ **ASA | aB | a | AS | SA**

  $S \rightarrow$ **ASA | aB | a |AS | SA**

  $A \rightarrow B|S$

  $B \rightarrow b$

- Next remove the rule for $A \rightarrow B$

  $S_0 \rightarrow$ ASA | aB | a | AS | SA

  $S \rightarrow$ ASA | aB | a |AS | SA

  $A \rightarrow$ **b**|S

  $B \rightarrow b$

- Next remove the rule for $A \rightarrow S$

  $S_0 \rightarrow$ ASA | aB | a | AS | SA

  $S \rightarrow$ ASA | aB | a |AS | SA

  $A \rightarrow$ b| **ASA | aB | a |AS | SA**

  $B \rightarrow b$

Example

- Finally convert the remaining rules to the proper form by adding variables and rules when we have more than three things on the RHS

  $S_0 \rightarrow$ ASA | aB | a | AS | SA

  $S \rightarrow$ ASA | aB | a |AS | SA

  $A \rightarrow$ b| ASA | aB | a |AS | SA

  $B \rightarrow$ b

- Becomes

  $S_0 \rightarrow$ **AA_1** | **A_2B** | a | AS | SA

  $A_1 \rightarrow SA$

  $A_2 \rightarrow a$

  $S \rightarrow$ **AA_1** | **A_2B** | a |AS | SA

  $A \rightarrow$ b| **AA_1** | **A_2B** | a |AS | SA

  $B \rightarrow$ b

We are done!

CNF and Parse Trees

- Chomsky Normal Form is useful to interpret a grammar as a parse tree
  - CNF forms a **binary tree**!
  - Consider the string **babaaa** on the **previous grammar**

    $S_0 \rightarrow AS \rightarrow bS \rightarrow bAS \rightarrow bASS \rightarrow baSS \rightarrow baASS \rightarrow babSS \rightarrow babSAS \rightarrow babaAS \rightarrow babaaS \rightarrow$ **babaaa**

$S_0 \rightarrow AA_1 \mid A_2B \mid a \mid AS \mid SA$

$A_1 \rightarrow SA$

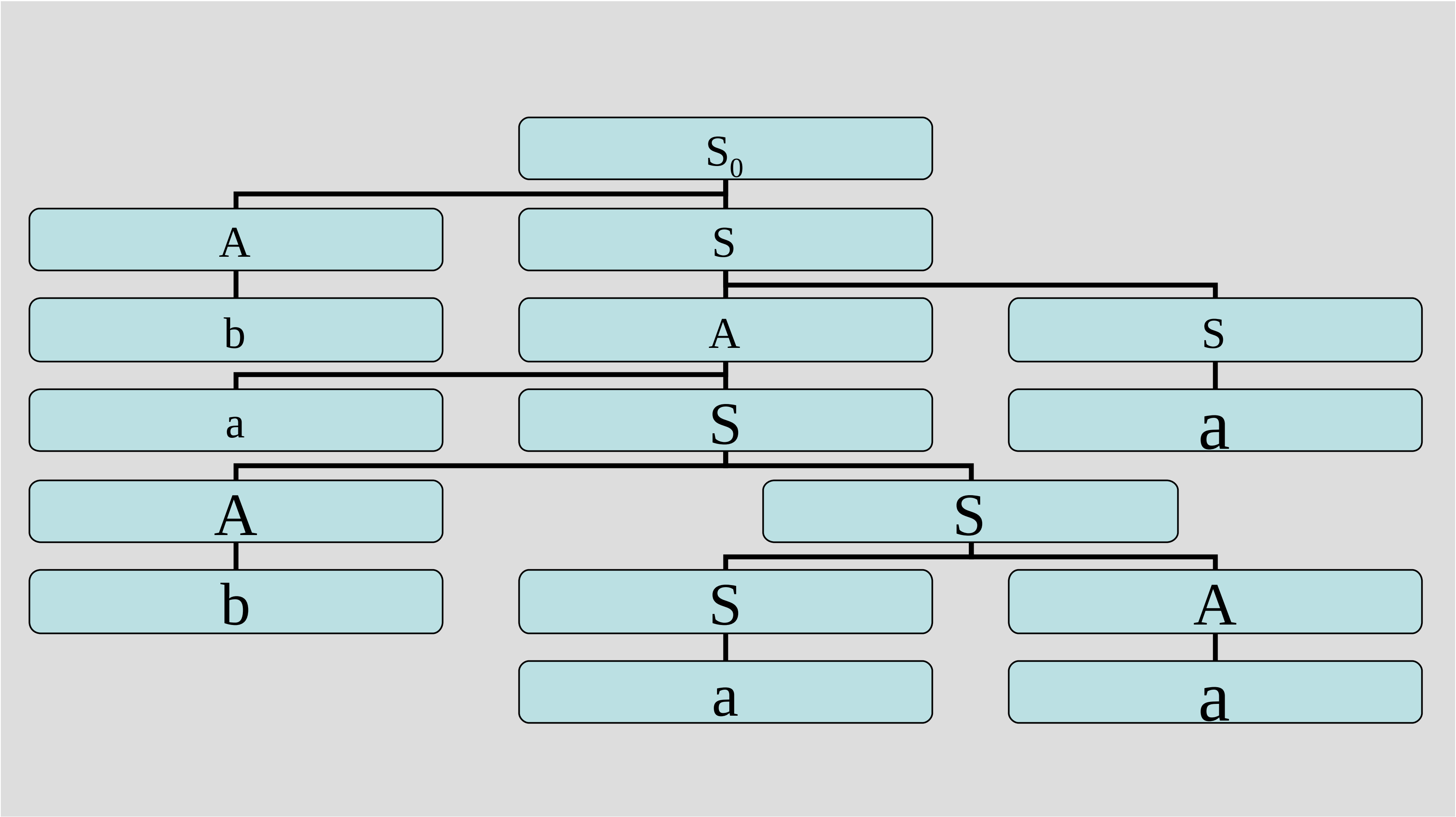
$A_2 \rightarrow a$

$S \rightarrow AA_1 \mid A_2B \mid a \mid AS \mid SA$

$A \rightarrow b \mid AA_1 \mid A_2B \mid a \mid AS \mid SA$

$B \rightarrow b$

Grammar as a Parse Tree



Why is this useful?

- Because we know lots of things about binary trees
- We can now apply these things to context-free grammars since any CFG can be placed into the CNF format
- For example
  - If yield of the tree is a terminal string w
  - If n is the height of the longest path in the tree
  - Then $|w| \leq 2^{n-1}$

Non-context-free languages

- Certain languages are not context-free. For example:
  - The language $B = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free
  - The language $C = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$ is not context-free

Turing Machines and Complexity

Devices of Increasing Computational Power

- So far:
  - Finite Automata – good for devices with small amounts of memory, relatively simple control
  - Pushdown Automata – stack-based automata
- But both have limitations for even simple tasks, too restrictive as general purpose computers
- Enter the **Turing Machine**
  - First proposed by Alan Turing in 1936
  - More powerful than either of the above
  - Essentially a finite automaton but with unlimited memory
  - Although theoretical, can do everything a general purpose computer of today can do
    - **If a TM can't solve it, neither can a computer**
- What does it mean to be able to compute a function?
  - Alonzo Church and Alan Turing independently arrived at equivalent conclusions: *A function is computable if it can be computed by a Turing machine*.

Computability

- We will start to examine **problems** that are **at the threshold** and **beyond the theoretical limits of what is possible to compute using computers today**.

- We will examine the following issues with the help of TM's

Recursively Enumerable Languages

- We use the simplicity of the TM model to prove formally that there are specific problems (i.e. languages) that the TM cannot solve.

  When we start a TM on an input, three outcomes are possible: The machine may ***accept***, ***reject***, ***loop*** (the machine simply does not halt)

  Two classes of languages:

  - **recursively enumerable** : TM can **accept** the strings in the language but **cannot tell for certain that a string is not in the language**. Sometimes these are called "decidable" or "non-decidable" problems.

  - **non-recursively enumerable** : **no TM can even recognize the members of the language**.

P and NP

- We then look at problems (languages) that do have **TM's that accept them and always halt**;
  - i.e. they not only recognize the strings in the language, but they tell us when they are sure the string is not in the language.

- The classes **P** and **NP** are those languages recognizable by deterministic and nondeterministic TM's, respectively, that halt within a time that is some polynomial in the input.
  - Polynomial is as close as we can get, because real computers and different models of (deterministic) TM's can differ in their running time by a polynomial function, e.g., a problem might take $O(n^2)$ time on a real computer and $O(n^6)$ time on a TM

P and NP

- P is the class of languages that are decidable in polynomial time on a deterministic (single-tape) Turing machine.
  - Example: A directed graph G contains nodes *s* and *t*. The PATH problem is to determine whether a directed path exists from *s* to *t*.
  - We can in many cases/problems avoid brute-force search and obtain polynomial time solutions.

P and NP

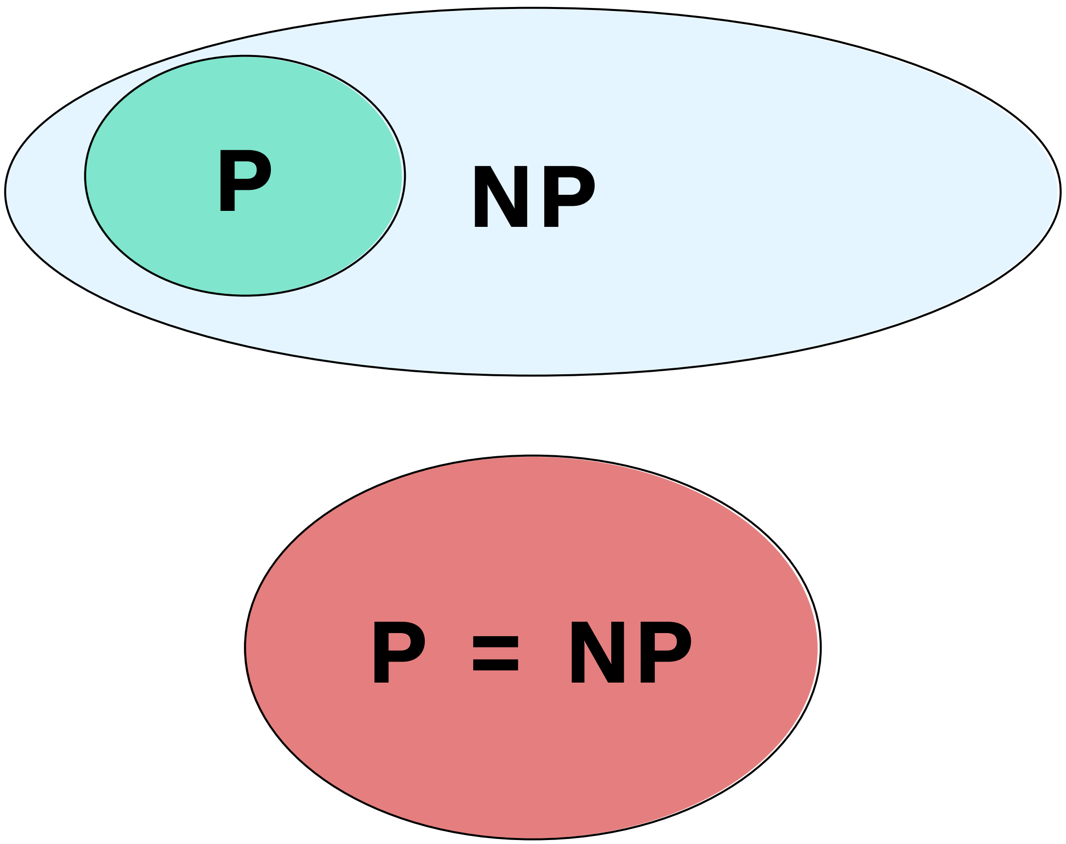
- Attempts to avoid brute force in many problems have not been successful, and polynomial time algorithms that solve them are not known to exist.
- Why have "we" been unsuccessful in finding polynomial time algorithms for these problems?
- The complexity of many problems are linked. The discovery of a polynomial time algorithm for one such problem can be used to solve an entire class of problems.

P = the class of languages where membership can be ***decided*** quickly

NP = the class of languages where membership can be ***verified*** quickly.

The P versus NP question

We are unable to prove the existence of a single language in NP that is not in P.



?

The question whether P=NP is one of the greatest unsolved problems in theoretical CS and contemporary mathematics.

NP Complete

- These are in a sense the "**hardest**" **problems in NP**.
  - These problems correspond to languages that are recognizable by a nondeterministic TM.
  - However, we will also be able to show that in polynomial time we can reduce any NP-complete problem to any other problem in NP.
    - This means that if we could prove an NP Complete problem to be solvable in polynomial time, then P = NP.
  - We will examine some specific problems that are NP-complete: satisfiability of Boolean (propositional logic) formulas, traveling salesman, etc.

Intuitive Argument for an Undecidable Problem

- Given a program that prints “hello, world” **is there another program that can test if a program given as input prints** “hello, world”?

- This is *tougher* than it may sound at first glance. For some programs it is easy to determine if it prints hello world. Here is perhaps the simplest:

```
#include “stdio.h”
void main()
{
    printf(“hello, world\n”);
}
```

Not as easy as it looks…

- It would be fairly easy to write a program to test to see if another program consisting solely of printf statements will output "hello, world". But what we want is a **program that can take any arbitrary program** and determine if it prints "hello, world".

- This is much more difficult. Consider the following program:

Obfuscated Hello World Program

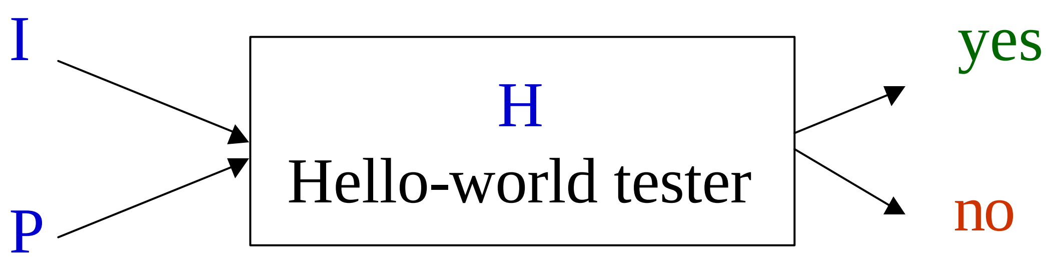
```
#include "stdio.h"
#define e 3
#define g (e/e)
#define h ((g+e)/2)
#define f (e-g-h)
#define j (e*e-g)
#define k (j-h)
#define l(x) tab2[x]/h
#define m(n,a) ((n&(a))==(a))
long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };

main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s;
    if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); }
    else switch(m1-=h){
        case f:
            a=(b=(c=(d=g)<<g)<<g)<<g;
            return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
        case h:
            for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
        case g:
            if(n<h)return(g);
            if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
            else{c='\r'-'\b';n-=j-g;o[f]=o[g]=g;}
            if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
            return(o[b-g]%n+k-h);
        default:
            if(m1-=e) main(m1-g+e+h,s+g); else *(s+g)=f;
            for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char *)m1);
        }
}
```

Hello World Tester

- Problem: Create a program that determines if any arbitrary program prints “hello world”
- We can show there is no program to solve that problem (called **undecidable**)
- Suppose that there were such a program H, the “hello-world-tester."
- H takes as input a program P and an input file I for that program, and tells whether P, with input I, prints “hello world” and outputs “yes” if it does, “no” if it does not

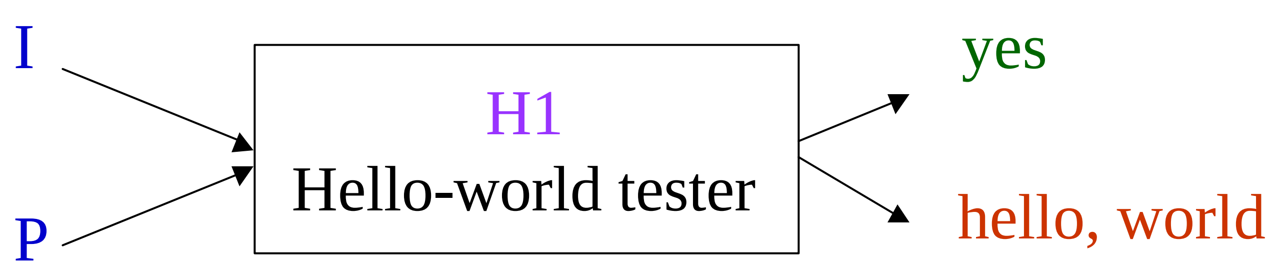
Hello World Tester



Hello World Tester

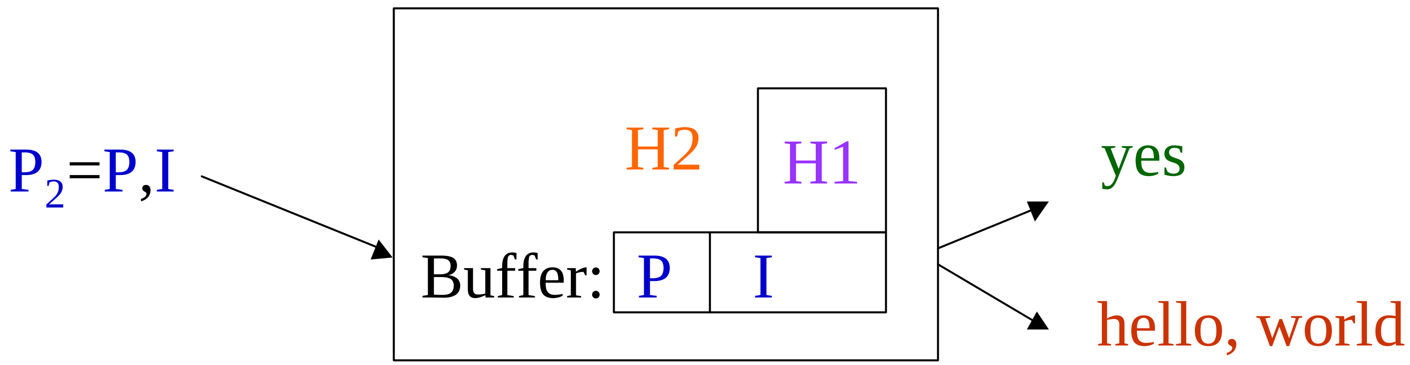
- Next we modify H to a new program H1 that acts like H, but when H prints no, H1 prints "hello, world."

- To do this, we need to find where "no" is printed and instead output "hello world" instead:



Hello World Tester

- Next modify H1 to H2 . The program H2 takes only one input, P2, instead of both P and I.
- To do this, the new input P2 must include the data input I and the program P.
- The program P and data input I are all stored in a buffer in program H2. H2 then simulates H1, but whenever H1 reads input, H2 feeds the input from the buffered copy. H2 can maintain two index pointers into the buffered data to know what current data and code should be read next:



Hello World Tester

- However, **H2 cannot exist**. If it did, what would H2(*H2*) do?
- That is, we give H2 as input to itself:

If *H2* on the left outputs = "yes", then H2 given *H2* as input will print "hello, world". But we just supposed that the first output *H2* makes is "yes" and not "hello world".

The situation is paradoxical and we conclude that H2 cannot exist and this problem is **undecidable**.

Problem Reducibility

- Once we have a single problem known to be undecidable we can determine that other problems are also undecidable by **reducing** a known undecidable problem to the new problem.
  - We will use this same idea later when we talk about proving problems to be NP-Complete.
- To use this idea, we must take a problem we know to be undecidable. Call this problem U. Given a new problem, P, if U can be reduced to P so that P can be used to solve U, then P must also be undecidable.

Problem Reducibility

- Important – we must show that U reduces to P, not vice versa
  - If we show that our P reduces to U then we have only shown that a new problem can be solved by the undecidable problem
  - It might still be possible to solve problem P by other means; e.g. we might be taking the tough path to solve P
- But if we can show the other direction, that P can solve U, then P must be at least as hard as U, which we already know to be undecidable.

Reducibility Example

- Does program Q ever call function foo?
  - This problem is also undecidable
- Just as we saw with the ‘hello world’ problem, it is easy to write a program that can determine if some programs call function foo.
- But we could have a program that contains lots of control logic to determine whether or not function foo is invoked. This general case is much harder, and in fact undecidable

Reducibility Example

- Use the reduction technique for the Hello-World problem
  - Rename the function "foo" in program Q and all calls to that function.
  - Add a function "foo" that does nothing and is not called.
  - Modify the program to remember the first 12 characters that it prints, storing them in array A
  - Modify the program so that whenever it executes any output statement, it checks the array A to see if the 12 characters written are "hello, world" and if so, invokes function foo.
  - If the final program prints "hello, world" then it must also invoke function foo. Similarly, if the program does not print "hello, world" then it does not invoke foo.

Foo Caller

- Say that we have a program F-Test that can determine if a program calls foo.
- If we run F-Test on the modified program above, not only can it determine if a program calls foo, it can also determine if the program prints “hello, world”.
- But we would then be solving the “hello-world-tester” problem which we already know is undecidable, therefore our F-Test problem must be undecidable as well