More Knowledgeable and Expressive Chunking

Mazin Assanie
University of Michigan
33rd Soar Workshop
mazina@umich.edu

Four Sections

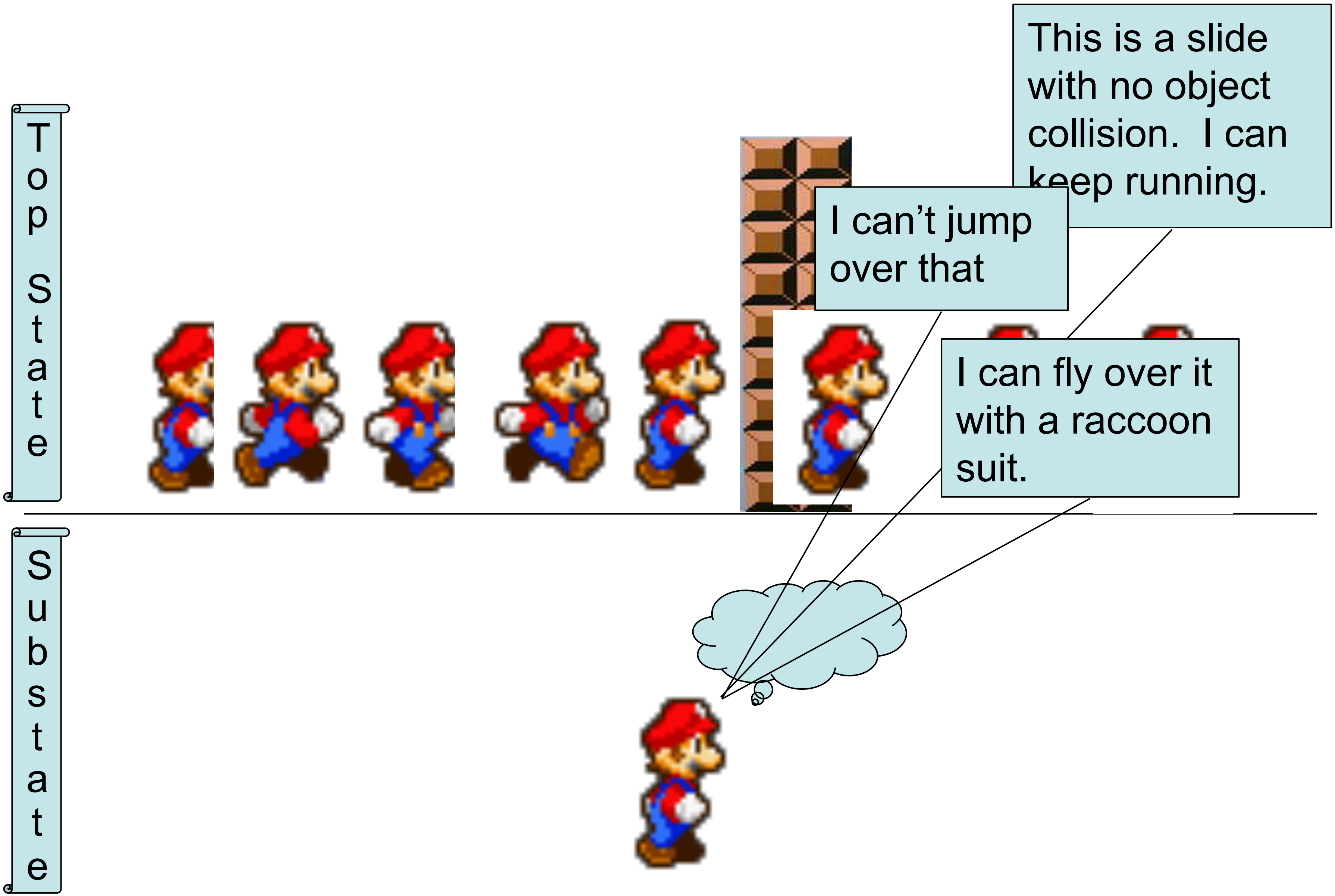
1. 9.4-Minute Intro to Chunking

   New 9.4 Chunking Features

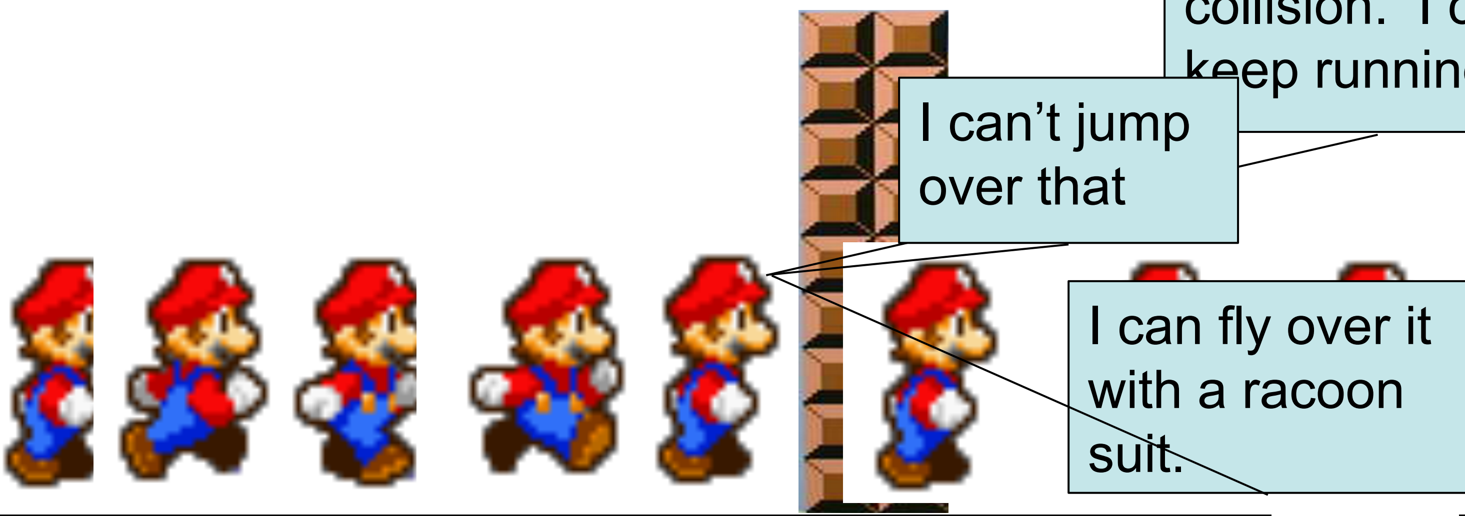
2. General Variablization of Symbols
3. Constraints on Variables
4. Chunking Operator Desirability Knowledge

What is chunking?

- Automatic mechanism that creates productions which summarize problem-solving.
- These chunks will fire in future similar situations avoiding the same problem-solving.


Top State
Substate
This is a slide with no object collision. I can keep running.
I can't jump over that
I can fly over it with a raccoon suit.

Top State



Substate

How Chunking Learns

1. **Dependency analysis**
   - Analyzes substate's problem-solving to determine "what's necessary" to produce results
2. **Variablization**
   - Abstracts away from specific working memory elements
   - Generalizes problem solving to other situations with similar relationships between symbols
3. **Adds Constraints**
   - Increases specificity by requiring that a variable satisfies tests on its value

Dependency Analysis

- Determines all working memory elements linked to a higher level state that were used in a substate to produce a *result*.
- A *result* is working memory element that is added *to a higher level state*.
- Algorithm is called backtracing.
- This set of working memory elements compiled by backtracing will become the left-hand side of a chunk.

Simple Backtracing Example

- Grading agent that subgoals to determine whether a student passes
- Agent has four main rules in substate
  - 3 collect information used during problem-solving
  - 1 uses that information and stores a result in the top-state
- Top state contains student info, grades, and the average score.

Problem-Solving Rules in Substate

get*grading*cut-off

if I’m in a substate

average score is <avg-score>

↓

the grade cut-off is <avg-score>

get*grader*bias

if I’m in a substate

grader is not the student

↓

grader is unbiased

get*love

if I’m in a substate

↓

I love grading

*all of these wmes are added to substate

Rule That Creates Result

apply*grade

if I'm in a substate

the grader is unbiased

the grade cut-off is <min-score>

I love grading

student score is > <min-score>

↓

student PASS (in top-state)

Backtracing

Collected Conditions

Top State

- grader is not student
- average score is 75
- maximum score is 99
- student name is Mary
- student score is 92

Substate

get*love
- this is a substate
- → I love grading

get*grading*cut-off
- this is a substate
- average score is 75
- → the grade cut-off is 75

get*grader*bias
- this is a substate
- grader is not the student
- → grader is unbiased

apply*grade
- this is a substate
- grader is unbiased
- the grade cut-off is 75
- I love grading
- student score is 92
- → student PASS (in top-state)

SOAR
Mazin Assanie
University of Michigan Soar Group

University of Michigan
Engineering and Computer Science

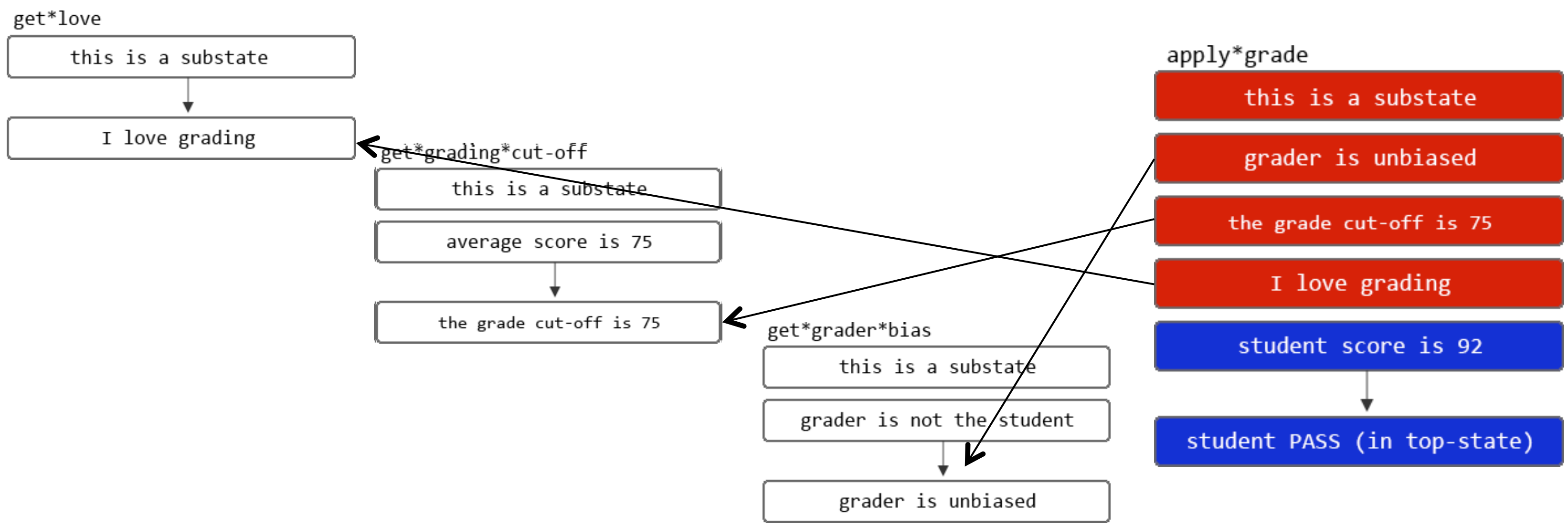
Backtracing

Collected Conditions

Top State

- grader is not student
- average score is 75
- maximum score is 99
- student name is Mary
- student score is 92

Substate

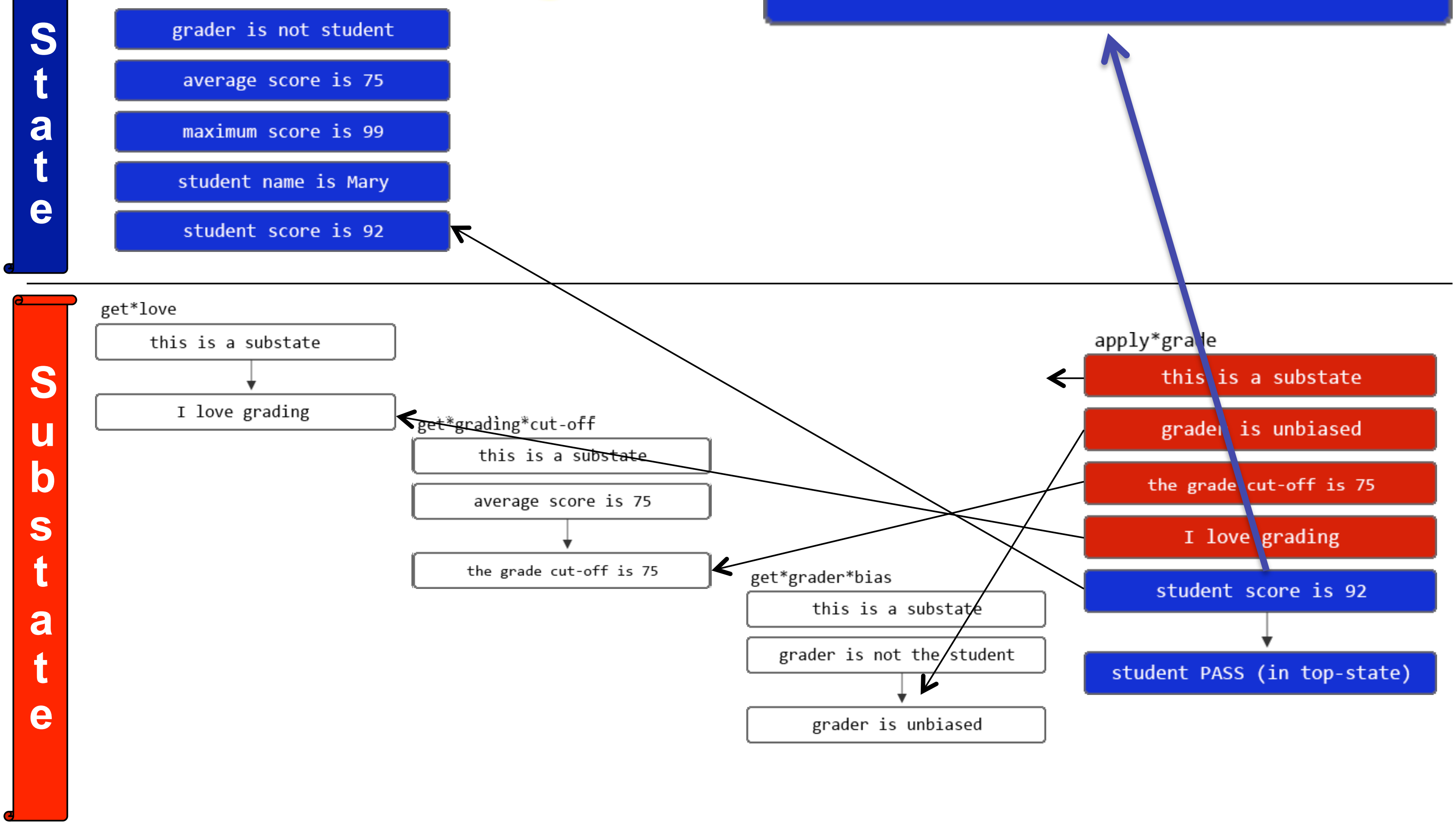


Mazin Assanie
University of Michigan Soar Group

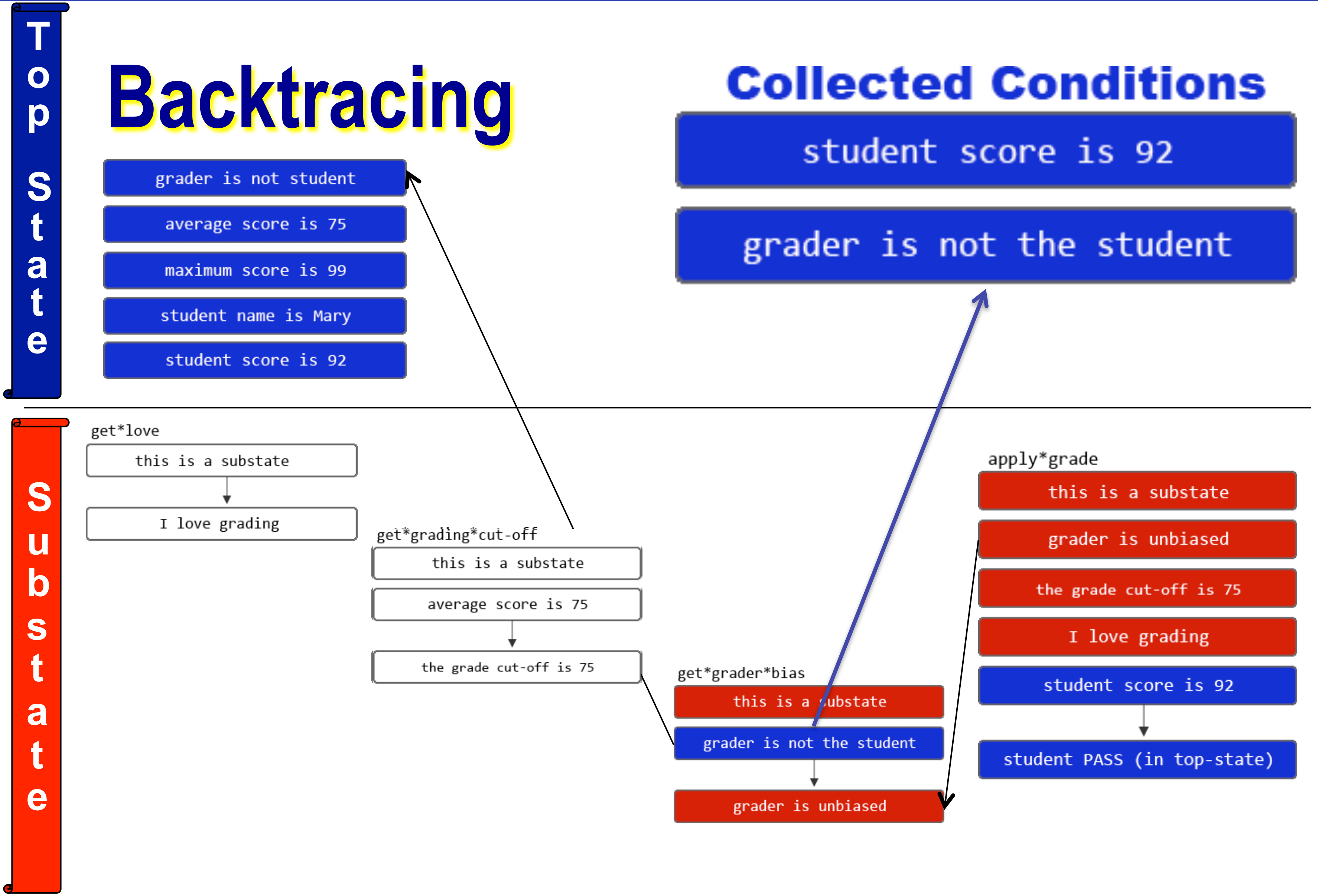
Backtracing

Collected Conditions

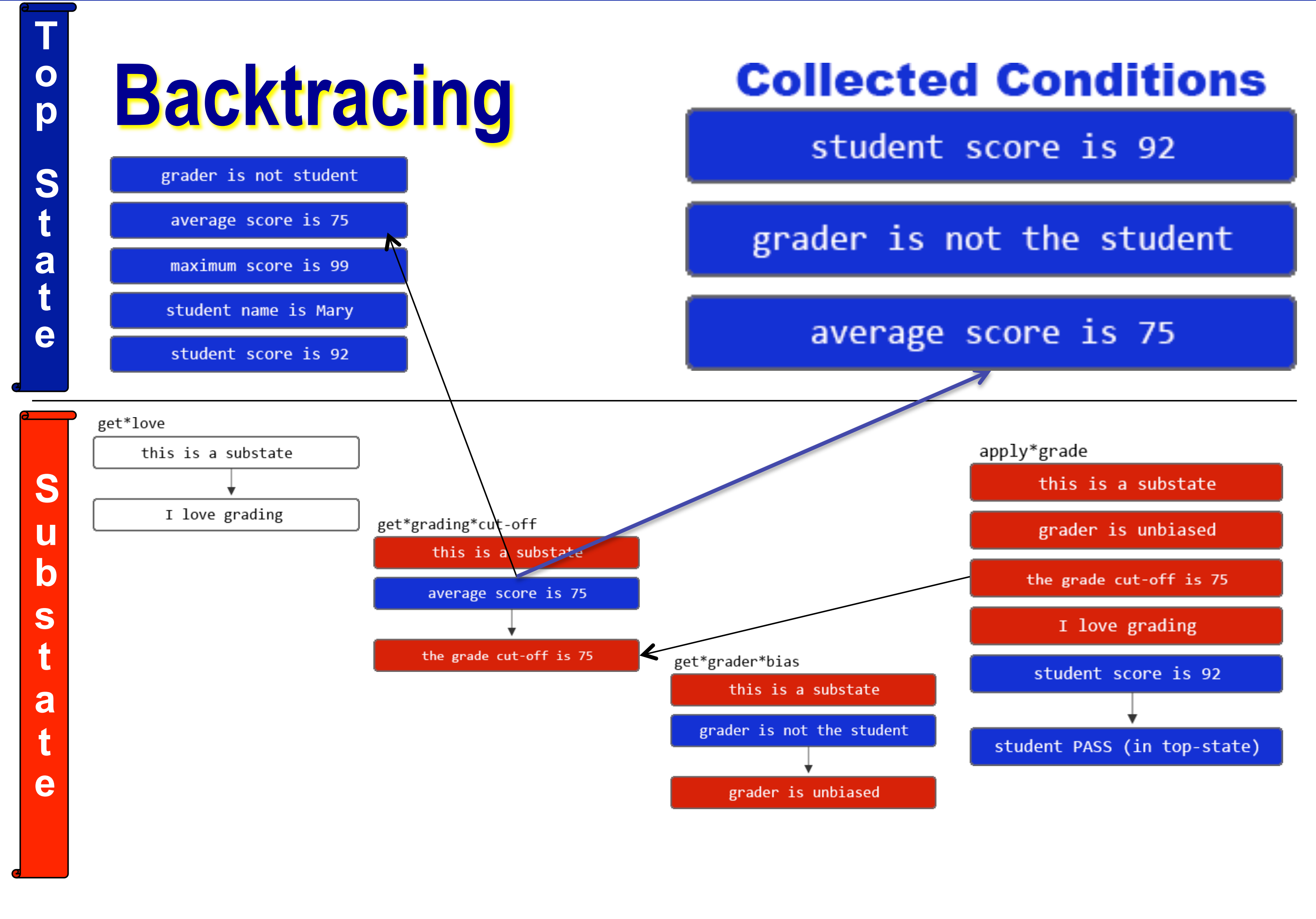
student score is 92



Backtracing



Backtracing



Backtracing

Top State

- grader is not student
- average score is 75
- maximum score is 99
- student name is Mary
- student score is 92

Collected Conditions

- student score is 92
- grader is not the student
- average score is 75

Substate

get*love
- this is a substate
- I love grading

get*grading*cut-off
- this is a substate
- average score is 75
- the grade cut-off is 75

get*grader*bias
- this is a substate
- grader is not the student
- grader is unbiased

apply*grade
- this is a substate
- grader is unbiased
- the grade cut-off is 75
- I love grading
- student score is 92
- student PASS (in top-state)

Final Chunk

chunk*apply*grade

grader is not the student

average score is 75

student score is 92

↓

student PASS

More Expressive Chunking

Low-Hanging Fruit

- There's knowledge in the original productions that we are not utilizing.
- Previously we erred on the side of caution and made very specific chunks.
- Soar 9.4 will now use this knowledge to make more general yet accurate chunks.

Variablization

Conditions from instantiation that we base a chunk on

```
(S1 ^passing-score  75)
(S1 ^superstate     nil)
(S1 ^student-info   I1)
(S1 ^me-info        M1)
(I1 ^test-score     92)
(I1 ^name           Mary)
```

Chunk being formed

```
(<S1> ^passing-score  75)
(<S1> ^superstate     nil)
(<S1> ^student-info   <I1>)
(<S1> ^me-info        <M1>)
(<I1> ^test-score     92)
(<I1>  ^name          Mary)
```

Variablization

Conditions from original productions

```
(<s1> ^passing-score <min>)
(<s1> ^superstate    nil)
(<s1> ^student-info  <s2>)
(<s1> ^me-info    <m1> { <> <i1> })
(<i1> ^test-score <sc> { > <min> })
(<i1> ^name <name>)
```

Chunk being formed

```
(<S1> ^passing-score  75)
(<S1> ^superstate     nil)
(<S1> ^student-info   <I1>)
(<S1> ^me-info        <M1>)
(<I1> ^test-score     92)
(<I1>  ^name          Mary)
```

- Not utilizing everything that the production tells us about relationships between symbols.
- Not utilizing everything that the production tells us about constraints.

Variablization

Conditions from original productions

```
(<s1> ^passing-score <min>)
(<s1> ^superstate    nil)
(<s1> ^student-info  <s2>)
(<s1> ^me-info    <m1> { <> <i1> })
(<i1> ^test-score <sc> { > <min> })
(<i1> ^name <name>)
```

Chunk being formed

```
(<S1> ^passing-score  <P1>)
(<S1> ^superstate     nil)
(<S1> ^student-info   <I1>)
(<S1> ^me-info        <M1>)
(<I1> ^test-score     <T1>)
(<I1> ^name           <N1>)
```

- Utilizes everything that the production tells us about relationships between symbols.
- Not utilizing everything that the production tells us about constraints.

Variablization

Conditions from original productions

```
(<s1> ^passing-score <min>)
(<s1> ^superstate    nil)
(<s1> ^student-info  <s2>)
(<s1> ^me-info    <m1> { <> <i1> })
(<i1> ^test-score <sc> { > <min> })
(<i1> ^name <name>)
```

Chunk being formed

```
(<S1> ^passing-score <P1>)
(<S1> ^superstate    nil)
(<S1> ^student-info  <I1>)
(<S1> ^me-info         <M1> {<> <I1>})
(<I1> ^test-score     <T1> {> <P1>})
(<I1> ^name          <N1>)
```

- Utilizes everything that the production tells us about relationships between symbols.
- Utilizes everything that the production tells us about constraints.

Comparison of Chunks

```
sp {chunk*apply*grade
   (state <s1> ^passing-score 75
               ^superstate    nil
               ^student-info <s2>
               ^me-info       <m1> { <> <s2> })
    (<s2>      ^test-score    92
               ^name          Mary)
    -->
    (<s1>      ^decision      <d1>)
    (<d1>      ^name          Mary
               ^score         92
               ^grade         PASS)}
```

```
sp {chunk*apply*grade
   (state <s1> ^passing-score <p1>
               ^superstate    nil
               ^student-info  <s2>
               ^me-info       <m1> { <> <s2> })
    (<s2>      ^test-score    <s3> { > <p1> }
               ^name          <n1>)
    -->
    (<s1>      ^decision      <d1>)
    (<d1>      ^name          <n1>
               ^score         <s3>
               ^grade         PASS)}
```

- Note that RHS constant symbols are also variablized based on how their corresponding variable on the LHS is variablized.

Summary

- Chunks will now also variablize numbers, strings and LTIs.
- Chunks conditions can now include complex tests that provide constraints on those variables.
    - Relational (>, >=, <, <=, ⇔)
    - Disjunction between constants
    - Conjunctions of multiple tests

Implications of this Change

- Your chunks will be *more general* and can apply to a wider variety of situations, but they should not become over-general.
- We expect agents will need to learn *fewer* chunks that will become applicable to future situations *sooner*.
- Should be useful to all agents.

More Knowledgeable Chunking

Including Operator Preference
Knowledge In Chunks

A Different Agent Design

- Proposes an operator for both PASS and FAIL with no conditions
- Uses four operator preference rules to choose which grade to give
- Has one application rule that writes to the top-state whether the student passed

Problem-Solving Rules in Substate

```
sp {pref*PASS
   (if a pass operator is proposed)
   (and the student scored over 85)
→
   (PASS operator BEST)}
```

```
sp {pref*PASS*if-I-like_them
   (if a FAIL operator is proposed)
   (and a PASS operator is proposed)
   (and the student scored over 75)
   (and I like the student)
→
   (PASS operator BETTER than
         FAIL operator)}
```

```
sp {pref*FAIL
   (if a FAIL operator is proposed)
   (and the student scored below 90)
→
   (FAIL operator BEST)}
```

```
sp {pref*always-pass-self
   (if a PASS operator is proposed)
   (and I am the student)
→
   (FAIL operator REJECT)}
```

Proposals And Rule That Creates Result

```
sp {propose*pass
   (if I'm in a substate)
→
   (propose PASS operator)}
```

```
sp {propose*fail
   (if I'm in a substate)
→
   (propose FAIL operator)}
```

```
sp {apply*grade
   (if I'm in a substate)
   (and PASS op is selected)
   (and we have student name)
→
   (student PASS)}
```

What We Would Like Agent To Learn

- Chunk that says pass scores over 90
- Chunk that says pass scores over 75 if you like the student
- Chunk that says always pass your own test

What Agent Actually Learns

```
sp {chunk*grade
    (if we have student name)
→
    (student PASS)}
```

Chunk that says pass any student with a name.

Why?

```
sp {apply*grade
    (if I'm in a substate)
    (and PASS op is selected)
→
    (student PASS)}

sp {propose*pass
    (if I'm in a substate)
→
    (propose pass operator)}
```

```
sp {chunk*grade
    (if we have student name)
→
    (student PASS)}
```

Chunking currently only backtraces through these two rules to form this chunk.

What Needs To Be Added

- We need a way to include *why* an operator was selected into the knowledge that summarizes the problem-solving.
  - Operator desirability knowledge
- Must expand chunking's dependency analysis to include this operator desirability knowledge

Context-Dependent Preference Set

- *The set of **relevant** operator desirability preferences that led to the selection of an operator.*
- Every operator application instantiation in a substate has a CDPS.
- Chunking will now include the conditions of the rules that produced the desirability preferences of the CDPS in its dependency analysis.

CDPS For A Liked Score of 89

```
sp {pref*PASS
   (if a pass operator is proposed)
   (and the student scored over 85)
→
   (PASS operator BEST)}

sp {pref*PASS*if-I-like_them
   (if a FAIL operator is proposed)
   (and a PASS operator is proposed)
   (and the student scored over 75)
   (and I like the student)
→
   (PASS operator BETTER than FAIL
    operator)}
```

CDPS For PASS Operator

PASS operator BEST

PASS operator BETTER than FAIL

Chunking now backtraces through the two preferences on the left, which adds the following conditions to the chunk:

1. (the student scored over 75)
2. (I like the student)

What An Agent Learns in 9.4

```
sp {chunk*grade
   (if we have student name)
   (the student scored over 75)
   (I like the student)
→
   (student PASS)}
```

But something was left out…

```
sp {pref*PASS
   (if a pass operator is proposed)
   (and the student scored over 85)
→
   (PASS operator BEST)}
```

```
sp {pref*PASS*if-I-like_them
   (if a FAIL operator is proposed)
   (and a PASS operator is proposed)
   (and the student scored over 75)
   (and I like the student)
→
   (PASS operator BETTER than FAIL)}
```

CDPS For PASS Operator

PASS operator BEST

PASS operator BETTER than FAIL

So, shouldn't we have…

1. (the student scored over 75)
2. (I like the student)
3. (and the student scored over 85)

How do you know which desirability preferences will make it into a chunk?

- Notion of "***relevant*** *operator desirability preferences*" closely linked to the preference semantics Soar uses to choose an operator during the decision phase
- If a preference is used during this process, we add it to the CDPS for that operator.

How Soar Chooses an Operator and Builds the CDPS

Preference Semantics

- Require
- Prohibit/Reject
- Better/Worse
- Best
- Worse
- Indffierent

After each stage, it adds the relevant preferences of that type to the CDPS

So, was something was left out?

```
sp {pref*PASS
   (if a pass operator is proposed)
   (and the student scored over 85)
→
   (PASS operator BEST)}
```

```
sp {pref*PASS*if-I-like_them
   (if a FAIL operator is proposed)
   (and a PASS operator is proposed)
   (and the student scored over 75)
   (and I like the student)
→
   (PASS operator BETTER than FAIL)}
```

CDPS For PASS Operator

PASS operator BEST

PASS operator BETTER than FAIL

So, shouldn't we have…

1. (the student scored over 75)
2. (I like the student)
3. (and the student scored over 85)

No.

Implications of this Change

- Your chunks will become more specific.
  - It may require some agent re-design.
- Some agents that could not previously utilize chunking, will now able to.

Nuggets

- Improved all three sources of chunking power
  1. Dependency analysis via backtracing
     - Determines "what's important" (+ **CDPS**)
  2. Variablization of **ALL symbols**
     - Abstracts away more elements of specific instance
     - Generalizes problem solving to other situations with similar relationships between symbols
  3. Constraints on variables (**ALL test types**)
     - Increases specificity by requiring that a variable in a chunk passes a given predicate, possibly relational

Nuggets

- Including operator desirability preferences in chunks
    - Could have interesting possibilities for RL agents
    - Addresses key source of over-general chunks
    - No significant performance cost

Coal

- General variablization and complex constraints:
  - Still needs debugging. Involved significant changes to kernel.
  - Performance cost not yet evaluated for it or CDPS combined with it.
- You may need to design your agent's problem-solving with the CDPS in mind.