L12A: Logic Programming

CS1101S: Programming Methodology

Martin Henz

November 3, 2021

1. Some history
2. Deductive information retrieval
3. How the query system works
4. Implementing the query system

History of Logic Programming

Early history

Logic programming has its foundations in mathematical logic and A.I. (Carl Hewitt: Planner, 1969)

History of Logic Programming

Early history

Logic programming has its foundations in mathematical logic and A.I. (Carl Hewitt: Planner, 1969)

Discovery

Simultaneously, the 1970s:

- Bob Kowalski (Imperial College) discovered deduction for computation
- Alain Colmerauer (Marseille) developed *Prolog*, based on unification and deduction.

History of Logic Programming

A new programming "paradigm"

Logic programming is a different way of describing computation.

History of Logic Programming

A new programming "paradigm"

Logic programming is a different way of describing computation.
Maybe too different?

History of Logic Programming

A new programming "paradigm"

Logic programming is a different way of describing computation. Maybe too different?

Influence

Logic programming had a deep influence on

- database research
- programming languages (e.g. Oz)

Representing information with assertions

```
address(list("Bitdiddle","Ben"),
        list("Slumerville",list("Ridge","Road"),
             10))
```

```
job(list("Bitdiddle", "Ben"),
    list("computer", "wizard"))
```

```
salary(list("Bitdiddle", "Ben"), 60000)
```

- First symbols `address`, `job`, `salary` are the *kind of information*
- "arguments"—lists, literals—are *data*

More information

```
supervisor(list("Bitdiddle", "Ben"),
           list("Warbucks", "Oliver"))

address(list("Warbucks", "Oliver"),
        list("Swellesley",
             list("Top", "Heap", "Road")))

job(list("Warbucks", "Oliver"),
    list("administration", "big", "wheel"))

salary(list("Warbucks", "Oliver"), 150000)
```

...and more info on Alyssa P. Hacker, Cy D. Fect,...

More information

```
can_do_job(list("computer", "wizard"),
           list("computer", "programmer"))
can_do_job(list("computer", "wizard"),
           list("computer", "technician"))

can_do_job(list("computer", "programmer"),
           list("computer", "programmer",
                "trainee"))

can_do_job(list("administration","assistant"),
           list("administration","big","wheel"))
```

Simple queries

```
// Query input:
job($x, list("computer", "programmer"))

// Query results:
job(list("Hacker", "Alyssa", "P"),
    list("computer", "programmer"))

job(list("Fect", "Cy", "D"),
    list("computer", "programmer"))
```

Simple queries

```
// Query input:
job($x, list("computer", "programmer"))

// Query results:
job(list("Hacker", "Alyssa", "P"),
    list("computer", "programmer"))

job(list("Fect", "Cy", "D"),
    list("computer", "programmer"))
```

How do queries look like?

Simple queries

```
// Query input:
job($x, list("computer", "programmer"))

// Query results:
job(list("Hacker", "Alyssa", "P"),
    list("computer", "programmer"))

job(list("Fect", "Cy", "D"),
    list("computer", "programmer"))
```

How do queries look like?

Queries have the shape of assertions but can contain *logic variables*.

Another query

```
// Query input:
address($x, $y)
```

Another query

```
// Query input:
address($x, $y)

// Query input:
supervisor($x, $x)
```

Another query

```
// Query input:
address($x, $y)

// Query input:
supervisor($x, $x)
```

The notion of "matching"

An assertion matches a query if we can *instantiate* the query's logic variables with data and obtain the assertion.

Example matches

```
// Query input:
job($x, list("computer", $type))
```

Example matches

```
// Query input:
job($x, list("computer", $type))

// Query results:
job(list("Bitdiddle", "Ben"),
    list("computer", "wizard"))
job(list("Hacker", "Alyssa", "P"),
    list("computer", "programmer"))
job(list("Fect", "Cy", "D"),
    list("computer", "programmer"))
job(list("Tweakit", "Lem", "E"),
    list("computer", "technician"))
```

Queries with pairs

```
// Query input:
job($x, list("computer", $type))
```

does not match

```
job(list("Reasoner", "Louis"),
    list("computer", "programmer", "trainee"))
```

Queries with pairs

```
// Query input:
job($x, list("computer",  $type))
```

does not match

```
job(list("Reasoner", "Louis"),
    list("computer", "programmer", "trainee"))
```

but

```
// Query input:
job($x, pair("computer", $type))
```

matches the assertion!

Queries with pairs

```
// Query input:
job($x, list("computer",  $type))
```

does not match

```
job(list("Reasoner", "Louis"),
    list("computer", "programmer", "trainee"))
```

but

```
// Query input:
job($x, pair("computer", $type))
```

matches the assertion!
How?

Queries with pairs

```
// Query input:
job($x, list("computer",  $type))
```

does not match

```
job(list("Reasoner", "Louis"),
    list("computer", "programmer", "trainee"))
```

but

```
// Query input:
job($x, pair("computer", $type))
```

matches the assertion!
How? Substitute `$type` with `list("programmer", "trainee")`

Query processing

Query processing

- find all assignments to variables in the query pattern that satisfy the pattern.

Query processing

- find all assignments to variables in the query pattern that satisfy the pattern.
    - the kind of information specified in the pattern needs to match the kind of information in an assertion

Query processing

- find all assignments to variables in the query pattern that satisfy the pattern.
    - the kind of information specified in the pattern needs to match the kind of information in an assertion
    - the assertion must result from the pattern by instantiating the pattern variables with values

Query processing

- find all assignments to variables in the query pattern that satisfy the pattern.
    - the kind of information specified in the pattern needs to match the kind of information in an assertion
    - the assertion must result from the pattern by instantiating the pattern variables with values
- system responds to query by listing all instantiations of the query pattern with the variable assignments that satisfy it

Query processing

- find all assignments to variables in the query pattern that satisfy the pattern.
    - the kind of information specified in the pattern needs to match the kind of information in an assertion
    - the assertion must result from the pattern by instantiating the pattern variables with values
- system responds to query by listing all instantiations of the query pattern with the variable assignments that satisfy it

Special case

If the pattern has no variables, the query reduces to a determination of whether that pattern is in the data base. The *empty assignment* satisfies that pattern for that data base.

and queries

```
// Query input:
and(job($person, list("computer", "programmer")),
    address($person, $where))

// Query results:
and(job(list("Hacker", "Alyssa", "P"),
        list("computer", "programmer")),
    address(list("Hacker", "Alyssa", "P"),
            list("Cambridge", "Mass Ave", 78)))
and(job(list("Fect", "Cy", "D"),
        list("computer", "programmer")),
    address(list("Fect", "Cy", "D"),
            list("Cambridge",
                 list("Ames", "Street"), 3)))
```

Matching of for and queries

$\texttt{and}\,(\,query_1\,,\ \ query_2\,,\ \ \ldots\,,\ \ \ query_n)$

is satisfied by all sets of values for the pattern variables that simultaneously satisfy

$query_1\,,\ \ \ldots\,,\ \ query_n$

or queries

```
or(supervisor($x, list("Bitdiddle", "Ben")),
   supervisor($x, list("Hacker", "Alyssa", "P")))

// matches
or(supervisor(list("Hacker", "Alyssa", "P"),
              list("Bitdiddle", "Ben")),
   supervisor(list("Hacker", "Alyssa", "P"),
              list("Hacker", "Alyssa", "P")))
// and also
or(supervisor(list("Fect", "Cy", "D"),
              list("Bitdiddle", "Ben")),
   supervisor(list("Fect", "Cy", "D"),
              list("Hacker", "Alyssa", "P")))
...
```

Matching of `or` queries

`or` ($query_1$, $query_2$, ... , $query_n$)

is satisfied by all sets of values for the pattern variables that satisfy at least one of

$query_1$, ... , $query_n$

not queries

```
and(supervisor($x, list("Bitdiddle", "Ben")),
    not(job($x, list("computer", "programmer"))))
```

finds all people supervised by Ben Bitdiddle who are *not* computer programmers.

Matching `not` queries

`not` ($query_1$)

is satisfied by all assignments to the pattern variables that do not satisfy $query_1$.

javascript_predicate queries

```
javascript_predicate(predicate)
```

is satisfied by assignments to the pattern variables in the predicate for which the instantiated predicate is true.

```
and(salary($person, $amount),
    javascript_predicate($amount > 30000))

// find all people whose salary
// is greater than $30,000
```

Rules: Examples

```
rule(lives_near($person_1, $person_2),
     and(address($person_1, pair($town,$rest_1)),
         address($person_2, pair($town,$rest_2)),
         not(same($person_1, $person_2))))
```

Rules: Examples

```
rule(lives_near($person_1, $person_2),
     and(address($person_1, pair($town,$rest_1)),
         address($person_2, pair($town,$rest_2)),
         not(same($person_1, $person_2))))

rule(same($x, $x))
```

Rules: Examples

```
rule(lives_near($person_1, $person_2),
     and(address($person_1, pair($town,$rest_1)),
         address($person_2, pair($town,$rest_2)),
         not(same($person_1, $person_2))))

rule(same($x, $x))

rule(wheel($person),
     and(supervisor($middle_manager, $person),
         supervisor($x, $middle_manager)))
```

The shape or rules

General form of a rule

rule(conclusion, body)

where *conclusion* is a pattern and *body* is any query.

The shape or rules

General form of a rule

rule(conclusion, body)

where *conclusion* is a pattern and *body* is any query.

Meaning of rules

A rule represents a large (even infinite) set of assertions:
All instantiations of the rule conclusion with variable assignments that satisfy the rule body.

Rules example

```
rule(lives_near($person_1, $person_2),
     and(address($person_1, pair($town,$rest_1)),
         address($person_2, pair($town,$rest_2)),
         not(same($person_1, $person_2))))

// Query input:
lives_near($x, list("Bitdiddle", "Ben"))

// Query results:
lives_near(list("Reasoner", "Louis"),
           list("Bitdiddle", "Ben"))
lives_near(list("Aull", "DeWitt"),
           list("Bitdiddle", "Ben"))
```

Rules can be used in compound queries

```
and(job($x, pair("computer", something)),
    lives_near($x, list("Bitdiddle", "Ben")))
```

Recursive rules

```
rule(outranked_by($staff_person, $boss),
     or(supervisor($staff_person, $boss),
        and(supervisor($staff_person,
                       $middle_manager),
            outranked_by($middle_manager,
                         $boss))))
```

Appending lists in logic programming

Appending lists in logic programming

```
append($x, $y, $z)
```

Appending lists in logic programming

```
append($x, $y, $z)
```

means lists $x and $y append to form $z

Appending lists in logic programming

```
append($x, $y, $z)
```

means lists `$x` and `$y` append to form `$z`:

```
rule(append(null, $y, $y))
```

Appending lists in logic programming

```
append($x, $y, $z)
```

means lists $x and $y append to form $z:

```
rule(append(null, $y, $y))

rule(append(pair($u, $v), $y, pair($u, $z)),
     append($v, $y, $z))
```

Appending lists in logic programming

```
rule(append(null, y, y))

rule(append(pair(u, v), y, pair(u, z)),
     append(v, y, z))
```

Appending lists in logic programming

```
rule(append(null, y, y))

rule(append(pair(u, v), y, pair(u, z)),
     append(v, y, z))

// Query input:
append(list("a", "b"), list("c", "d"), $z)
```

Appending lists in logic programming

```
rule(append(null, y, y))

rule(append(pair(u, v), y, pair(u, z)),
     append(v, y, z))

// Query input:
append(list("a", "b"), list("c", "d"), $z)

  // Query results:
append(list("a", "b"), list("c", "d"),
              list("a", "b", "c", "d"))
```

Using rules backwards

```
rule(append(null, $y, $y))

rule(append(pair($u, $v), $y, pair($u, $z)),
     append($v, $y, $z))
```

Using rules backwards

```
rule(append(null, $y, $y))

rule(append(pair($u, $v), $y, pair($u, $z)),
     append($v, $y, $z))

// Query input:
append(list("a", "b"), $y,
       list("a", "b", "c", "d"))
```

Using rules backwards

```
rule(append(null, $y, $y))

rule(append(pair($u, $v), $y, pair($u, $z)),
     append($v, $y, $z))

// Query input:
append(list("a", "b"), $y,
       list("a", "b", "c", "d"))

// Query results:
append(list("a", "b"),list("c", "d"),
       list("a", "b", "c", "d"))
```

How many ways to make a given list?

```
rule(append(null, $y, $y))

rule(append(pair($u, $v), $y, pair($u, $z)),
     append(v, y, z))
```

How many ways to make a given list?

```
rule(append(null, $y, $y))

rule(append(pair($u, $v), $y, pair($u, $z)),
     append(v, y, z))

// Query input:
append($x, $y, list("a", "b", "c", "d"))
```

How many ways to make a given list?

```
rule(append(null, $y, $y))

rule(append(pair($u, $v), $y, pair($u, $z)),
     append(v, y, z))

// Query input:
append($x, $y, list("a", "b", "c", "d"))

// Query results:
append(null, list("a","b","c","d"), list("a","b","c","d"))
append(list("a"), list("b","c","d"), list("a","b","c","d"))
append(list("a","b"), list("c","d"), list("a","b","c","d"))
append(list("a","b","c"), list("d"), list("a","b","c","d"))
append(list("a","b","c","d"), null, list("a","b","c","d"))
```

Pattern matcher

Frames

Frame

keeps track of the bindings of logic variables

Frames

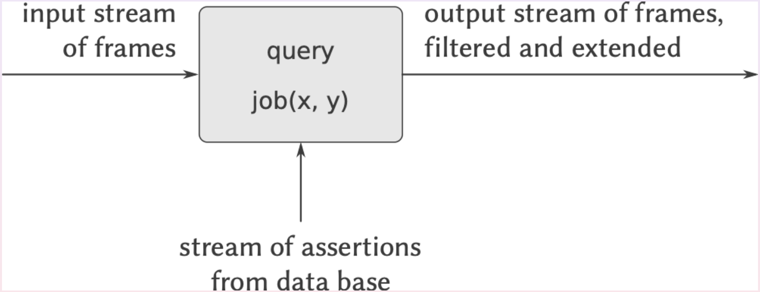
Frame

keeps track of the bindings of logic variables

Example frame

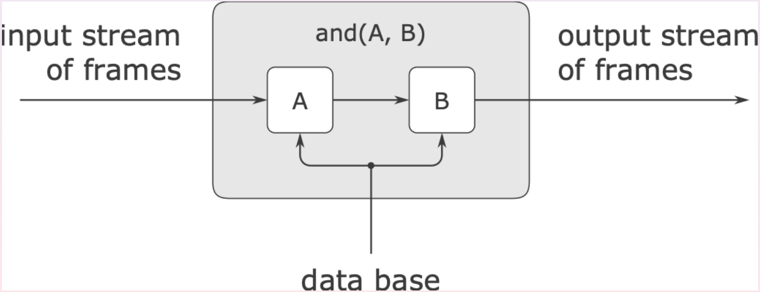
```
$x:       ["Reasoner", ["Louis", null]]
$type:    ["programmer", ["trainee", null]]
```

Streams of frames



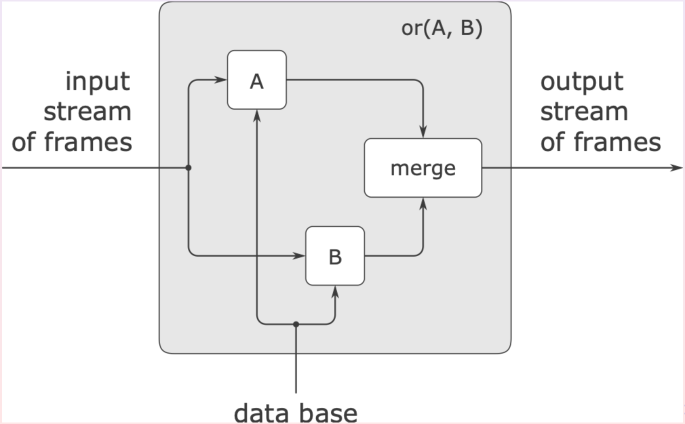
and queries

```
and(can_do_job($x, list("computer", "programmer",
                        "trainee")),
    job($person, $x))
```



or queries

```
or(supervisor($x, list("Bitdiddle", "Ben")),
   supervisor($x, list("Hacker", "Alyssa", "P")))
```



Handling rules: unification

Handle rules in query processing...

Handling rules: unification

Handle rules in query processing...

... find the rules whose conclusions match a given query pattern

Handling rules: unification

Handle rules in query processing...

... find the rules whose conclusions match a given query pattern

The job of a *unifier* for *two* patterns...

Handling rules: unification

Handle rules in query processing...

... find the rules whose conclusions match a given query pattern

The job of a *unifier* for *two* patterns...

... is to find a frame that makes the patterns equal

Unification example

Unify `list($x, $x)` and
`list(list("a", $y, "c"), list("a", "b", $z))`

Unification example

Unify `list($x, $x)` and
`list(list("a", $y, "c"), list("a", "b", $z))`

```
$x = list("a", $y, "c"), $x = list("a", "b", $z)
```

Unification example

Unify `list($x, $x)` and
`list(list("a", $y, "c"), list("a", "b", $z))`

```
$x = list("a", $y, "c"), $x = list("a", "b", $z)

list("a", $y, "c") = list("a", "b", $z)
```

Unification example

Unify `list($x, $x)` and
`list(list("a", $y, "c"), list("a", "b", $z))`

```
$x = list("a", $y, "c"), $x = list("a", "b", $z)

list("a", $y, "c") = list("a", "b", $z)

"a" = "a", $y = "b", "c" = $z
```

Unification example

Unify `list($x, $x)` and
`list(list("a", $y, "c"), list("a", "b", $z))`

```
$x = list("a", $y, "c"), $x = list("a", "b", $z)

list("a", $y, "c") = list("a", "b", $z)

"a" = "a", $y = "b", "c" = $z

$x = list("a", "b", "c")
```

Another example

Unify `list($x, "a")` and `list(list("b", $y), $z)`

Another example

Unify `list($x, "a")` and `list(list("b", $y), $z)`

```
$x = list("b", $y), "a" = $z
```

Another example

Unify `list($x, "a")` and `list(list("b", $y), $z)`

```
$x = list("b", $y), "a" = $z
```

Remaining binding: `$x` to pattern `list("b", $y)`

Applying rules

Applying rules in logic programming

- Unify the query with the conclusion of the rule to form, if successful, an extension of the original frame.
- Relative to the extended frame, evaluate the query formed by the body of the rule.

Applying rules

Applying rules in logic programming

- Unify the query with the conclusion of the rule to form, if successful, an extension of the original frame.
- Relative to the extended frame, evaluate the query formed by the body of the rule.

Applying functions in conventional programming

- Bind the function's parameters to its arguments to form a frame that extends the original function environment.
- Relative to the extended environment, evaluate the expression formed by the body of the function.

1. Some history
2. Deductive information retrieval
3. How the query system works
4. Implementing the query system

Driver loops

driver loop for conventional evaluator

Driver loop for query processing (simplified)

```
function query_driver_loop()             {
 const input = user_read(input_prompt);
 if (is_null(input)) {} else  {
  const q =convert_to_query_syntax(parse(input));
  if (is_assertion(q)) {
   add_rule_or_assertion(assertion_body(q));
  } else {
   display_stream(
    stream_map(
     frame=>unparse(instantiate_expr(exp,frame)),
     evaluate_query(q, singleton_stream(null))));
  }
  return query_driver_loop(); }       }
```

Driver loops for query processing

driver loop for query processing

Summary

Summary

- Assertions represent information in a data base

Summary

- Assertions represent information in a data base
- Queries allow us to retrieve information

Summary

- Assertions represent information in a data base
- Queries allow us to retrieve information
- Logic variables let us form patterns

Summary

- Assertions represent information in a data base
- Queries allow us to retrieve information
- Logic variables let us form patterns
- Compound queries let us form queries from queries

Summary

- Assertions represent information in a data base
- Queries allow us to retrieve information
- Logic variables let us form patterns
- Compound queries let us form queries from queries
- Rules let us describe (possibly infinite) collections of assertions

Summary

- Assertions represent information in a data base
- Queries allow us to retrieve information
- Logic variables let us form patterns
- Compound queries let us form queries from queries
- Rules let us describe (possibly infinite) collections of assertions
- We can *program* with this

Summary

- Assertions represent information in a data base
- Queries allow us to retrieve information
- Logic variables let us form patterns
- Compound queries let us form queries from queries
- Rules let us describe (possibly infinite) collections of assertions
- We can *program* with this in *interesting* ways!

Summary

- Assertions represent information in a data base
- Queries allow us to retrieve information
- Logic variables let us form patterns
- Compound queries let us form queries from queries
- Rules let us describe (possibly infinite) collections of assertions
- We can *program* with this in *interesting* ways!
- Query system is an interesting use case for stream processing