Announcements
- Final exam review session this weekend
  - Friday 1-5 pm, room TBA
  - See Piazza Poll to vote on additional times
- Potential extra credit – more information later in the week

Logic Language Review
Expressions begin with `query` or `fact` followed by relations:

- `logic>` `(fact (parent eisenhower fillmore))`
- `logic>` `(fact (parent fillmore abraham))`
- `logic>` `(fact (ancestor ?a ?y) (parent ?a ?y))`
- `logic>` `(query (ancestor ?who abraham))`

Success!
who: fillmore
who: eisenhower

If a fact has more than one relation, the first is the conclusion, and it is satisfied if the remaining relations, the hypotheses, are satisfied.

If a query has more than one relation, all must be satisfied.

The interpreter lists all bindings that it can find to satisfy the query.

Example: Combining Multiple Data Sources
Which dogs have an ancestor of the same color?


Success!
name: barack color: tan ancestor: eisenhower
name: clinton color: white ancestor: abraham
name: grover color: tan ancestor: eisenhower
name: herbert color: brown ancestor: fillmore

Example: Appending Lists
Two lists append to form a third list if:
- The first list is empty and the second and third are the same:
  `() (a b c)`
- Both of the following hold:
  - List 1 and 3 have the same first element
  - The rest of list 1 and all of list 2 append to form the rest of list 3

- `logic>` `(fact (append-to-form () ?x ?y))`
- `logic>` `(fact (append-to-form (a b) (c d e)) (a b c d e f))`
Pattern Matching
The basic operation of the Logic interpreter is to attempt to unify two relations
Unification is finding an assignment to variables that makes two relations the same

| (a b) c (a b) | True, {x: (a b)} |
| a ?x c ?x | True, {x: (a b)} |
| (a b) c (a b) | True, {y: b, z: c} |
| (a b) c ?z (a b) | False |

( ?x ?x ?x )

Substituting values for variables may require multiple steps

lookups ' ?x' (a ?y c) lookup ' ?y' (a) b

Unification with Two Variables
Two relations that contain variables can be unified as well

( ?x ?y ?z ) True, {x: (a ?y c), y: b, z: c}

Substituting values for variables may require multiple steps

lookups ' ?x' (a ?y c) lookup ' ?y' (a) b

Unification

Unification unifies each pair of corresponding elements in two relations, accumulating an assignment

1. Look up variables in the current environment
2. Establish new bindings to unify elements

Unification with Two Variables
Two relations that contain variables can be unified as well

( ?x ?x ) True, {x: (a b)}

Substituting values for variables may require multiple steps

lookups ' ?x' (a ?y c) lookup ' ?y' (a) b

Searching for Proofs
The Logic interpreter searches the space of facts to find unifying facts

An assignment to variables may require multiple steps

lookups ' ?x' (a ?y c) lookup ' ?y' (a) b

Underspecified Queries
Now that we know about unification, let’s look at an underspecified query
What are the results of these queries?

> (fact (append-to-form (1 2) (3)))
> (fact (append-to-form (1 2 . ?x) ?x (1 2 . ?x)))
> (query (append-to-form (1 2 . ?x) ?x (1 2 . ?x)))
Success!
what: [(1 2 3)]
> (query (append-to-form (1 2 . ?x) ?x (1 2 . ?x)))
Success!
what: [(1 2 3)]

Variables are local to facts and queries

lookups ' ?x' (a ?y c) lookup ' ?y' (a) b
Search for possible unification
The space of facts is searched exhaustively, starting from the query and following a depth-first search order.

A possible proof is explored exhaustively before another one is considered.

```
def search(clauses, env):
    for fact in facts:
        env_head = unify(conclusion of fact, first clause, env)
        if unification succeeds:
            env_rule = search(hypotheses of fact, env_head)
            result = search(rest of clauses, env_rule)
            yield each result
```

Some good ideas:
- Limiting depth of the search avoids infinite loops
- Each time a fact is used, its variables are renamed
- Bindings are stored in separate frames to allow backtracking

Implementing Search
```
def search(clauses, env, depth):
    if clauses is nil:
        yield env
    env[DEPTH LIMIT] is None or depth < DEPTH LIMIT:
        for fact in facts:
            #...where appropriate
        env_head = Frame(env)
        if unify(fact.first, clauses.first, env_head):
            for env_rule in search(fact.second, env_head, depth+1):
                for result in search(clauses.second, env_rule, depth+1):
                    yield result
    else:
        #...where appropriate
```

An Evaluator in Logic
We can define an evaluator in Logic; for example, we define numbers:

```
def fact (ints 1 2)
    fact (ints 3 4)
    fact (ints 4 5)
```

Then we define addition:

```
def fact (add 1 7 2) (ints 7 7)
    for (add 7 7) (add 7 7 7) (add 7 7 7 7):
        fact = yield
```

Finally, we define the evaluator:

```
def fact (eval 1 x z) (ints 1 x)
    fact (eval (+ v0 v2) v3)
    fact (eval (+ v1 v2) v3)
```

We define the evaluator:

```
def fact (eval (+ 1 (+ ?what 2)) 5)
    eval (add ?x ?y ?z)
```

The Halting Problem

```
Robert Huang
August 7, 2013
```

Function Streams
Given a stream of 1-argument functions, we can construct a function that is not in the stream, assuming that all functions in the stream terminate.

```
def func_not_in_stream(s):
    return lambda n: not s[n](m)
```
## Programs and Mathematical Functions

A mathematical function \(f(x)\) maps elements from its input domain \(D\) to its output range \(R\):

\[
f : D \rightarrow \{0, 1\}, \quad f(x) = x^2 \mod 2
\]

A Python function \(\text{func}\) computes a mathematical function \(f\) if the following conditions hold:

- \(\text{func}\) has the same number of parameters as inputs to \(f\)
- \(\text{func}\) terminates on every input in \(D\)
- The return value of \(\text{func}(x)\) is the same as \(f(x)\) for all \(x\) in \(D\)

```python
def func(x):
    return (x * x) % 2
```

A mathematical function \(f\) is computable if there exists a program (i.e., a Python function) \(\text{func}\) that computes it.

## Turing... what?

```python
def turing(f):
    while True:  # infinite loop
        pass
    else:
        return True  # halts

turing(turing)  # * what?
```

If this sounds fishy, it should. Should the call \(\text{turing}(\text{turing})\) halt or go into an infinite loop?

- \(\text{turing}(\text{turing})\) loops \(\rightarrow \text{halts}(\text{turing}, \text{turing})\) returns true
- However, \(\text{turing}(\text{turing})\) should have halted

- \(\text{turing}(\text{turing})\) halts \(\rightarrow \text{halts}(\text{turing}, \text{turing})\) returns false
- However, \(\text{turing}(\text{turing})\) should not have halted

We have a contradiction! Our assumption that \(\text{halts}\) exists is false.

## Bitstrings and Functions

Let's develop another proof, assuming that we have a \(\text{halts}\) program that computes the mathematical function \(\text{halts}\).

Let's create a stream of all 1-argument Python functions, then use \(\text{halts}\) to filter out non-terminating programs from that stream.

Assume we have the following Python functions:

```python
def is_valid_python_function(bitstring):
    """Determine whether or not a bitstring represents a syntactically valid 1-argument Python function.""

def bitstring_to_python_function(bitstring):
    """Coerce a bitstring representation of a Python function to the function itself.""
```
Bitstrings and Functions

Let’s develop another proof, assuming that we have a `halts` program that computes the mathematical function `halts`.

Let’s create a stream of all 1-argument Python functions, then use `halts` to filter out non-terminating programs from that stream.

Then the following produces all valid 1-argument Python functions:

```python
def function_stream():
    """Return a stream of all valid 1-argument Python functions."""
    bitstring_stream = iter_to_stream(bitstrings())
    valid_stream = filter_stream(in_valid_python_function, bitstring_stream)
    return map_stream(bitstring_to_python_function, valid_stream)
```

Filtering Out Non-Terminating Programs

With `halts`, we can’t filter out programs that don’t halt on all input. But we can filter out programs that don’t halt on a specific input. Specifically, let’s make sure that a program halts on its index in the resulting stream of programs:

```python
def make_halt_checker():
    index = 0
    def halt_checker(fn):
        nonlocal index
        if halts(fn, index):
            index += 1
            return True
        return False
    halt_checker

programs = filter_stream(make_halt_checker(), function_stream())
```

Developing a Contradiction

We now have a stream of programs that halt when given their own index as input:

```python
programs = filter_stream(make_halt_checker(), function_stream())
```

Recall the following function that produces a function that is not in a given stream:

```python
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

Consider the following:

```python
church = func_not_in_stream(programs)
```

Does `church` appear anywhere in `programs`?

Every element in `programs` halts when given its own index as input. Thus, `church` halts on all inputs `n`, since it calls the `n`th element in `programs` on `n`.

If `church` is in `programs`, it has an index `m`, so what does `church(m)` do?

It calls the `m`th element in `programs`, which is `church` itself, on `m`.

This results in an infinite loop, which means `halt_checker` will return false on `church`, since it does not halt given its own index.

Developing a Contradiction

```python
def func_not_in_stream(s):
    return lambda n: not s[n](n)

church = func_not_in_stream(programs)
```

Does `church` appear anywhere in `programs`?

Every element in `programs` halts when given its own index as input. Thus, `church` halts on all inputs `n`, since it calls the `n`th element in `programs` on `n`.

If `church` is in `programs`, it has an index `n`, so what does `church(n)` do?

It calls the `n`th element in `programs`, which is `church` itself, on `n`.

This results in an infinite loop, which means `halt_checker` will return false on `church`, since it does not halt given its own index.

Developing a Contradiction

```python
def func_not_in_stream(s):
    return lambda n: not s[n](n)

church = func_not_in_stream(programs)
```

Does `church` appear anywhere in `programs`?

Every element in `programs` halts when given its own index as input. Thus, `church` halts on all inputs `n`, since it calls the `n`th element in `programs` on `n`.

If `church` is in `programs`, it has an index `n`, so what does `church(n)` do?

It calls the `n`th element in `programs`, which is `church` itself, on `n`.

This results in an infinite loop, which means `halt_checker` will return false on `church`, since it does not halt given its own index.

Developing a Contradiction

```python
def func_not_in_stream(s):
    return lambda n: not s[n](n)

church = func_not_in_stream(programs)
```

We have a contradiction!

```python
halt_checker(church) returns true, which means that church is in programs.
```

But if `church` is in `programs`, then `church(m)`, where `m` is `church`’s index in `programs`, is an infinite loop, so `halt_checker(church)` returns false.

So we made a false assumption somewhere.
False Assumption

We assumed we had the following Python functions:
* `halts`
* `is_valid_python_function`
* `bitstring_to_python_function`

Everything else we wrote ourselves.

The latter two functions can be built using components of the interpreter.

Thus, it is our assumption that there is a Python function that computes halts that is invalid.

\[
\text{halts} : \text{Programs} \times \mathbb{N} \rightarrow \{0, 1\},
\]
\[
\text{halts}(P, n) = \begin{cases} 
1 & \text{if } P \text{ halts on input } n \\
0 & \text{otherwise}
\end{cases}
\]

Uncomputable Functions

It gets worse; not only can we not determine programmatically whether or not a given program halts, we can’t determine anything “interesting” about the behavior of a program in general.

For example, suppose we had a program `prints_something` that determines whether or not a given program prints something to the screen when run on a specific input:

Then we can write `halts`:

```python
def halts(fn, i):
    delete all print calls from fn
    replace all return statements with print
    return prints_something(fn, i)
```

Since we know we can’t write `halts`, our assumption that we can write `prints_something` is false.

Incompleteness Theorem

In 1931, Kurt Gödel proved that any mathematical system that contains the theory of non-negative integers must be either incomplete or inconsistent:
* A system is incomplete if there are true facts that cannot be proven
* A system is inconsistent if there are false claims that can be proven

A proof is just a sequence of statements, which can be represented as bits:
* We can generate all proofs the same way we generated all programs.

It is also possible to check the validity of a proof using a computer:
* Given a finite set of axioms and inference rules, a program can check that each statement in a proof follows from the previous ones.

Thus, if a valid proof exists for a mathematical formula, then a computer can find it.

The Halting Problem

The question of whether or not a program halts on a given input is known as the halting problem.

In 1936, Alan Turing proved that the halting problem is unsolvable by a computer.

That is, the mathematical function `halts` is uncomputable.

\[
\text{halts} : \text{Programs} \times \mathbb{N} \rightarrow \{0, 1\},
\]
\[
\text{halts}(P, n) = \begin{cases} 
1 & \text{if } P \text{ halts on input } n \\
0 & \text{otherwise}
\end{cases}
\]

We proved that `halts` is uncomputable in Python, but our reasoning applies to all languages.

It is a fundamental limitation of all computers and programming languages.

Consequences

There are vast consequences from the impossibility of computing halts, or any other sufficiently interesting mathematical functions on programs:

The best we can do is approximation.

For example, perfect anti-virus software is impossible:
* Anti-virus software must either miss some viruses (false negatives), mark some innocent programs as viruses (false positives), or fail to terminate on others.

We can’t write perfect security analyzers, optimizing compilers, etc.

Incompleteness Theorem

Given a sufficiently powerful mathematical system, we can write the following formula, which is a predicate form of the halts function:

\[
H(P, n) = \text{“program } P \text{ halts on input } n\”
\]

If \( H(P, n) \) is provable or disprovable for all \( P \) and \( n \), then we can write a program to prove or disprove it by generating all proofs and checking each one to see if it proves or disproves \( H(P, n) \).

But then this program would solve the halting problem, which is impossible.

Thus, there must be values of \( P \) and \( n \) for which \( H(P, n) \) is neither provable nor disprovable, or for which an incorrect result can be proven.

Thus, there are fundamental limitations not only to computation, but to mathematics itself.
Interpretation in Python

**eval**: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.

**exec**: Executes a statement in the current environment. Doing so may affect the environment.

```python
eval('2 + 2')
exec('def square(x): return x * x')
```

**os.system('python <file>')**: Directs the operating system to invoke a new instance of the Python interpreter.