Lecture 10 – Synthesis from Examples

PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav

Previously

- Abstraction-Guided Synthesis
  - changing the program to match an abstraction
  - transformations for sequential programs (equivalence)
  - transformations for concurrent programs
    - adding synchronization
    - program restriction

Today

- Synthesis from examples
- SMARTEdit
- String processing in spreadsheet from examples

- Acks
  - Some slide cannibalized from Tessa Lau
  - String processing in spreadsheets slides cannibalized from Sumit Gulwani

Programming by demonstration

- Learn a program from examples
- Main challenge: generalization
  - generalize from examples to something that is applicable in new situations
  - how can you generalize from a small number of examples?
  - how do you know that you're done (generalized "enough")?

Demo

SMARTedit

Programming by demonstration

- Can be viewed as a search in the space of programs that are consistent with the given examples

- how to construct the space of possible programs?
- how to search this space efficiently?

Program synthesis with inter-disciplinary inspiration

- Programming Languages
  - Design of an expressive language that can succinctly represent programs of interest and is amenable to learning
- Machine Learning
  - Version space algebra for learning straight-line code
  - other techniques for conditions/loops
- HCI
  - Input-output based interaction model

Version Spaces

- Hypothesis space
  - set of functions from input domain to output domain
- Version space
  - subspace of hypothesis space that are consistent with examples
  - partially ordered
  - generality ordering: $h_1 \sqsubseteq h_2$ iff h_2 covers more examples than h_1
  - (can also use other partial orders)

Version Spaces

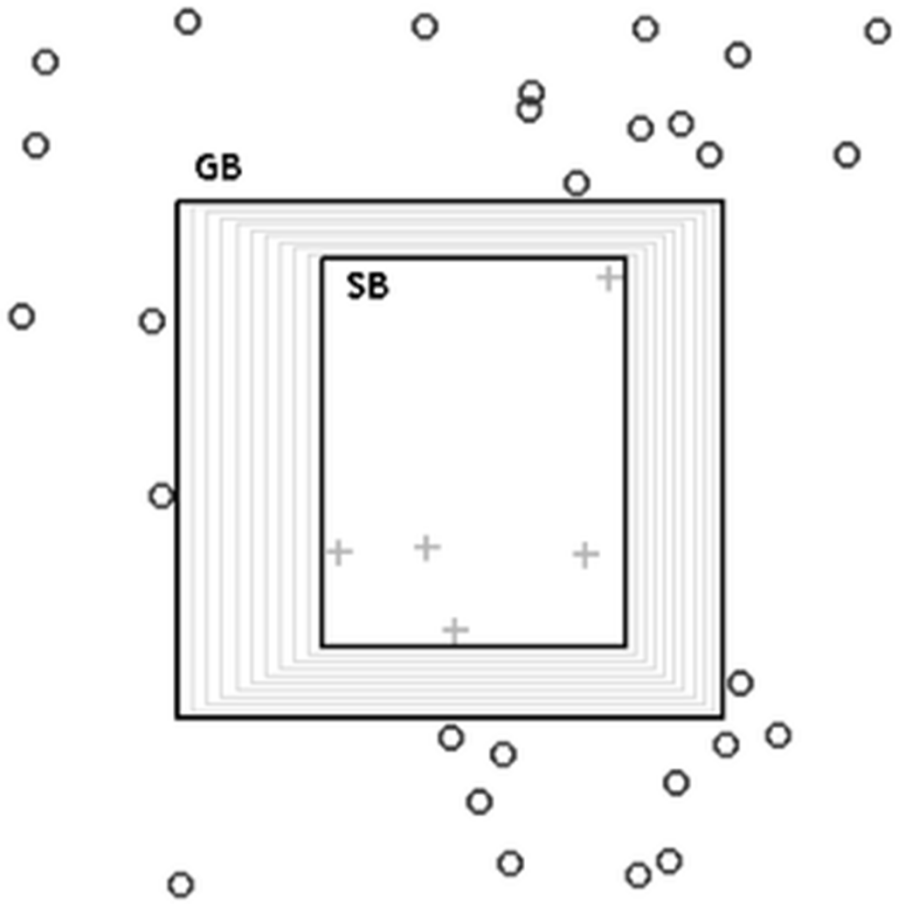
- A hypothesis h is consistent with a set of examples D iff
  h(x) = y for each example <x,y> $\in$ D

- The version space $VS_{H,D}$ w.r.t. hypothesis space H and examples D, is the subset of hypotheses from H consistent with all examples in D

Version Spaces

- when using generality ordering version space can be represented using just
  - the most general consistent hypotheses (least upper bound)
  - the most specific consistent hypotheses (greatest lower bound)

Version Spaces

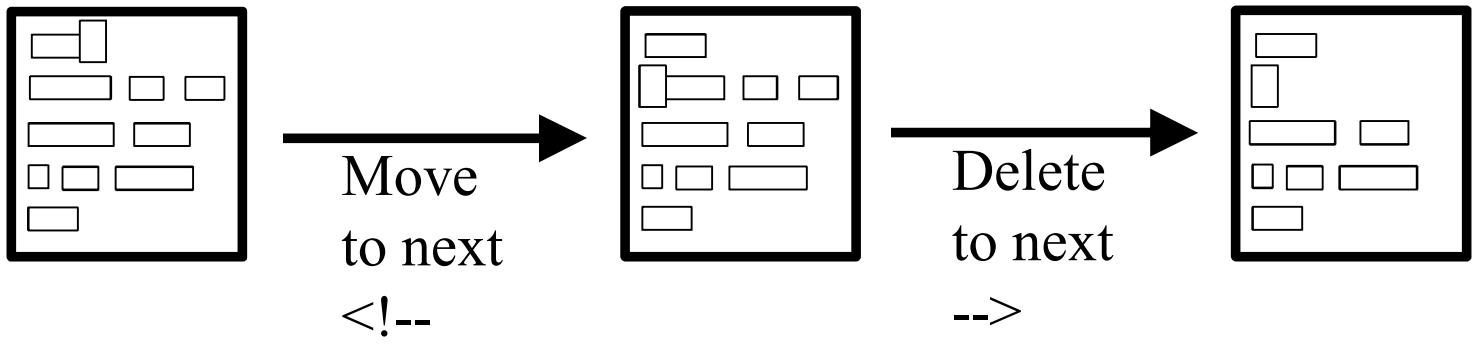


Version Space Algebra

- Union
  - $VS_{H1,D} \cup VS_{H2,D} = VS_{H1 \cup H2, D}$
- Join (what we would call reduced product)
  - D1 = sequence of examples over H1
  - D2 = sequence of examples over H2
  - $VS_{H1,D1} \bowtie VS_{H2,D2} =$
    $\{\langle h1,h2\rangle \mid h1 \in VS_{H1,D1}, h2 \in VS_{H2,D2,} C(\langle h1,h2\rangle,D)\}$
  - C(h,D) – consistency predicate, true when h consistent in D
- Independent join (product, no reduction)
  - $\forall D1,D2, h1 \in H1, h2 \in H2.$
    $C(h1,D1) \wedge C(h2,D2) \Rightarrow C(\langle h1,h2\rangle,D)$

How SMARTedit works

- Action is function : input state → output state
  - Editor state: text buffer, cursor position, etc.
  - Actions: move, select, delete, insert, cut, copy, paste



- Given a state sequence, infer actions
  - Many actions may be consistent with one example

What action?

```
PROBLEM #1
----------

This is some sample<!-- deleteme --> HTML
text from which the comments <!--
including contents -->ought to be
deleted before <!--ZZZ-->publication.

This comment deletion task is one example of the
types of repetitive tasks for which SMARTedit saves
user effort.
```

what is the source location?
- first location in text?
- any location?
- …

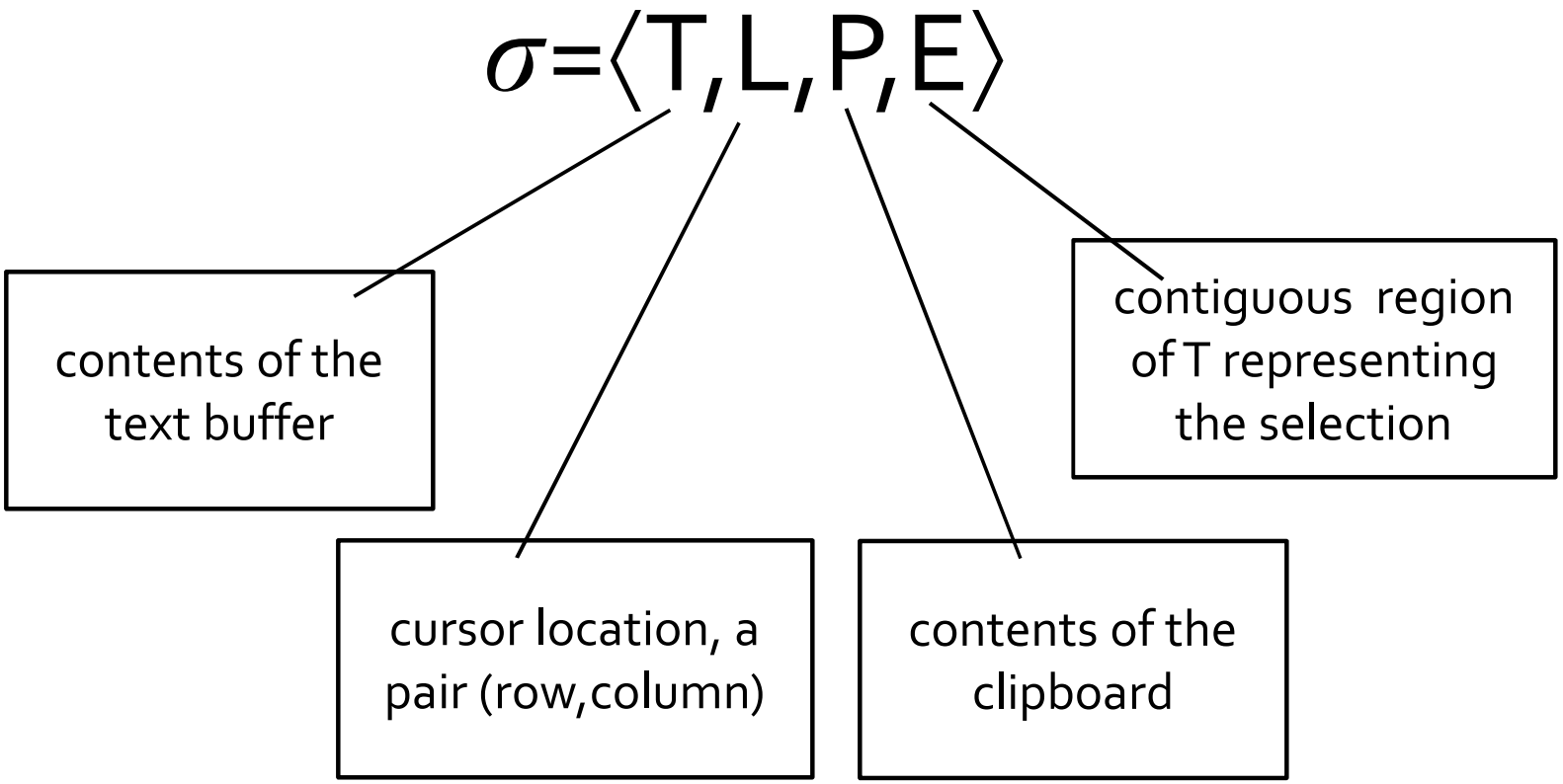
move?

```
PROBLEM #1
----------

This is some sample<!-- deleteme --> HTML
text from which the comments <!--
including contents -->ought to be
deleted before <!--ZZZ-->publication.

This comment deletion task is one example of the
types of repetitive tasks for which SMARTedit saves
user effort.
```

what is the target location?
- after "ple"?
- after "sample"?
- before "<!—"?
- (4,19) ?
- …

learned function has to be applicable in other settings

Editor State



Editor Transition (Action)

- Editor state $\sigma = \langle T, L, P, E \rangle$ out of set of possible editor states Σ

- Editor action is a function a: $\Sigma \rightarrow \Sigma$

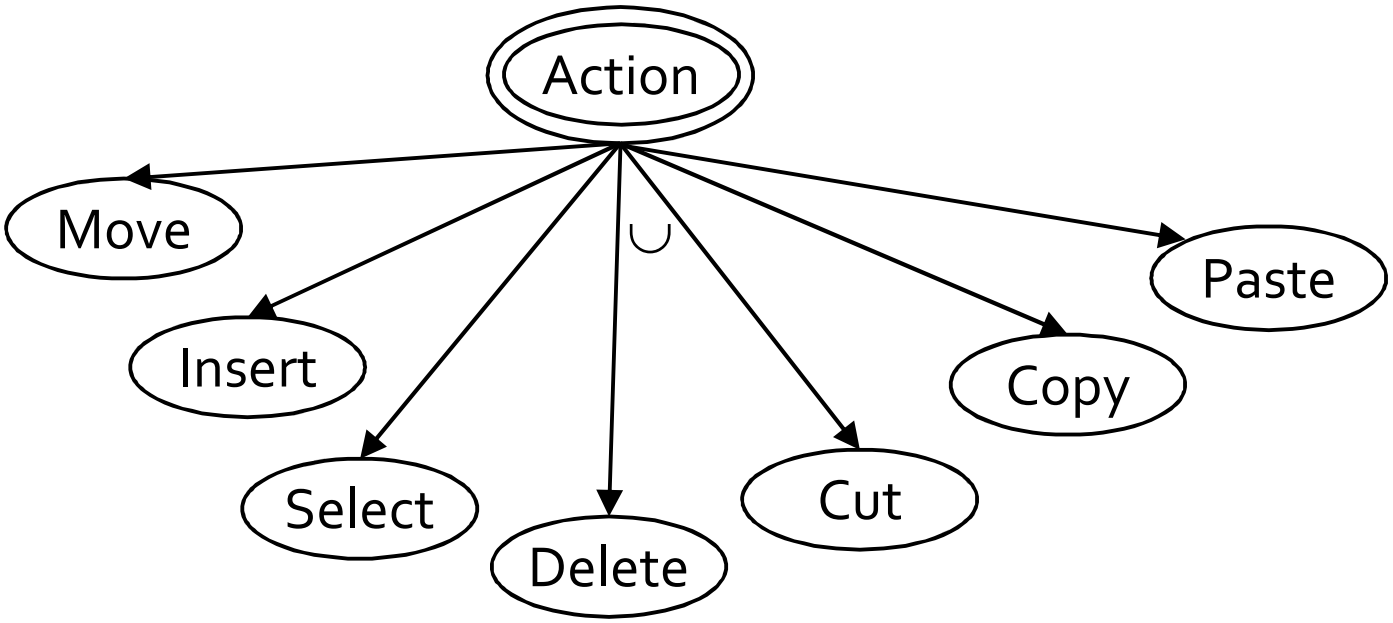
Editor Transition (Action)

$$\langle T,(42,0),P,E\rangle \Longrightarrow \langle T,(43,0),P,E\rangle$$

consistent
- "move to the next line"
- "move to the beginning of line 43"

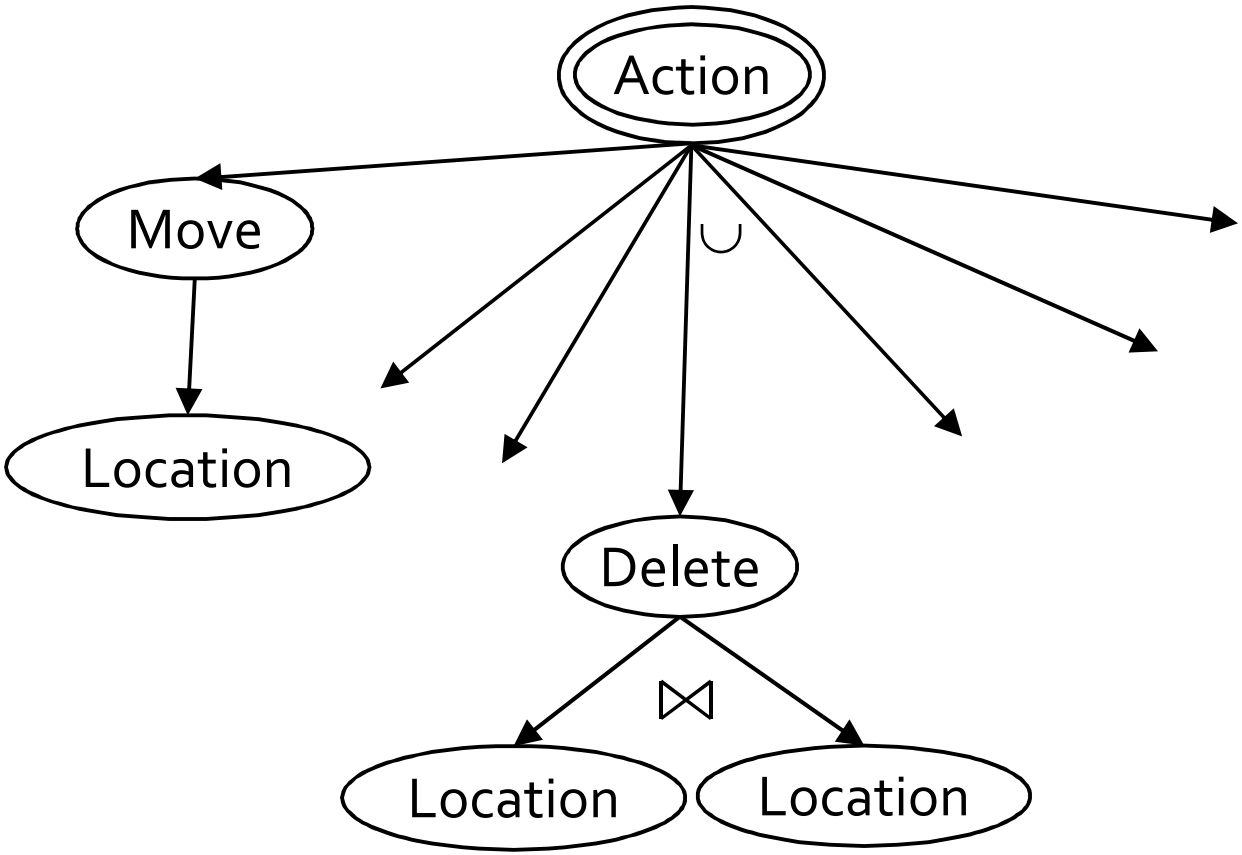
inconsistent
- "move to the beginning of line 47"
- "move to the end of line 41"

SMARTedit's version space



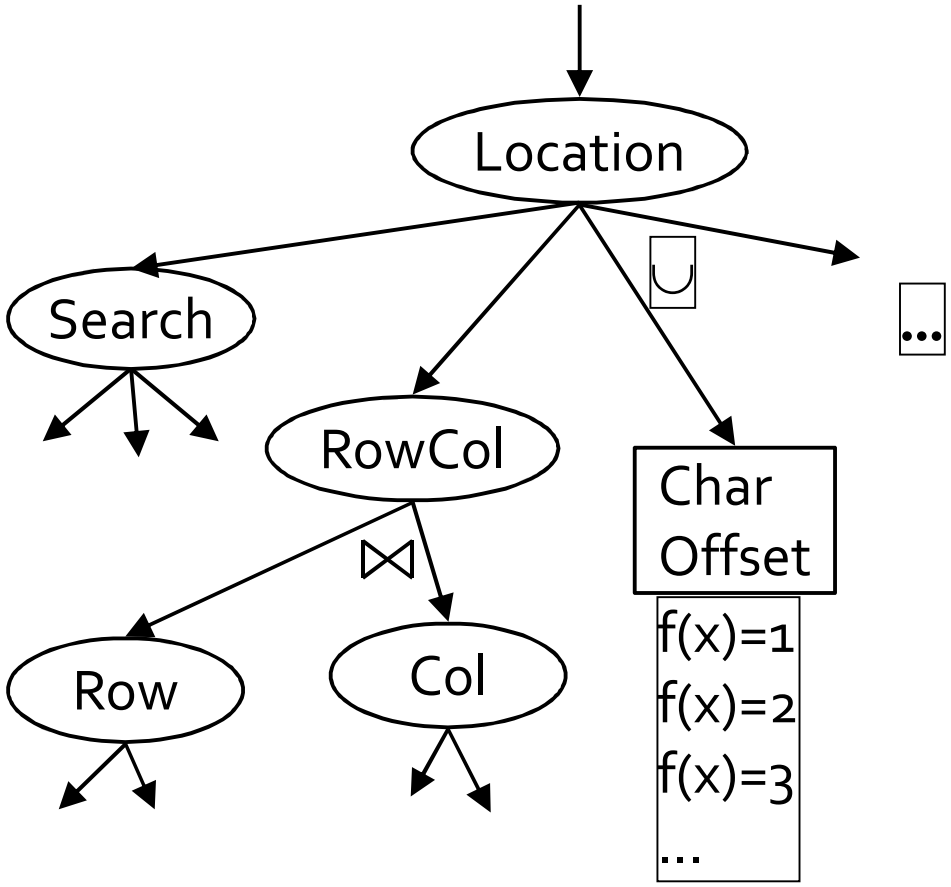
- Action function maps from one state to another
- Action version-space is a union of different kinds of actions

SMARTedit's version space



Express action functions in terms of locations

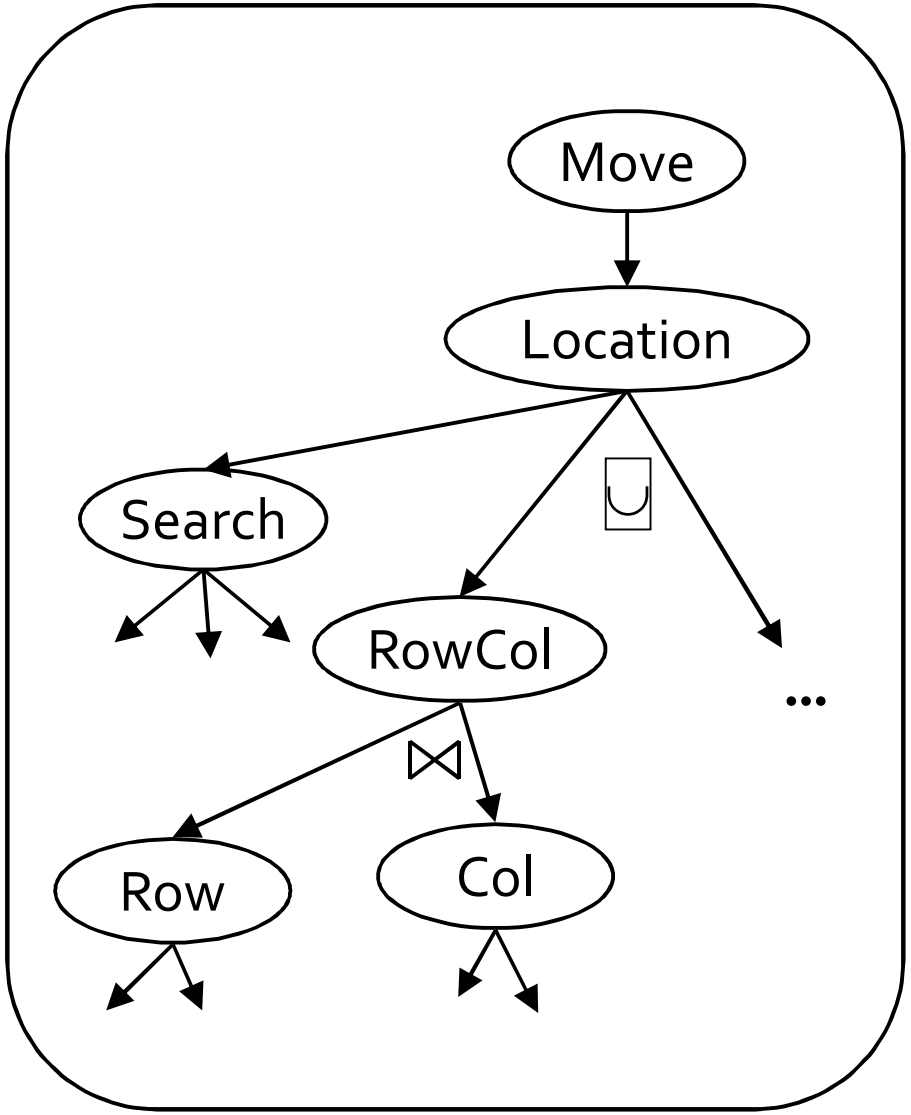
Location version space



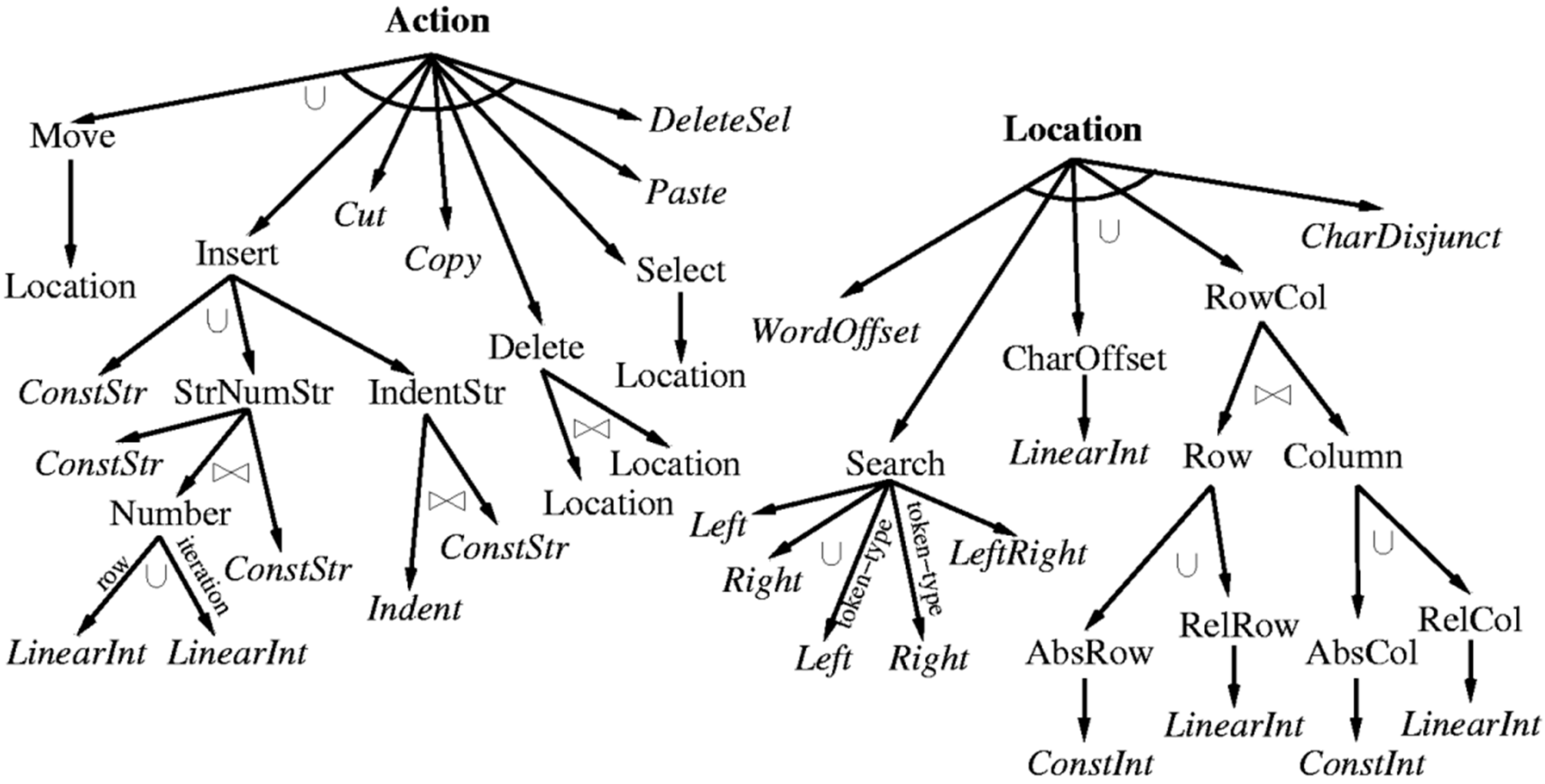
- Rectangle indicates atomic (leaf) version space
- Location functions map from text state (buf, pos) to position

Move Actions

- functions that change the location in the text
  - explicit target location in terms of row,column
  - relative location based on search
  - …



SMARTedit's version space



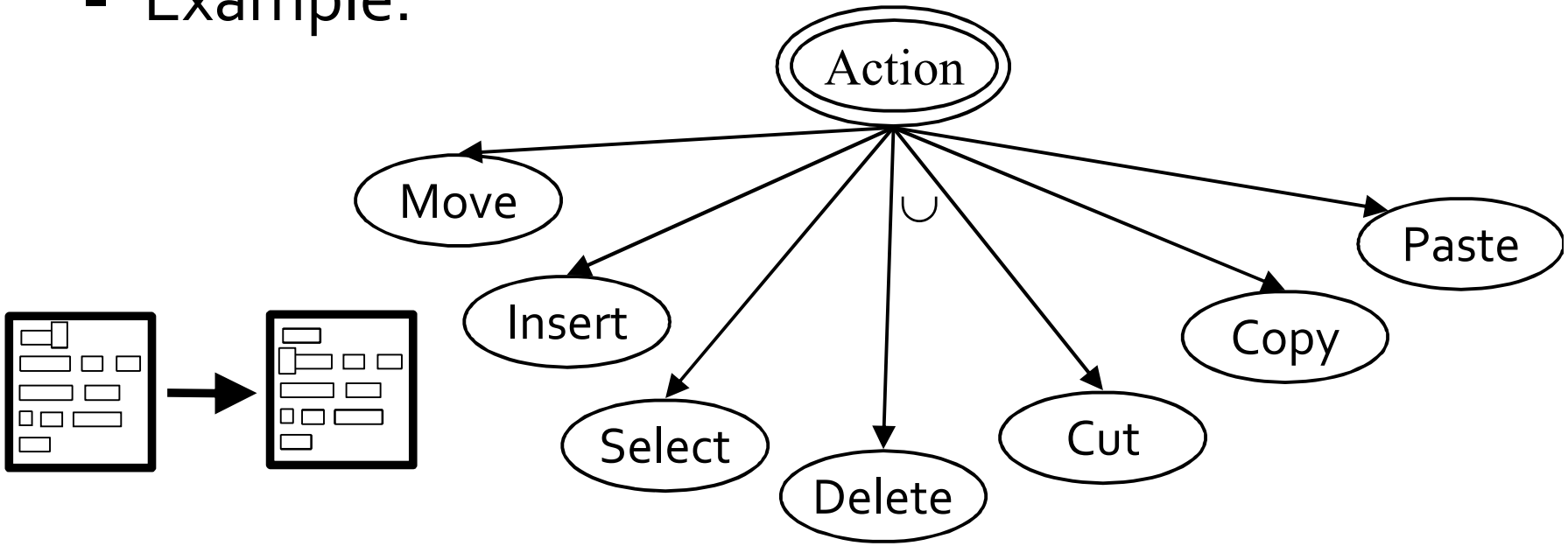
How does the system learn?

- Update version space on new example
  - Remove inconsistent hypotheses
  - Prune away parts of the hierarchy

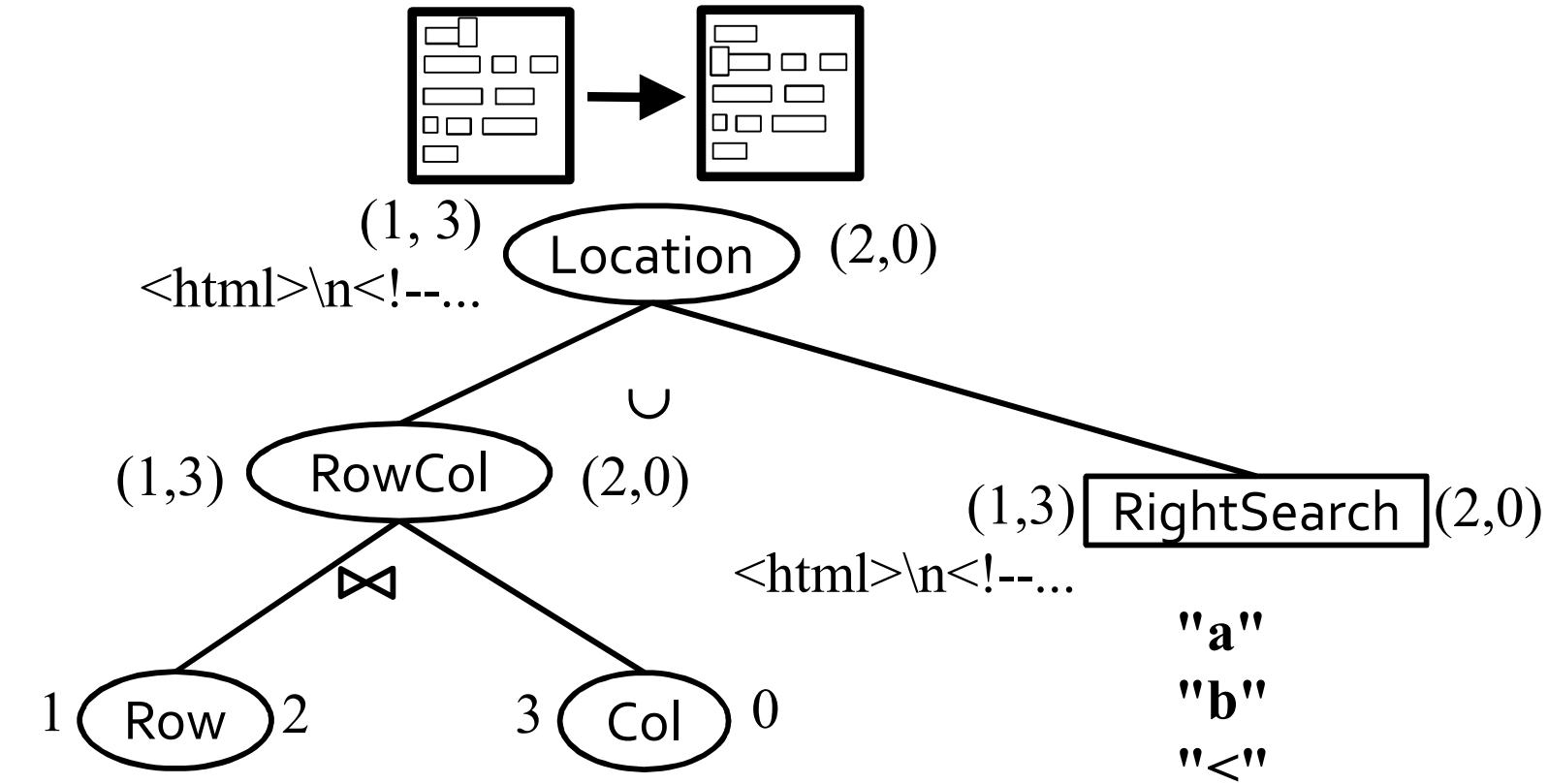
- Execute version space for prediction
  - Give system current state
  - What state would the user produce next?

Updating the version space

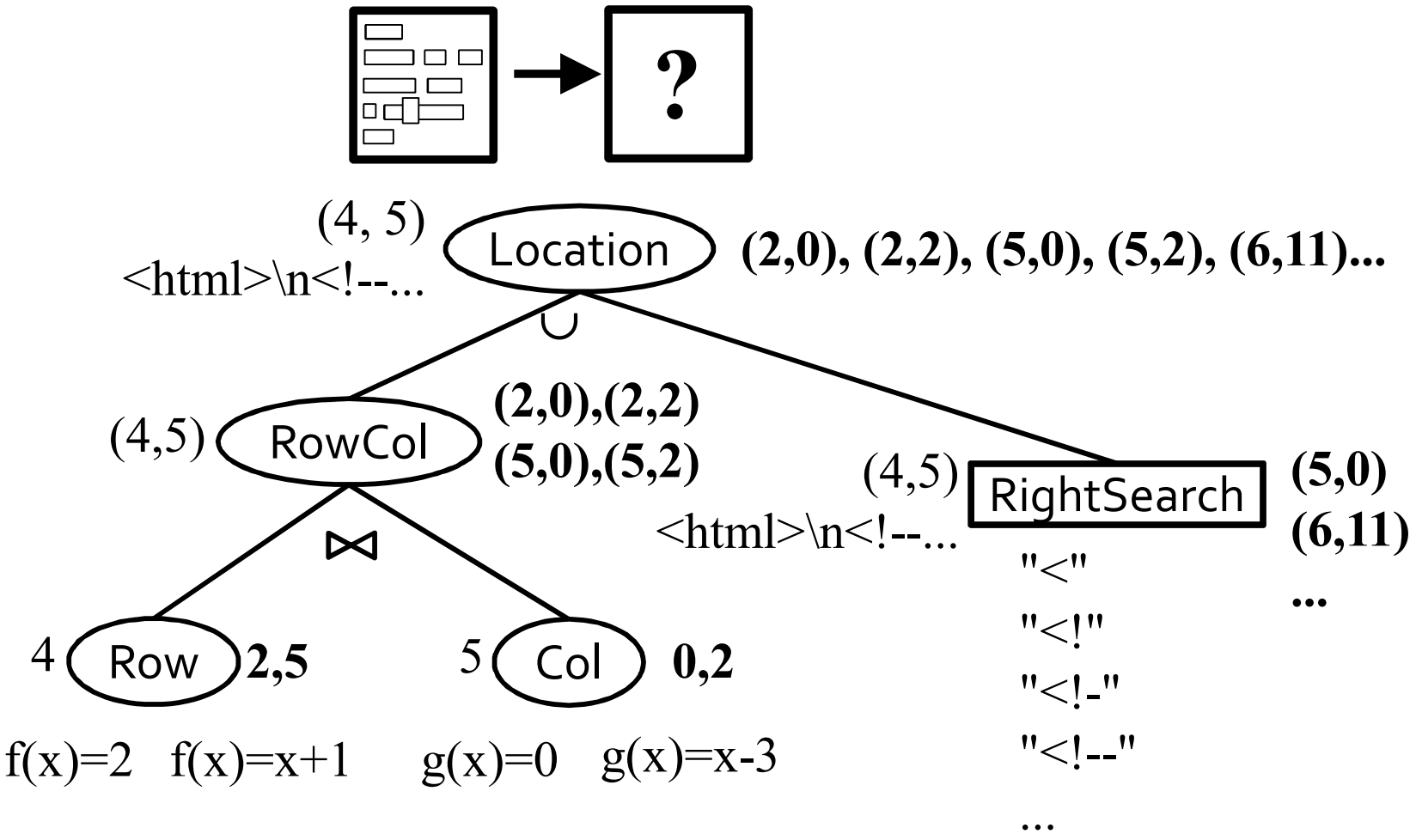
- Test consistency of example against entire version space
- Quickly prune subtrees
- Example:



Updating the version space



Executing the version space



Choosing between multiple outputs?

- How to choose between possible outputs?
- Associate probability with each hypothesis
  - Make better predictions
  - Introduce domain knowledge

- Introduce probabilities at two points in hierarchy
  - Probability distribution over hypotheses at leaf nodes
  - Weights for each VS in a union

How does it really work?

- what version spaces look like?
- how do you represent them efficiently?
- how do you update a version space?
- how do you execute a version space?

- dive deeper into *string searching* version spaces

String Searching

- need to express locations relative to a string or a pattern
  - e.g., move the cursor to the next <!--
- Let string $X = x_1 x_2 \ldots x_i \bullet x_{i+1} \ldots x_n$ be a string over some alphabet A
  - the dot • denotes position in the string
  - X.left = substring before the dot
  - X.right = substring after the dot

Right-search Hypotheses

- right-search hypotheses output the next position such that a particular string is to its right

- For each sequence of tokens S, the right-hypothesis of S, hright_S is a hypothesis that given an input state $\langle T,L,P,E\rangle$ outputs the first position Q > L such that S is a prefix of T.right(Q)

Example: Right-search Hypotheses

- the user moves cursor the beginning of text occurrence "Candy"
- 5 right-hypotheses consistent with this action are:
  - $\text{hright}_{\text{Candy}}$
  - $\text{hright}_{\text{Cand}}$
  - $\text{hright}_{\text{Can}}$
  - $\text{hright}_{\text{Ca}}$
  - hright_{C}

- how do you represent the right-search version space?

Representing right-search version space

- define the partial order $\prec^{\text{right}}$ to be the string prefix relation

- $\text{hright}_{s1} \prec^{\text{right}} \text{hright}_{s2}$
  iff s1 is a proper prefix of s2

- $\text{hright}_{\text{Candy}}$ is the most general hypothesis for the previous example

Updating right-search version space

- LUB S initialized to a token representing all strings of length K (greater than buffer size)
- GLB C initialized to a token representing all strings of unit length
- Given an example d = $\langle T,L,P,E\rangle \longrightarrow \langle T,L',P,E\rangle$
  - cursor moved from position L to L'
  - T.right(L') is the longest possible string the user could have been was searching
  - In moving from L to L', user may have skipped over a prefix of T.right(L') --- another occurrence --- such prefix is not the target hypothesis. Denote by S_N the longest prefix of T.right(L') that begins in the range [L,L')

Updating right-search version space

- Given an example d = ⟨T,L,P,E⟩ ⟶ ⟨T,L′,P,E⟩
  - T.right(L′) is the longest possible string the user could have been was searching
  - S_N the longest prefix of T.right(L′) that begins in the range [L,L′)
- LUB = longest common prefix of LUB and T.right(L′)
- GLB = longer string of GLB or S_N

- if GLB is equal to or prefixed by LUB, version space collapses into the null set.

Example

speak •spaceship

- LUB = “spaceship”
- GLB = “sp”

- version space contains all prefixes of string in the LUB expect for the hypothesis “s” and “sp”

Executing right-search version space

- the version space is equivalent to a set of strings
  - longest one is in the LUB
  - others are some prefixes of the LUB
- execution applies each hypothesis to the input state and computes set of outputs

- we don't want to explicitly enumerate all hypotheses (substrings) in the space
  - leverage relationship between hypotheses

Executing right-search version space

- executing single hypothesis
  - search for the next occurrence of a string relative to starting position L
- for each hypothesis
  - find the next occurrence of the associated string in the text
  - output the location and the probability of the hypothesis
- match longest string against every position of the text, look for partial matches
  - can probably exploit KMP string matching algorithm

Generalizing String Searching

- can represent a string search version space as two offsets in a sequence of tokens

positive { dependent on
dependently

longest common prefix = "dependent"

negative { decline

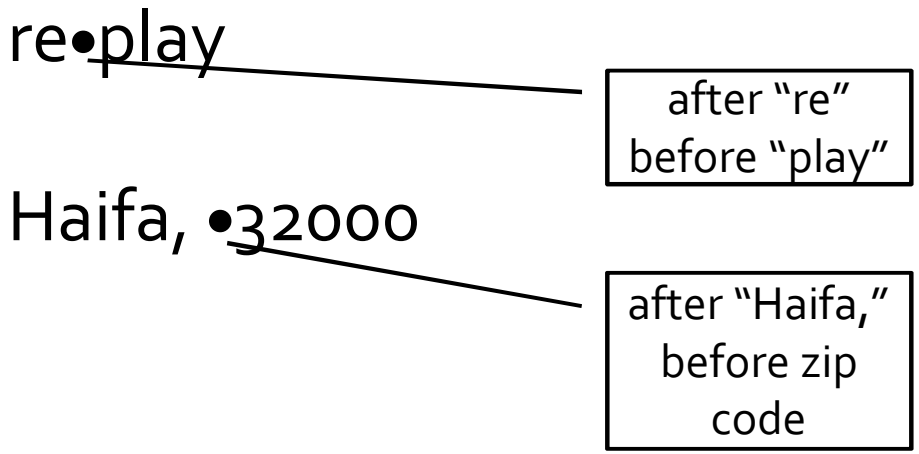
VS = all prefixes of "dependent on" that are longer than 2 characters and shorter than 10 characters

Generalizing String Searching

- a hypothesis classifies positions as "true" when surrounding text matches the search string, "false" otherwise
- can define generality order
- $h1 \sqsubseteq h2$ iff set of positions covered by h1 is a subset of the set of positions covered by h2

Conjunctive String Searching

- string conjunction for left and right search hypotheses



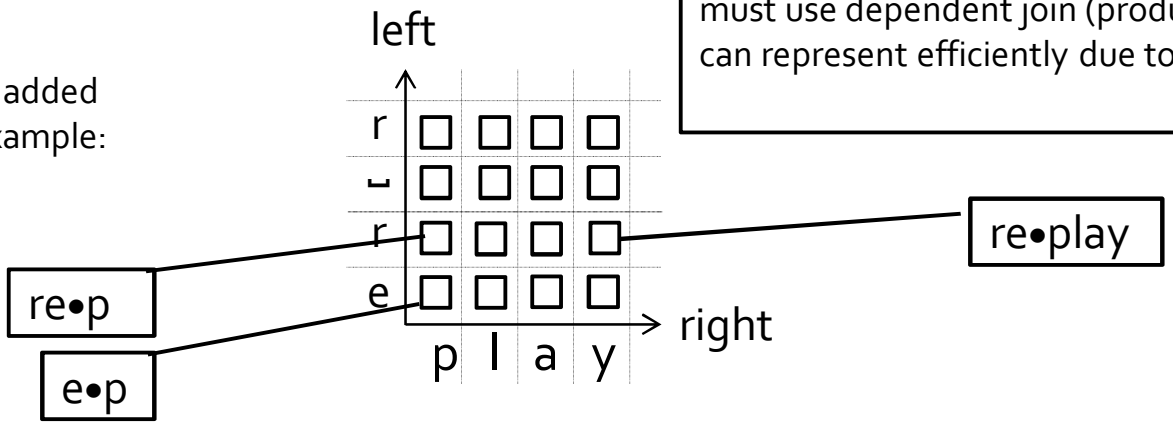
Conjunctive String Searching

A display was rendered for re+play. We re+played it.

shortest consistent hypothesis in the left-search space

independent join can only represent rectangles…
must use dependent join (product)
can represent efficiently due to continuity

assume we added negative example: de•plane



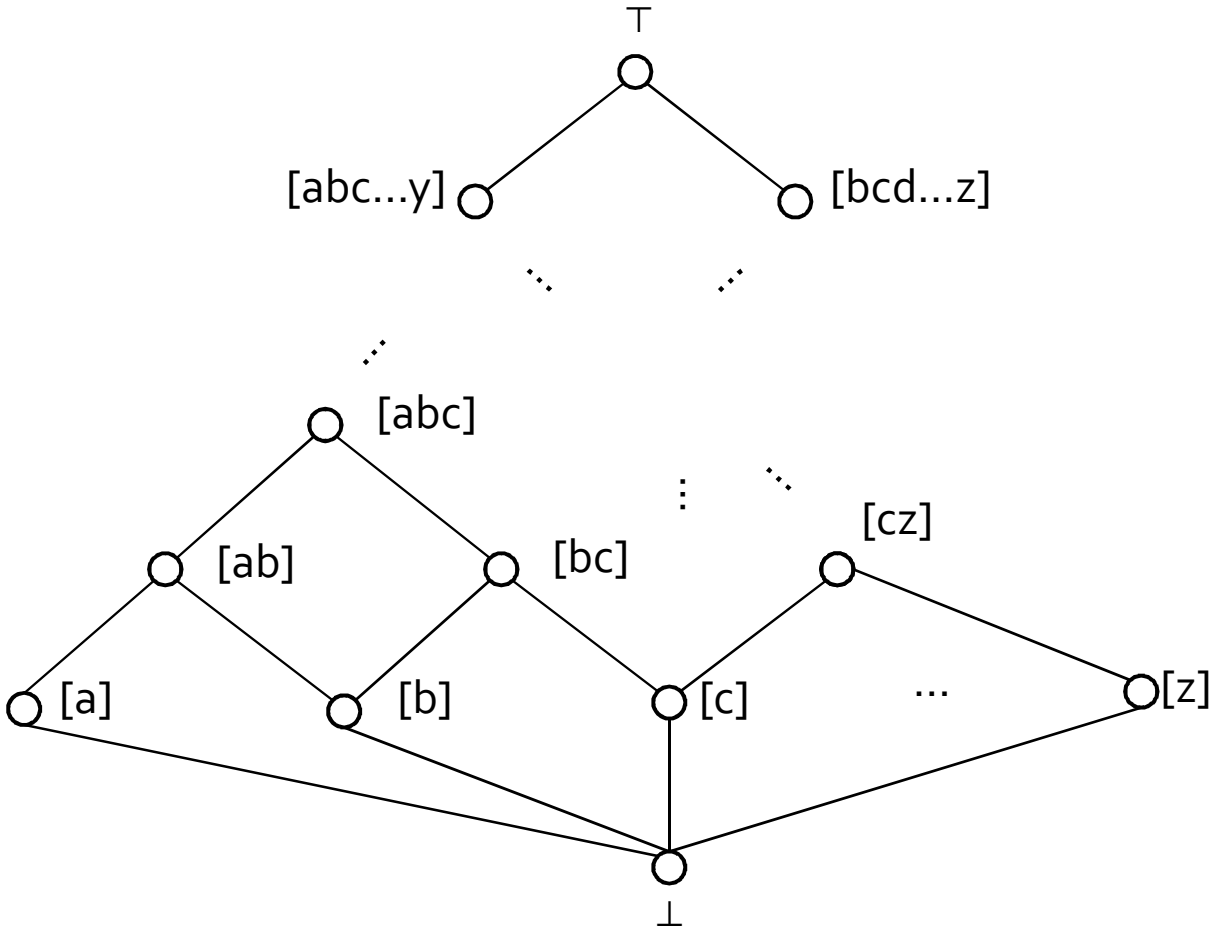
target hypothesis: re•play

Can we use independent VSs – one for left ("re") and one for right ("play")?

Disjunctive String Searching

- "move to the next occurrence of <UL> or <DL>"
- difficult to learn
- h = "disjunction of all observed examples" is always valid
- example
  - search for the next occurrence of any single token from a set of "allowed' tokens
  - positive example: token target location
  - negative examples: all tokens that were skipped to reach the target

Example



example: user moves to “a”, skips “b” and “c”
VS: all charater-class hypotheses that contain “a” and do not contain “b” and “c”

Example

- alphabet: a, b, c
- text: abcbac
- target hypothesis: {b,c} (move to next b or c)
- d1 = •abcbac ⟶ a•bcbac

- no set containing "a" is consistent with d1
- version space only contains {b} and {b,c}

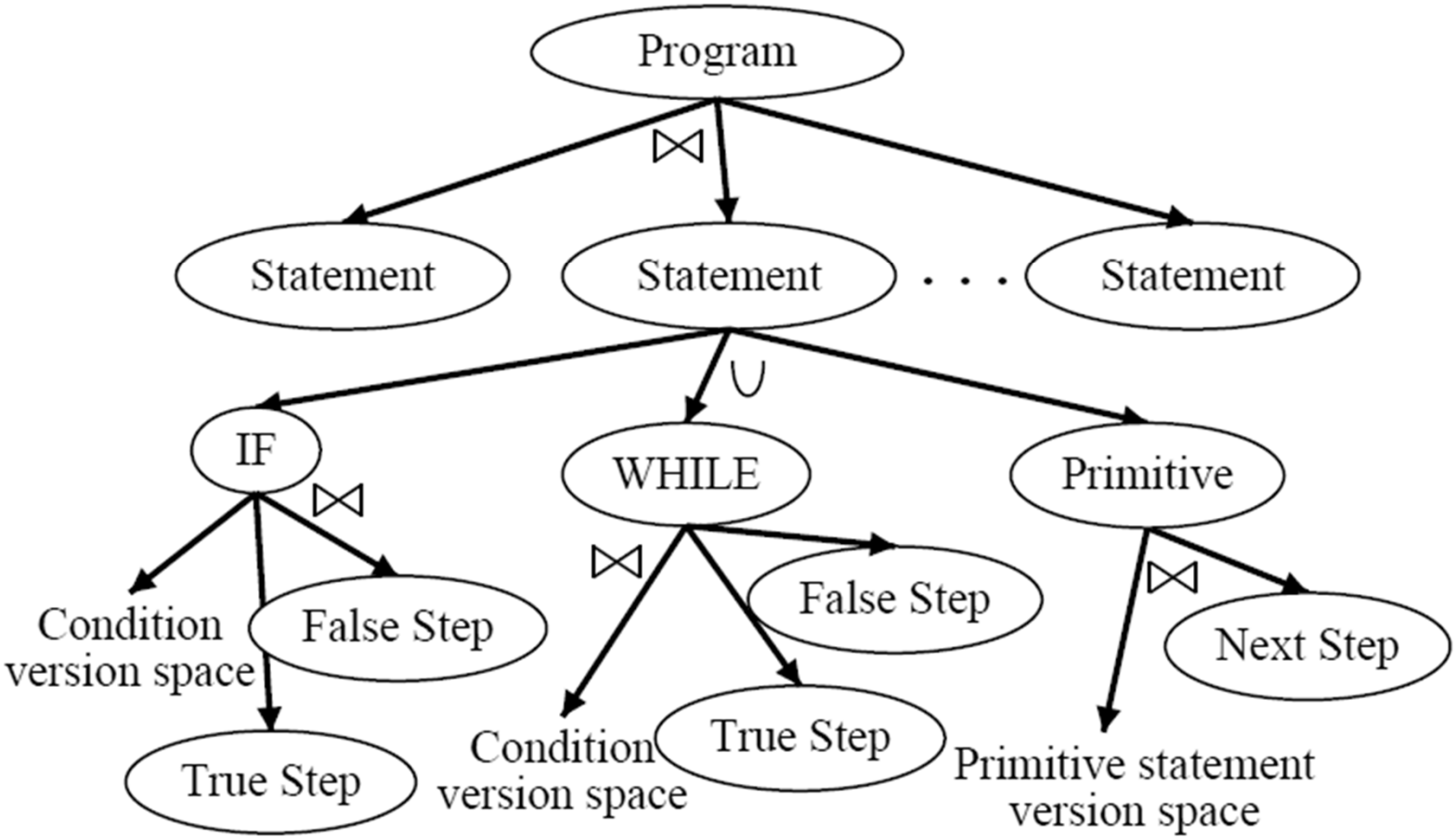
Scenario	train	total
c++comments	1	5
column-reordering	1	14
country-codes	1	4
modify-to-rgb-calls	1	20
number-fruits	1	14
subtype-interaction	1	3
xml-comment-attribute	1	24
addressbook	2	6
citation-creation	2	13
html-comments	2	13
latex-macro-swap	2	8
number-citations	2	13
number-iterations	2	7
smartedit-results	2	27
zipselect	2	6
game-score	3	7
html-latex	3	7
indent-voyagers	3	32
mark-format	3	6
bold-xyz	4	50
citation-to-bibtex	5	10
bindings	6	11
boldface-word	6	11
ul-to-dl	6	7
OKRA	10	14
outline	10	14
pickle-array	19	117

Experimental results

Very few examples needed!

Results indicate examples that must be demonstrated, out of total number of examples

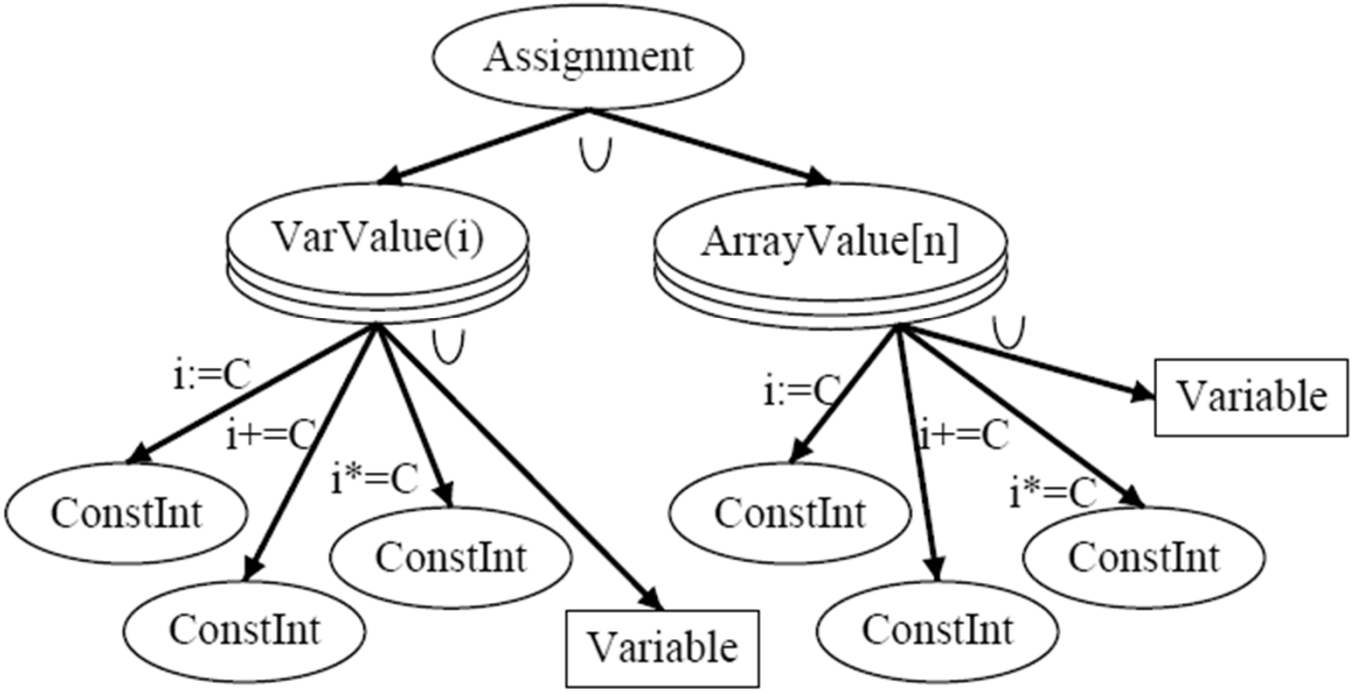
Learning Programs from Traces



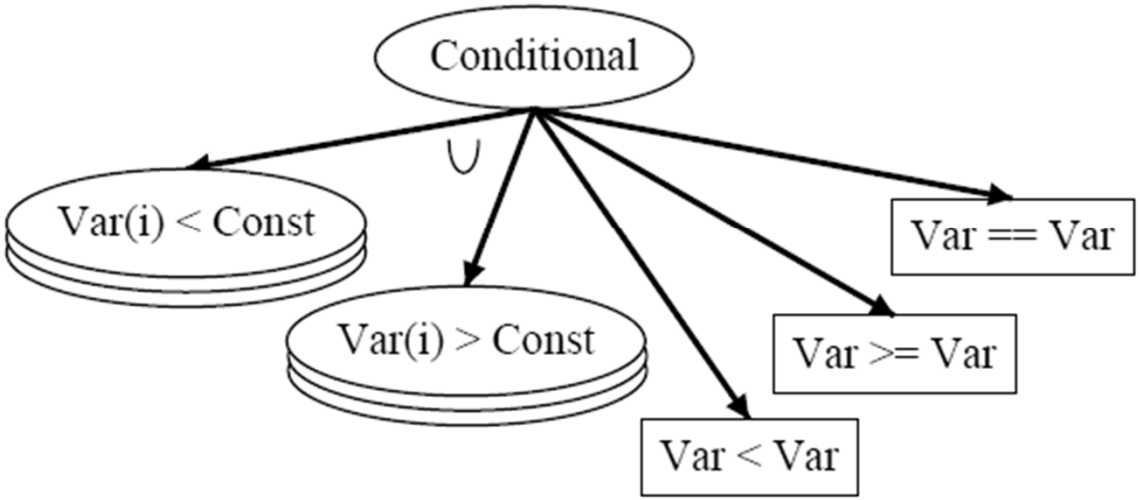
Learning Programs from Traces

- State configuration
  - incomplete: state contains subset of variables, some relevant variables hidden
  - variables observable: state includes all variables in the program
  - step observable: variable observable + unique identification of the step executed between every pair of states
  - fully observable: step observable + change predicates indicating which variables have changed

Primitive Statements



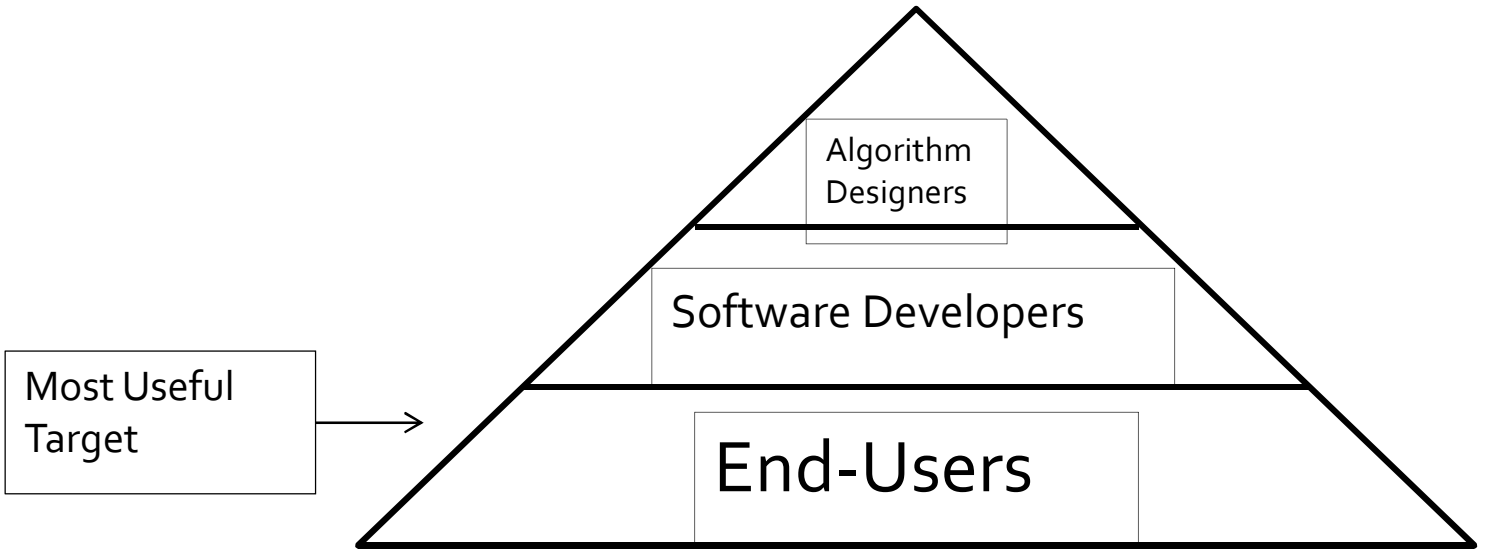
Conditionals



AUTOMATING STRING PROCESSING IN SPREADSHEETS USING INPUT-OUTPUT EXAMPLES

Sumit Gulwani

Potential Consumers of Synthesis Technology



Pyramid of Technology Users

Example

Input v_1	Output
(425)-706-7709	425-706-7709
510.220.5586	510-220-5586
235 7654	425-235-7654
745-8139	425-745-8139

Format phone numbers

Language for Constructing Output Strings

Guarded Expression G := Switch((b_1,e_1), …, (b_n,e_n))

String Expression e := Concatenate(f_1, …, f_n)

Base Expression f := s // Constant String

| SubStr(v_i, p_1, p_2)

Index Expression p := k // Constant Integer

| Pos(r_1, r_2, k) // k^{th} position in string whose left/right side matches with r_1/r_2

Notation: SubStr2(v_i,r,k) ≡ SubsStr(v_i,Pos(ε,r,k),Pos(r,ε,k))

- Denotes k^{th} occurrence of regular expression r in v_i

Example: format phone numbers

Input v_1	Output
(425)-706-7709	425-706-7709
510.220.5586	510-220-5586
235 7654	425-235-7654
745-8139	425-745-8139

Switch((b_1, e_1), (b_2, e_2)), where

$b_1 \equiv$ Match(v_1,NumTok,3), $b_2 \equiv \neg$Match(v_1,NumTok,3),

$e_1 \equiv$ Concatenate(SubStr2(v_1,NumTok,1), ConstStr("-"),
SubStr2(v_1,NumTok,2), ConstStr("-"),
SubStr2(v_1,NumTok,3))

$e_2 \equiv$ Concatenate(ConstStr("425-"),SubStr2(v_1,NumTok,1),
ConstStr("-"),SubStr2(v_1,NumTok,2))

Key Synthesis Idea: Divide and Conquer

Reduce the problem of synthesizing expressions into sub-problems of synthesizing sub-expressions.

- Reduction requires computing *all* solutions for each of the sub-problems:
  - This also allows to rank various solutions and select the highest ranked solution at the top-level.
  - A challenge here is to efficiently represent, compute, and manipulate huge number of such solutions.

- I will show three applications of this idea in the talk
  - Read the paper for more tricks!

Synthesizing Guarded Expression

Goal: Given input-output pairs: (i_1,o_1), (i_2,o_2), (i_3,o_3), (i_4,o_4), find P such that $P(i_1)=o_1$, $P(i_2)=o_2$, $P(i_3)=o_3$, $P(i_4)=o_4$.

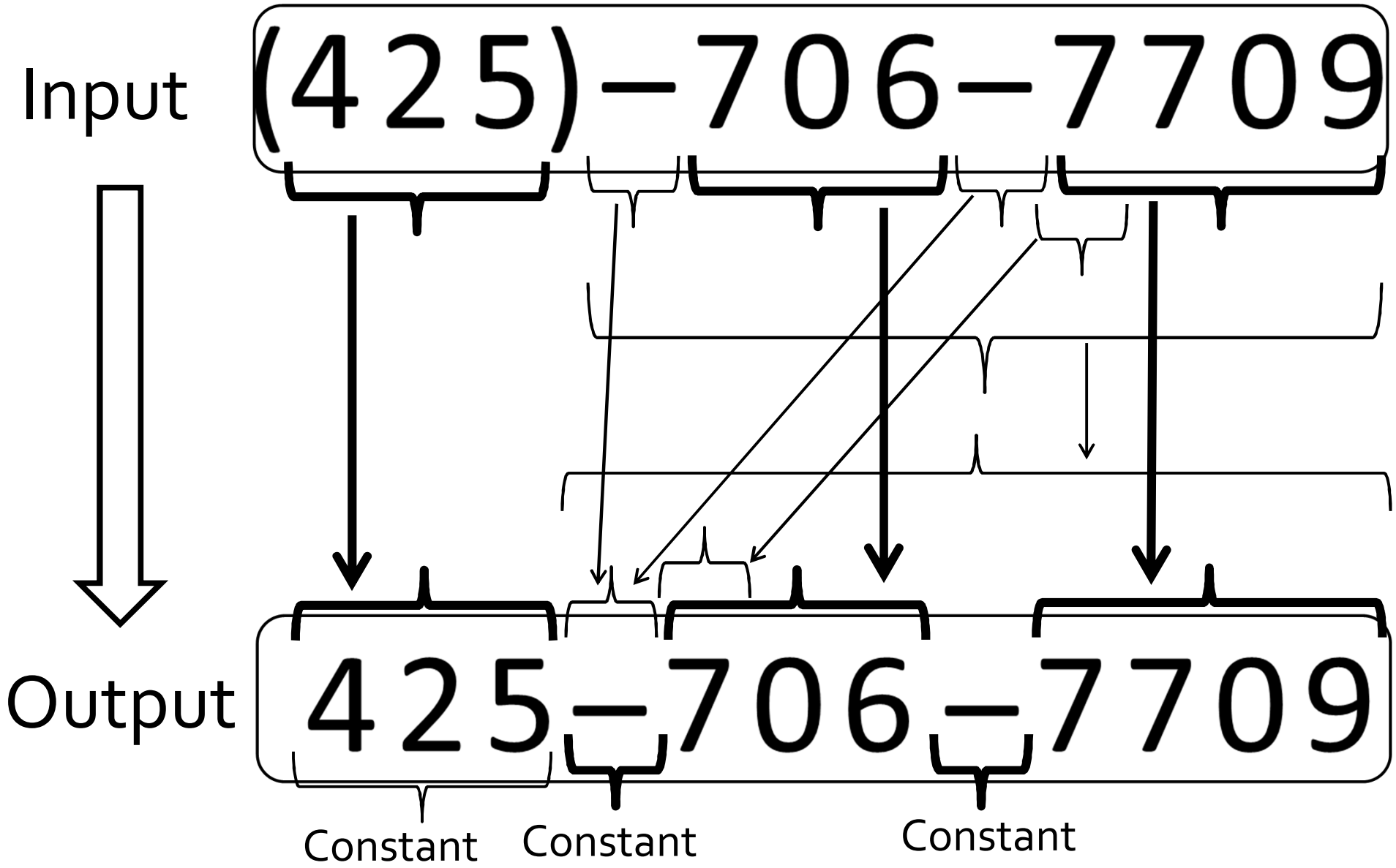
Application #1: Reduce the problem of learning guarded expression P to the problem of learning string expressions for each input-output pair.

Algorithm:

1. Learn set S_1 of string expressions s.t. $\forall$e in S_1, $[[e]]\ i_1 = o_1$. Similarly compute S_2, S_3, S_4.
Let $S = S_1 \cap S_2 \cap S_3 \cap S_4$.

2(a) If $S \neq \emptyset$ then result is Switch((true,S)).

Example: Various choices for a String Expression



Synthesizing String Expressions

Number of all possible string expressions (that can construct a given output string o_1 from a given input string i_1) is exponential in size of output string.

Application #2: To represent/learn all string expressions, it suffices to represent/learn all base expressions for each substring of the output.

- # of substrings is just quadratic in size of output string!
- We use a DAG based data-structure, and it supports efficient intersection operation!

Example: Various choices for a SubStr Expression

Various ways to extract "706" from "425-706-7709":

- Chars after 1st hyphen and before 2nd hyphen.
  Substr(v_1, Pos(HyphenTok,ε,1), Pos(ε,HyphenTok,2))
- Chars from 2nd number and up to 2nd number.
  Substr(v_1, Pos(ε,NumTok,2), Pos(NumTok,ε,2))
- Chars from 2nd number and before 2nd hyphen.
  Substr(v_1, Pos(ε,NumTok,2), Pos(ε,HyphenTok,2))
- Chars from 1st hyphen and up to 2nd number.
  Substr(v_1, Pos(HyphenTok,ε,1), Pos(ε,HyphenTok,2))

⋮

Synthesizing SubStr Expressions

The number of $\mathtt{SubStr}(v,p_1,p_2)$ expressions that can extract a given substring w from a given string v can be large!

Application #3: To represent/learn all SubStr expressions, we can independently represent/learn all choices for each of the two index expressions.

- This allows for representing and computing $O(n_1 * n_2)$ choices for SubStr using size/time $O(n_1+n_2)$.

Back to Synthesizing Guarded Expression

Goal: Given input-output pairs: (i_1,o_1), (i_2,o_2), (i_3,o_3), (i_4,o_4), find P such that $P(i_1)=o_1$, $P(i_2)=o_2$, $P(i_3)=o_3$, $P(i_4)=o_4$.

Algorithm:

1. Learn set S_1 of string expressions s.t. $\forall$e in S_1, $[[e]]\ i_1 = o_1$. Similarly compute S_2, S_3, S_4. Let $S = S_1 \cap S_2 \cap S_3 \cap S_4$.

2(a). If $S \neq \emptyset$ then result is Switch((true,S)).

2(b). Else find a smallest partition, say $\{S_1,S_2\}$, $\{S_3,S_4\}$, s.t. $S_1 \cap S_2 \neq \emptyset$ and $S_3 \cap S_4 \neq \emptyset$.

3. Learn boolean formulas b_1, b_2 s.t.

b_1 maps i_1, i_2 to true and i_3, i_4 to false.

b_2 maps i_3, i_4 to true and i_1, i_2 to false.

4. Result is: Switch((b_1,$S_1 \cap S_2$), (b_2,$S_3 \cap S_4$))

Ranking Strategy

- Prefer shorter programs
  - Fewer number of conditionals
  - Shorter string expression, regular expressions
- Prefer programs with fewer constants

Recap

- SMARTedit
  - learn programs (macros) for repetitive editing tasks
  - version space algebra to learn actions
- String processing in spreadsheets
  - automate spreadsheet string transformations
  - version space algebra to learn actions
  - many other clever tricks to actually make it work