Leaning to Parse Database Queries Using Inductive Logic Programming

John M. Zelle and Raymond J. Mooney

Presented by Lena Dankin

Plan

- Task definition
- General background:
  - Prolog
  - Shift Reduce Parsing
- CHILL parser
- ILP (Inductive logic programming)
  - CHILLIN algorithm
- Experiments and Results

Plan

- Task definition
- General background:
  - Prolog
  - Shift Reduce Parsing
- CHILL parser
- ILP (Inductive logic programming)
  - CHILLIN algorithm
- Experiments and Results

Task Definition

- Executable semantic parsing:
  - Natural language ⇨ Executable DB query

How many people live in Iowa?
=> answer(P, (population(S,P), equal(S,stateid(iowa)))).

What is the capital of the state with the largest population?
=> answer(C, (capital(S,C), largest(P, (state(S), population(S,P))))).

Task Definition

- Development of the database application requires two components:
  - A framework for parsing the natural language query into the logical query representations
  - A specific query language for the our database (domain specific)

Geobase – in short

- A database in USA geography
- contains about 800 **Prolog** facts asserting relational tables for basic information about U.S. states, including:
  - Population
  - Area
  - capital city
  - neighboring states
  - major rivers
  - etc

Plan

- Task definition
- General background:
  - Prolog
  - Shift Reduce Parsing
- CHILL parser
- ILP (Inductive logic programming)
  - CHILLIN algorithm
- Experiments and Results

Prolog

- Prolog is a logic programming language.
- Prolog consists of a series of rules and facts.
- A program is run by presenting some query and checking if it can be proved against these known rules and facts.

From: lyle.smu.edu/~mhd/2353sp09/prolog.pptx

Prolog: hands on

- A prolog rule base example:

facts

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).
```

rules

```
sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

From: https://en.wikipedia.org/wiki/Prolog#Rules_and_facts

Prolog: hands on

- Prolog queries:

```
?- mother_child(trude, tom).
false.

?- sibling(sally, erica).
true .

?- mother_child(X, sally).
X = trude.

?- sibling(sally, X).
X = sally ;
X = erica ;
X = sally.
```

Rule base:

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X),
                      parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

Prolog: hands on

- List notation with *head* and *tail*

```
?- [1,2|X] = [1,2,3,4,5].
X = [3, 4, 5]
```

Will be used later!

Back to Geobase

- Now that we know Prolog…

<u>Geobase predicates:</u>

```
state(name, abbreviation, capital, population, area,
                    state_number, city1, city2, city3, city4)
city(state, state_abbreviation, name, population)
river(name, length, [states through which it flows])
border(state, state_abbreviation, [states that border it])
highlow(state, state_abbreviation, highest_point,
highest_elevation, lowest_point,
                    lowest_elevation)
mountain(state, state_abbreviation, name, height)
road(number, [states it passes through])
lake(name, area, [states it is in])
```

Database available at: https://www.cs.utexas.edu/users/ml/nldata/geoquery.html

GeoQuery

- In order to express interesting questions about geography, we need a query language having a vocabulary sufficient for expressing interesting questions about geography
- GeoQuery – a query language used for our task (all predicates are, naturally, implemented in Prolog)

GeoQuery

- Predicated for basic objects:

Type	Form	Example
country	`countryid(Name)`	`countryid(usa)`
city	`cityid(Name, State)`	`cityid(austin,tx)`
state	`stateid(Name)`	`stateid(texas)`
river	`riverid(Name)`	`riverid(colorado)`
place	`placeid(Name)`	`placeid(pacific)`

GeoQuery

- Predicated for basic relations:

Form	Predicate
capital(C)	C is a capital (city).
city(C)	C is a city.
major(X)	X is major.
place(P)	P is a place.
river(R)	R is a river.
state(S)	S is a state.
capital(C)	C is a capital (city).
area(S,A)	The area of S is A.
capital(S,C)	The capital of S is C.
equal(V,C)	variable V is ground term C.
density(S,D)	The (population) density of S is P
elevation(P,E)	The elevation of P is E.
high_point(S,P)	The highest point of S is P.
higher(P1,P2)	P1's elevation is greater than P2's.
loc(X,Y)	X is located in Y.
low_point(S,P)	The lowest point of S is P.
len(R,L)	The length of R is L.
next_to(S1,S2)	S1 is next to S2.
size(X,Y)	The size of X is Y.
traverse(R,S)	R traverses S.

GeoQuery

- Meta predicates - distinguished in that they take completely-formed conjunctive goals as one of their arguments

Form	Explanation
`answer(V,Goal)`	`V` is the variable of interest in `Goal`.
`largest(V, Goal)`	`Goal` produces only the solution that maximizes the size of `V`
`smallest(V,Goal)`	Analogous to `largest`.
`highest(V,Goal)`	Like `largest` (with elevation).
`lowest(V,Goal)`	Analogous to `highest`.
`longest(V,Goal)`	Like `largest` (with length).
`shortest(V,Goal)`	Analogous to `longest`.
`count(D,Goal,C)`	`C` is count of unique bindings for `D` that satisfy `Goal`.
`most(X,D,Goal)`	`Goal` produces only the `X` that maximizes the count of `D`
`fewest(X,D,Goal)`	Analogous to `most`.

GeoQuery

- How many people live in Iowa?
- => answer(P, (population(S,P), equal(S,stateid(iowa)))).

A *GeoQuery* guery to be executed on *Geobase.*
The variable **P** holds the answer

Shift Reduce Parser

- <u>Goal of parser</u>:

  <u>Input</u>:

  Grammar, linear input text

  <u>Output</u>:

  The grammatical structure of linear input text
- Bottom-up parser : build a derivation by working from the input back toward the start symbol
  - Builds parse tree from leaves to root
  - Builds reverse rightmost derivation

Shift-Reduce

- A shift-reduce parser uses two data structures:
  - An input buffer to store words of a sentence that have not yet been examined.
  - A stack which stores information concerning sentence constituents that have been recognized so far.

- Initially, the stack is empty, and the input buffer contains all of the words of a sentence to be processed.

- the process of parsing a sentence is a search problem. The parser must find a sequence of operators that transforms the initial state into a final representation.

Shift-Reduce parsing

- For the given grammar we define:
  - Terminal: num, id, +, *
  - Non terminals: S, E

1. $S \rightarrow E$
2. $E \rightarrow E$ **+** E
3. $E \rightarrow E$ ***** E
4. $E \rightarrow$ **num**
5. $E \rightarrow$ **id**

<u>A handle</u>:

- matches the rhs (right hand side) of some rule
- allows further reductions back to the start symbol

Shift-reduce parser

- Two questions
  1. Have we reached the end of handles and how long is the handle?
  2. Which non-terminal does the handle reduce to?
- We use tables to answer the questions
  - ACTION table
  - GOTO table

Shift-Reduce parsing

- A shift-reduce parser has 4 actions:
  - Shift -- next input token is shifted onto the stack
  - Reduce -- handle is at top of stack
    - pop handle
    - push appropriate lhs

$$\underbrace{E}_{\text{lhs}} \rightarrow \underbrace{E * E}_{\text{Handle (rhs)}}$$

  - Accept -- stop parsing & report success
  - Error -- call error reporting/recovery routine

from https://www.cs.northwestern.edu/academics/courses/322/notes/05.ppt

Example: Shift-reduce parsing

Grammar:

1. S → E
2. E → E + E
3. E → E * E
4. E → num
5. E → id

Input to parse:

id_1 + num * id_2

Handles:

underlined

STACK	ACTION
$	Shift
$ id_1	Reduce (rule 5)
$ E	Shift
$ E +	Shift
$ E + num	Reduce (rule 4)
$ E + E	Shift
$ E + E *	Shift
$ E + E * id_2	Reduce (rule 5)
$ E + E * E	Reduce (rule 3)
$ E + E	Reduce (rule 2)
$ E	Reduce (rule 1)
$ S	Accept

from https://www.cs.northwestern.edu/academics/courses/322/notes/05.ppt

Plan

- Task definition
- General background:
  - Prolog
  - Shift Reduce Parsing
- CHILL parser
- ILP (Inductive logic programming)
  - CHILLIN algorithm
- Experiments and Results

CHILL

- CHILL (Constructive Heuristics Induction for Language Learning):

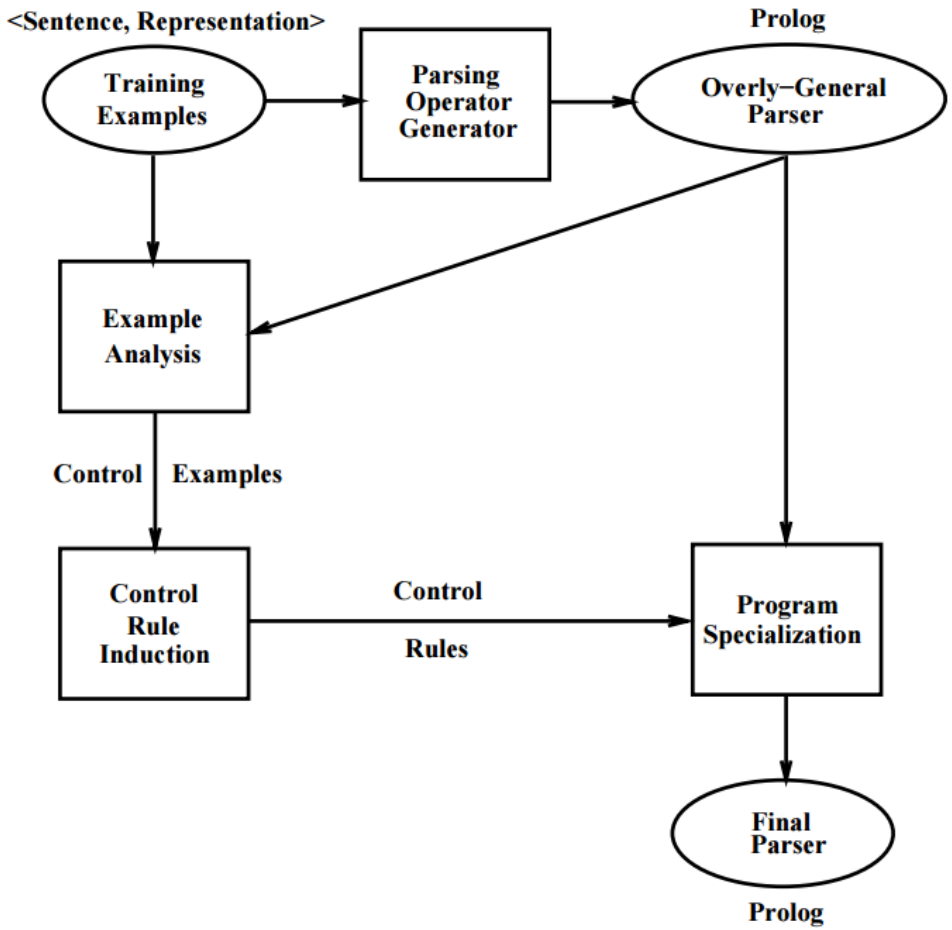
  Input:

  a set of training instances consisting of sentences paired with the desired parses

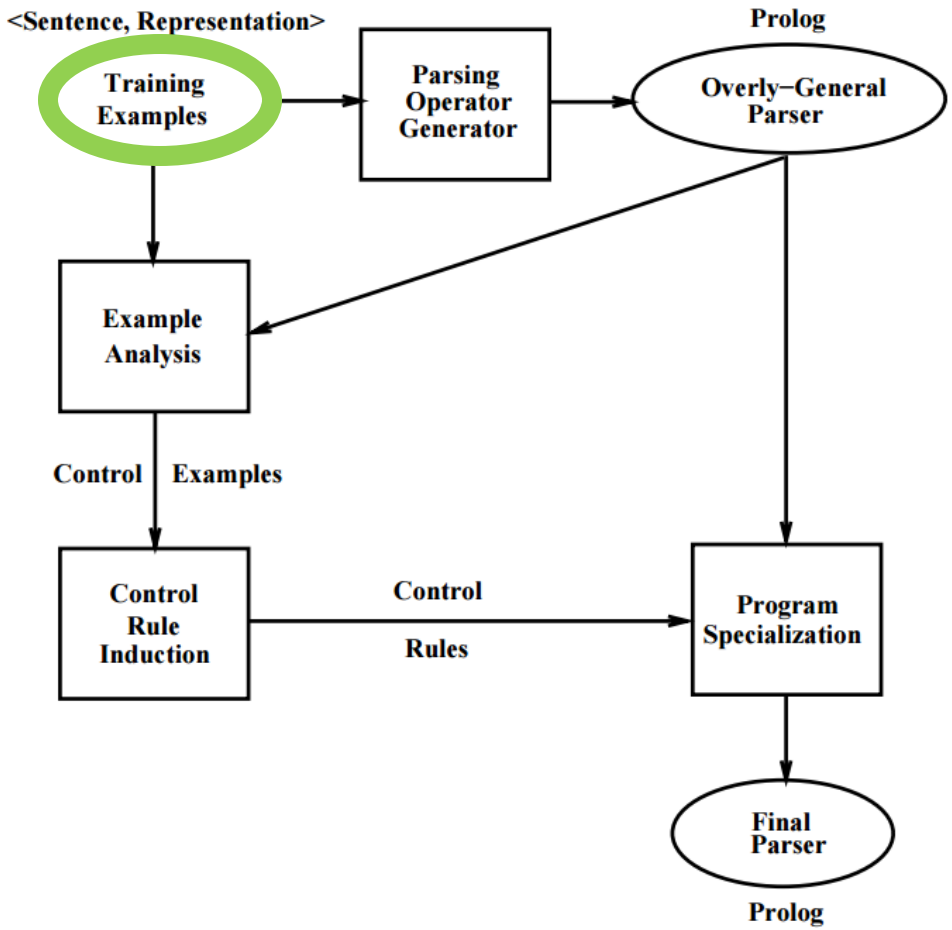
  Output:

  a deterministic shift-reduce parser in Prolog which maps sentences into parses

CHILL


<Sentence, Representation>
Training Examples
Parsing Operator Generator
Prolog
Overly−General Parser
Example Analysis
Control
Examples
Control Rule Induction
Control
Rules
Program Specialization
Final Parser
Prolog

CHILL

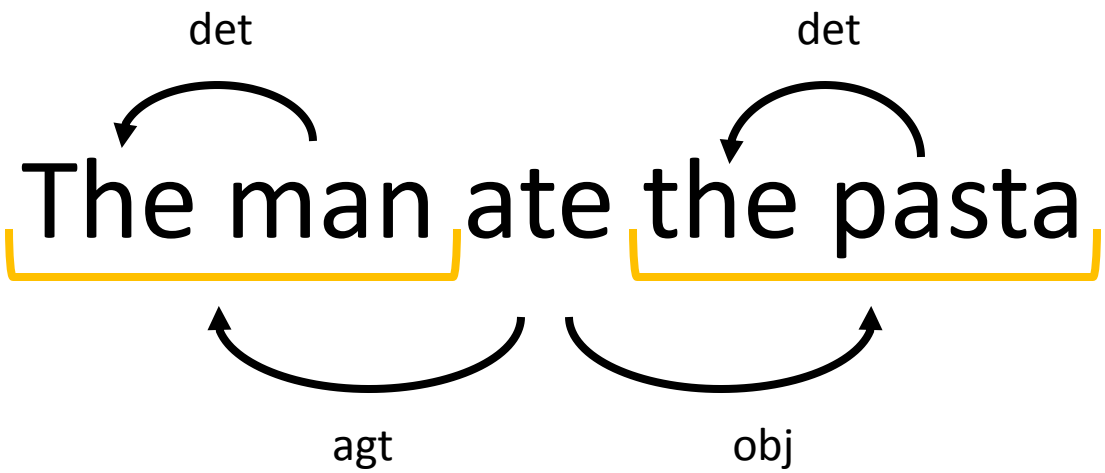

<Sentence, Representation>
Training Examples
Parsing Operator Generator
Prolog
Overly−General Parser
Example Analysis
Control
Examples
Control Rule Induction
Control
Rules
Program Specialization
Final Parser
Prolog

Training Examples

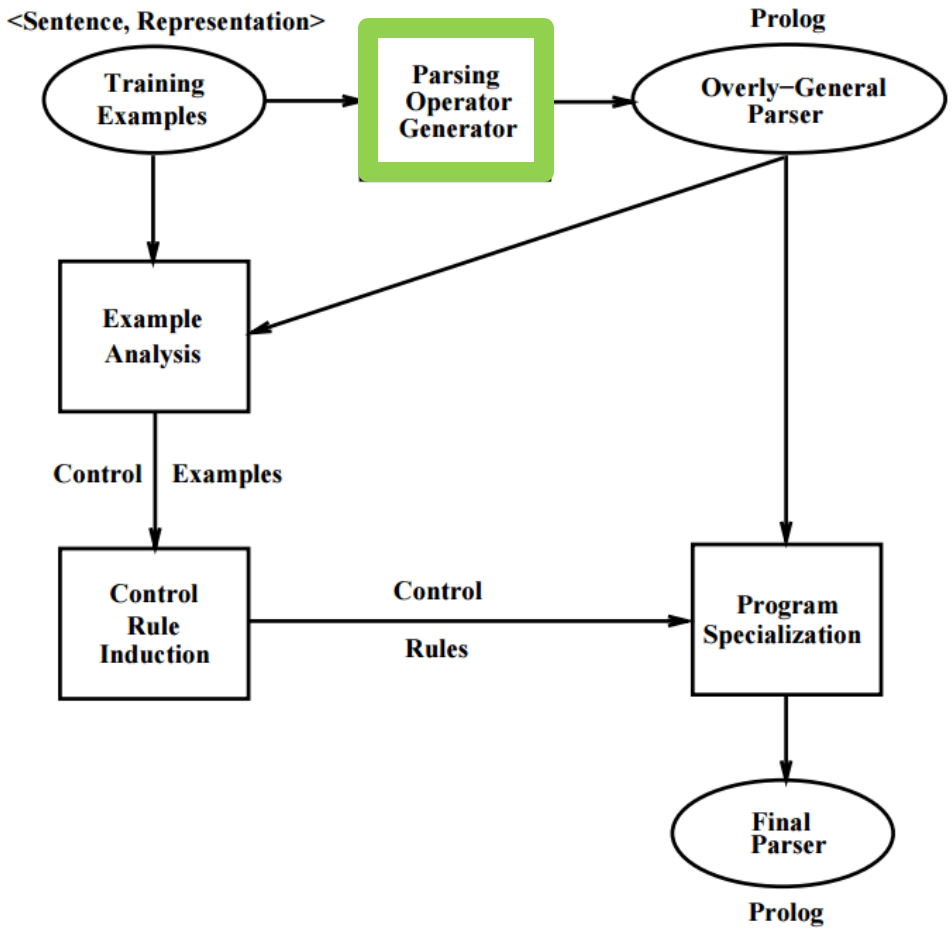
- <Sentence, Parse>:
  - For exmaple:

    <The man ate the pasta,

    [[ate, **obj:**[pasta, **det:**the], **agt:**[man, **det:**the]]] >



CHILL


<Sentence, Representation>
Training Examples
Parsing Operator Generator
Prolog
Overly−General Parser
Example Analysis
Control
Examples
Control Rule Induction
Control
Rules
Program Specialization
Final Parser
Prolog

Parsing Operator Generation

- The training examples are analyzed to formulate an overly-general shift-reduce parser that is capable of producing parses from sentences

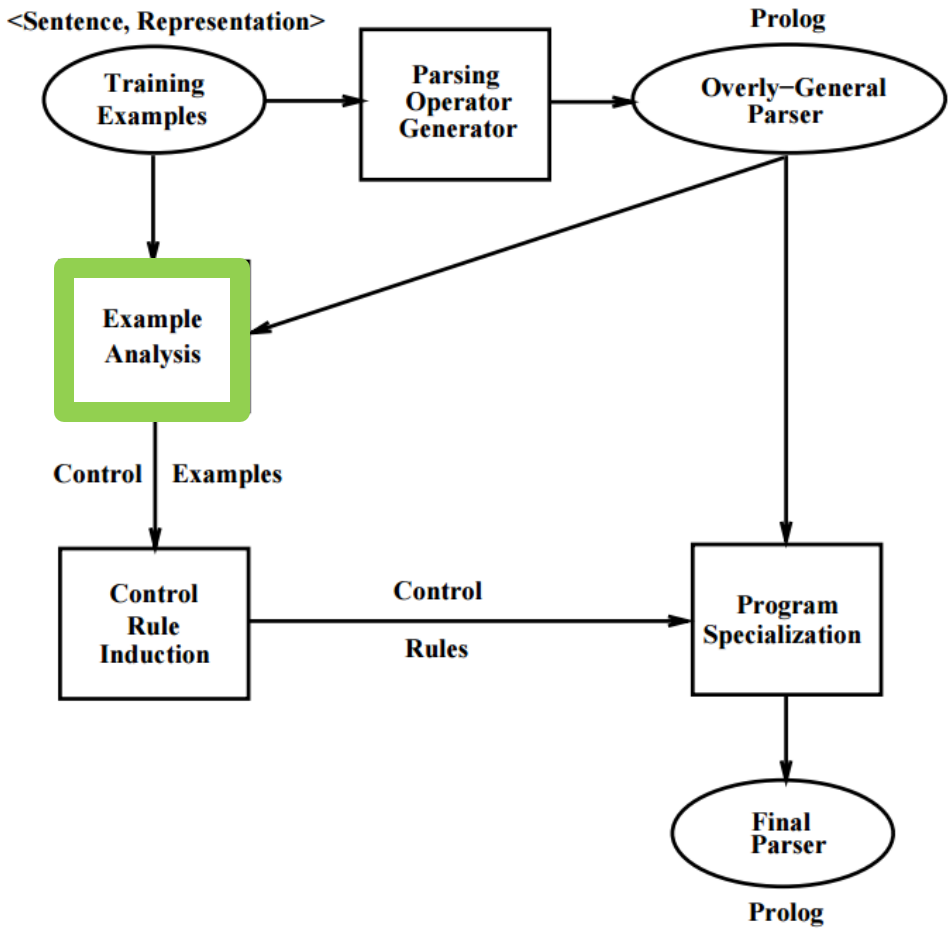
```
parse(S, Parse) :- parse([], S, [Parse], []).

parse(Stack, Input, Stack, Input).
parse(Stack0, In0, Stack, In) :-
        op(Stack0, In0, Stack1, In1), parse(Stack1, In1, Stack, In).

op([Top,Second|Rest],In,[NewTop|Rest],In) :-
        reduce(Top,agt,Second,NewTop).
op([Top,Second|Rest],In,[NewTop|Rest],In) :-
        reduce(Top,det,Second,NewTop).
op([Top,Second|Rest],In,[NewTop|Rest],In) :-
        reduce(Second,obj,Top,NewTop).
op(Stack,[Word|Words],[Word|Stack],Words).  % Shift operation.

reduce([Head|Slots],Role,Filler,[Head,Role:Filler|Slots]) :- !.
reduce(BareHead,Role,Filler,Result) :-
             reduce([BareHead],Role,Filler,Result).
```

CHILL


<Sentence, Representation>
Training Examples
Parsing Operator Generator
Prolog
Overly−General Parser
Example Analysis
Control
Examples
Control Rule Induction
Control
Rules
Program Specialization
Final Parser
Prolog

Example Analysis

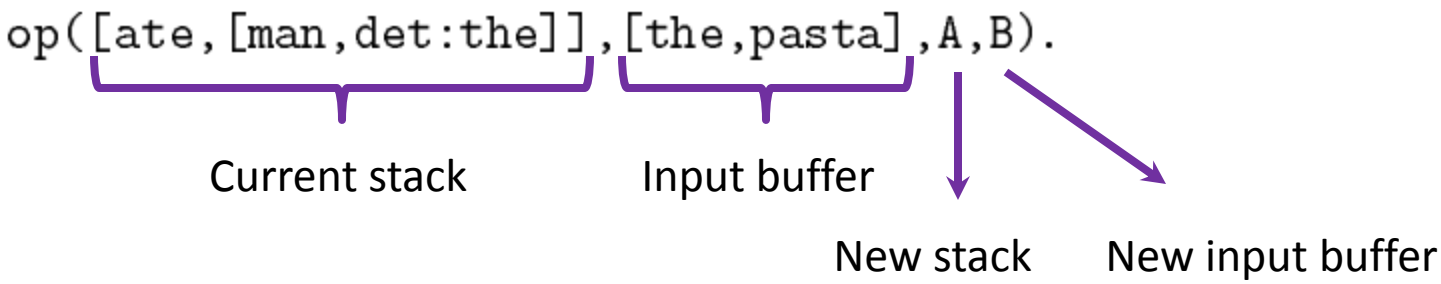
- For Each operator we want to generate two kind of examples:
  - Correct control examples: in what situation we **should** apply this operator
  - Incorrect control examples: in what situation we **shouldn't** apply this operator .

Example Analysis

<u>Positive control examples</u> :

for each example of the training instance, parse states to which the operator should be applied.

for example: the operation that reduced to *agt:*



Example Analysis

Negative control examples :

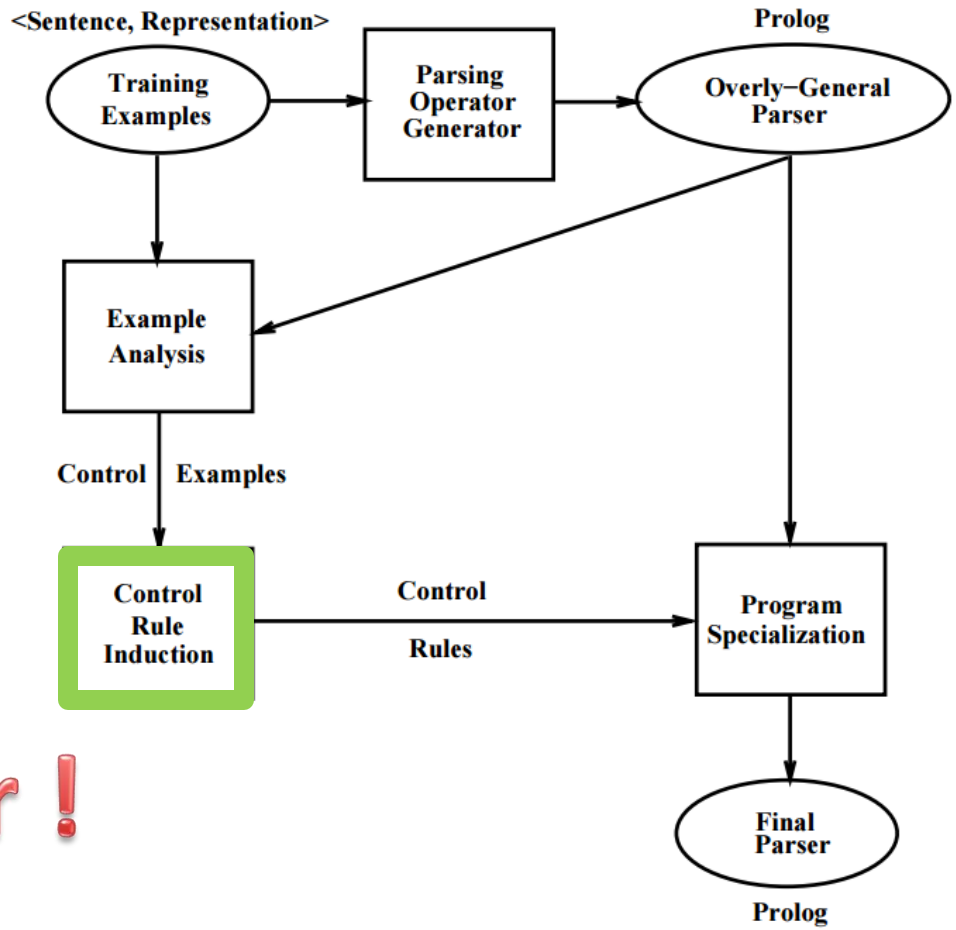
all contexts where this operator was not applied.

! We assume that the set of training examples includes a pair for every correct parsing for each unique sentence appearing in the set.

For example (*agt* reduction operator):

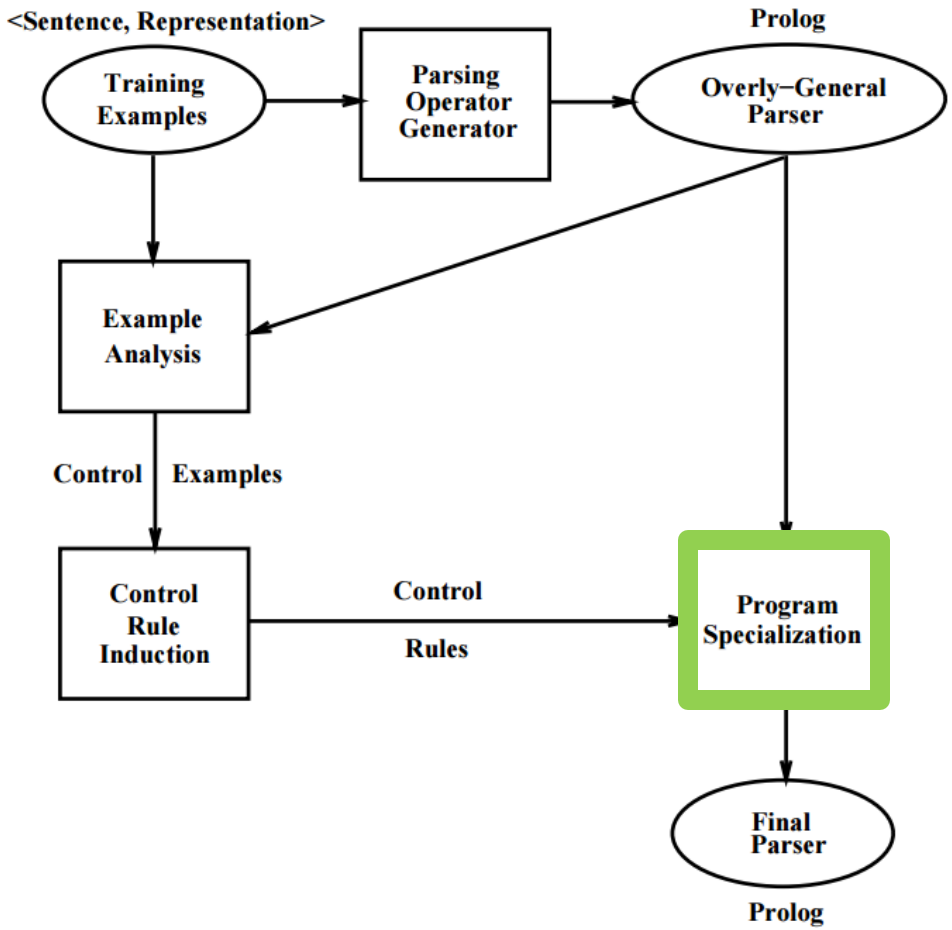
```
op([man,the],[ate,the,pasta], A, B)
op([the,[ate,agt:[man,det:the]]], [pasta], A,B)
```

CHILL


<Sentence, Representation>
Training Examples
Parsing Operator Generator
Prolog
Overly−General Parser
Example Analysis
Control
Examples
Control Rule Induction
Control
Rules
Program Specialization
Final Parser
Prolog

Later !

CHILL


<Sentence, Representation>
Training Examples
Parsing Operator Generator
Prolog
Overly−General Parser
Example Analysis
Control
Examples
Control Rule Induction
Control
Rules
Program Specialization
Final Parser
Prolog

Program Specialization

- “fold” the control information back into the overly-general parser. Each operator clause in the overly-general parser is modified by adding the learned control knowledge so that attempts to use the operator inappropriately fail immediately.

CHILL for GeoQuery

- `introduce` action:
  - The word *capital* might cause the *capital/2* predicate to be pushed on the stack
- `co-reference` action:
  - variables may be unified with variables appearing in other stack items.
  - For example, the first argument of the *capital/2* structure may be unified with the argument of a previously introduced *state/1* predicate
- `conjoin` action:
  - a stack item may be embedded into the argument of another stack item to form conjunctive goals inside of meta-predicates

CHILL for GeoQuery

- Parsing example for the query:

  What is the capital of Texas?

Assume:

- A lexicon that maps:

  ‘capital’ to capital(_)

  ‘of’ to loc(_)

  ‘Texas’ to const(_, stateid(texas))
- Each predicate on the parse stack has an attached buffer to hold the context in which it was introduced

CHILL for GeoQuery

- What is the capital of Texas?

Parse Stack	Input Buffer	Action
[answer(_,_):[]]	[*what,is,the,capital,of,texas,?*]	3* SHIFT
[answer(_,_):[*the,is,what*]]	[*capital,of,texas,?*]	INTRODUCE
[capital(_):[], answer(_,_) :[*the,is,what*]]	[*capital,of,texas,?*]	COREF
[capital(C):[], answer(C,_) :[*the,is,what*]]	[*capital,of,texas,?*]	SHIFT
[capital(C):[*capital*], answer(C,):[*the,is,what*]]	[*of,texas,?*]	INTRODUCE
[loc(,):[], capital(C):[*capital*], answer(C,_):[*the,is,what*]]	[*of,texas,?*]	COREF
[loc(C,):[], capital(C):[*capital*], answer(C,_):[*the,is,what*]]	[*of,texas,?*]	SHIFT

CHILL for GeoQuery

- What is the capital of Texas?

Parse Stack	Input Buffer	Action
…	…	…
[const(S,stateid(texas)):[], loc(C,S):[*of*], capital(C):[*capital*], answer(C,_):[*the*,*is*,*what*]]	[*texas*, *?*]	CONJ
[loc(C,S):[*of*], capital(C):[*capital*], answer(C, const(S,stateid(texas))) :[*the*,*is*,*what*]]	[*texas*, *?*]	CONJ
[capital(C):[*capital*], answer(C, (loc(C,S),const(S,stateid(texas)))) :[*the*,*is*,*what*]]	[*texas*, *?*]	CONJ
[answer(C, (capital(C),loc(C,S),const(S,stateid(texas)))) :[*the*,*is*,*what*]]		

Plan

- Task definition
- General background:
  - Prolog
  - Shift Reduce Parsing
- CHILL parser
- ILP (Inductive logic programming)
  - CHILLIN algorithm
- Experiments and Results

First order logic

Consist of:

1. The quantifier symbols $\forall$ and $\exists$
2. $\wedge$ for conjunction, $\vee$ for disjunction, $\rightarrow$ for implication, $\leftrightarrow$ for biconditional, $\neg$ for negation
3. Variables
4. An equality symbol (sometimes, identity symbol) $=$

Inductive logic programming

Positive examples +
Negative examples +
Background knowledge

$\Rightarrow$

hypothesis that entail <u>all</u> positive examples and <u>none</u> of the negatives

ILP – formal definitions

- Given
  - a logic program B representing background knowledge
  - a set of positive examples E^+
  - a set of negative examples E^-
- Find hypothesis H such that:

1. $B \cup H \models$ for every $e \in E^+$. (complete)
2. $B \cup H \not\models$ for every $f \in E^-$.
3. $B \cup H$ is consistent.

Assume that $B \not\models e$ for some $e \in E^+$.

ILP – List membership example

- For Example: consider learning the concept of list membership.
  - Positive examples:

    `member(1, [1,2]),`

    `member(2, [1,2]),`

    `member(1,[3,1]),` etc
  - Negative examples:

    `member(1,[]),`

    `member(2,[1,3]),` etc
  - Background:

    The predicate *components/3* which decomposes a list into its component head and tail.

ILP – List membership example

- Given the examples + background, we hope to learn the correct definition of member, namely:

```
member(X, List) :- components(List, X, Tail).
member(X, List) :- components(List, Head, Tail), member(X, Tail).
```

CHILLIN

CHILLIN (CHILL inductive algorithm)

<u>Input:</u>

- A set of positive and negative examples of a concept, expressed as a ground facts
- A set of background predicates, expresses as definite clauses

<u>Output:</u>

definite clause concept definition which entails the positives examples, but not the negative.

CHILLIN

1. The algorithm starts with a most specific definition (the set of all positive examples)
2. Then, it introduces a generalization which make the definition more compact
   - Compactness: measured by the sum of the program's clauses

CHILLIN

DEF := {E :- true | E ∈ Pos}
Repeat
  PAIRS := a sampling of pairs of clauses from DEF
  GENS := {G | G = build_gen(C_i,C_j,DEF,Pos,Neg)
    for $\langle C_i, C_j \rangle \in$ PAIRS}
  G := Clause in GENS yielding most compaction
  DEF := (DEF−(Clauses subsumed by G)) ∪ G
Until no further compaction

1. We start with a most specific definition: a set of all the positive examples

CHILLIN

DEF := {E :- true | E ∈ Pos}
Repeat
  PAIRS := a sampling of pairs of clauses from DEF
  GENS := {G | G = build_gen(C_i,C_j,DEF,Pos,Neg)
    for $\langle C_i, C_j \rangle \in$ PAIRS}
  G := Clause in GENS yielding most compaction
  DEF := (DEF−(Clauses subsumed by G)) ∪ G
Until no further compaction

2. A search for more general definition is carried out

CHILLIN

DEF := {E :- true | E ∈ Pos}
Repeat
  PAIRS := a sampling of pairs of clauses from DEF
  GENS := {G | G = build_gen(C_i,C_j,DEF,Pos,Neg)
    for $\langle C_i, C_j \rangle \in$ PAIRS}
  G := Clause in GENS yielding most compaction
  DEF := (DEF−(Clauses subsumed by G)) ∪ G
Until no further compaction

3. We sample 15 pairs of clauses from DEF

CHILLIN

DEF := {E :- true | E ∈ Pos}
Repeat
  PAIRS := a sampling of pairs of clauses from DEF
  GENS := {G | G = build_gen(C_i,C_j,DEF,Pos,Neg)
    for $\langle C_i, C_j \rangle \in$ PAIRS}
  G := Clause in GENS yielding most compaction
  DEF := (DEF−(Clauses subsumed by G)) ∪ G
Until no further compaction

4. For each pair we build a generalization

details in a few slides!

CHILLIN

DEF := {E :- true | E ∈ Pos}
Repeat
  PAIRS := a sampling of pairs of clauses from DEF
  GENS := {G | G = build_gen(C_i,C_j,DEF,Pos,Neg)
    for $\langle C_i, C_j \rangle \in$ PAIRS}
  G := Clause in GENS yielding most compaction
  DEF := (DEF−(Clauses subsumed by G)) ∪ G
Until no further compaction

5. Choose the best generalization (using the compaction measure)

CHILLIN

DEF := {E :- true | E ∈ Pos}
Repeat
  PAIRS := a sampling of pairs of clauses from DEF
  GENS := {G | G = build_gen(C_i,C_j,DEF,Pos,Neg)
    for $\langle C_i, C_j \rangle \in$ PAIRS}
  G := Clause in GENS yielding most compaction
  DEF := (DEF−(Clauses subsumed by G)) ∪ G
Until no further compaction

5. The reduction:
   - add G to DEF
   - Prove all positive examples with DEF, remove all clauses that were not used in any of the proofs.

CHILLIN - Generalizations

- The generalization process consists of three steps:
  1. Introduce a simple generalization of the input clauses
  2. If this generalization covers no negative examples, it is returned
  3. Else (the generalization is too general) try:
     a. Adding literals to the generalization
     b. Call a routine which invents a new predicate, so that no negative examples are covered.

CHILLIN - Generalizations

- For example:

  given the positive clauses:

  - `member(1,[1,2,3])`
  - `member(3, [3])`

  The least general generalization would be:

  `member(A, [A|B])`

  Which is a valid generalization (no need for stage 3)

Plan

- Task definition
- General background:
  - Prolog
  - Shift Reduce Parsing
- CHILL parser
- ILP (Inductive logic programming)
  - CHILLIN algorithm
- Experiments and Results

Experiment setting

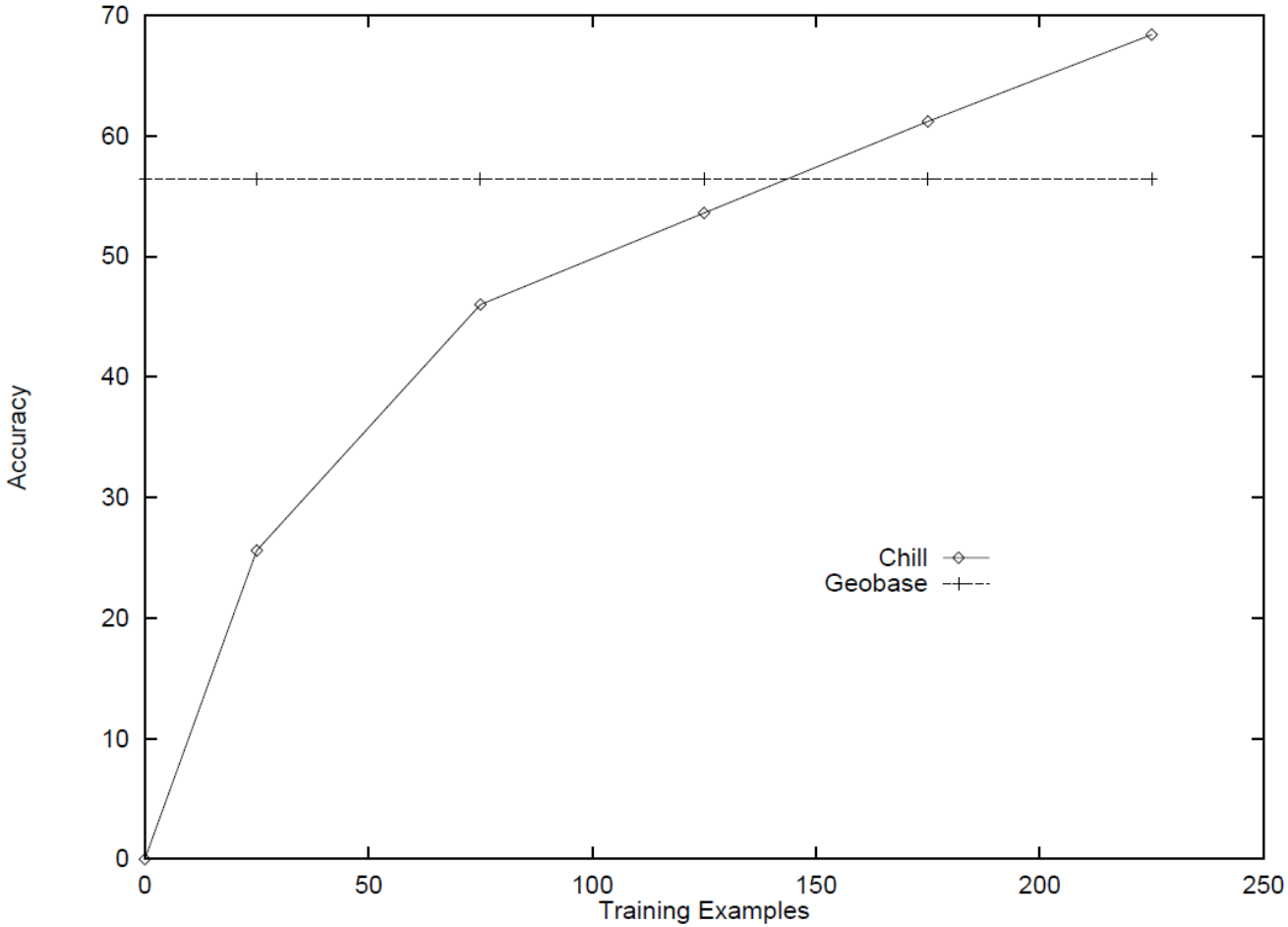
- A corpus of 250 sentences was gathered by submitting a questionnaire to 50 uninformed subjects.
- For evaluation purposes, the corpus was split into:
  - training sets of 225 examples
  - with the remaining 25 held-out for testing.
- Overall 10 folds

Experiment setting

<u>Baseline:</u>

- Geobase, uses a semantic-base parser which scans for words corresponding to the entities and relationships encoded in the database
- The system attempts to match sequences of entities and associations in sentences with an entity-association network describing the schemas present in the database

Expertiment result



Questions?

References

- Using Inductive Logic Programming to Automate the Construction of Natural Language Parsers, John M. Zelle, PHD Thesis

- Learning to Parse Database Queries using Inductive Logic Programming, John M. Zelle and Raymond J. Mooney

- Learning Semantic Grammars with Constructive Inductive Logic Programming, John M. Zelle and Raymond J. Mooney

- Using Multiple Clause Constructors in Inductive Logic Programming for Semantic Parsing, Lappoon R. Tang and Raymond J. Mooney

References

- https://en.wikipedia.org/wiki/Inductive_logic_programming
- https://en.wikipedia.org/wiki/Prolog
- lyle.smu.edu/~mhd/2353sp09/prolog.pptx
- https://www.cs.northwestern.edu/academics/courses/322/notes/05.ppt