2.5: Soar Syntax

Clare Bates Congdon

April 30, 1996, 3:30-4:00

Soar/IFOR Tutorial

University of Michigan AI Lab

Syntax of Working Memory

Template for WME:

(\texttt{identifier \textasciitilde attribute value})

Template for object in working memory:

(\texttt{identifier \textasciitilde attribute1 value1 \textasciitilde attribute2 value2 ... \textasciitilde attributeN valueN})

\textit{NOTE:} But there is NO semantics to the ordering of the attributes. They are printed in the alphabetical order.

For example, after one elaboration cycle of the blocks world:

\texttt{soar> print s1}

\texttt{(s1 \textasciitilde block C1 \textasciitilde block B1 \textasciitilde block A1 \textasciitilde block T1 \textasciitilde clear C4 \textasciitilde clear C5 \textasciitilde clear C3 \textasciitilde clear C6 \textasciitilde io I1 \textasciitilde ontop C2 \textasciitilde ontop B2 \textasciitilde ontop A2 \textasciitilde problem-space blocks \textasciitilde superstate nil \textasciitilde type state)}

\textit{NOTE:} Identifiers are created at runtime by the Soar architecture and may vary from one run of a program to the next.
Syntax of Working Memory

The value of a WME may be the identifier of another object in working memory; this indicates a link.

E.g., from the previous example of S1, the following are links: T1, C1, B1, and A1.

Remember that links are one-way, although there may be one link in each direction. (Cycles are allowed.)

Working Memory is a Set:
There will never be two duplicate WME’s at the same time.

Illustration of Working Memory

Initial State

Working Memory

Operator

(state

{S1 "type state"
{S1 "thing A1"
{S1 "thing B1"
{S1 "thing C1"
{S1 "thing T1"
{S1 "clear A1"
{S1 "clear B1"
{S1 "clear C1"
{S1 "clear T1"
{S1 "ontop 01"
{S1 "ontop 02"
{S1 "ontop 03"
{S1 "operator 07"

{A1 "type block"
{A1 "name A"
{B1 "type block"
{B1 "name B"
{C1 "type block"
{C1 "name C"
{T1 "type table"
{T1 "name table"

{01 "top-block A1"
{01 "bottom-thing T1"
{02 "top-block B1"
{02 "bottom-thing T1"
{03 "top-block C1"
{03 "bottom-thing T1"

{07 "name move-block"
{07 "moving-block B1"
{07 "destination C1"
Acceptable Preferences in Working Memory

Acceptable preferences for operators appear in both working memory and preference memory.

- These are the only preferences that ever appear in working memory.
- They appear in working memory so that they can appear in the conditions of productions — so that they can be used by operator-comparison productions.

For example, after one decision cycle of the blocks world:

(S1 ^block C1 ^block B1 ^block T1 ^block A1 ^clear C6 ^clear C5 ^clear C3 ^clear C4 ^io I1 ^ontop A2 ^ontop C2 ^ontop B2 ^operator O1 + ^operator O5 + ^operator O2 + ^operator O1 ^operator O6 + ^operator O3 + ^operator O4 + ^problem-space blocks ^superstate nil ^type state)

NOTE: Be sure to notice the difference in working memory between operators that have been proposed for the state and an operator that has been selected for the state. This is easy to confuse.

Syntax of Working Memory: Timetags

To look at the WME's individually, the wmes command could be used:

soar> wmes s1
(5: S1 ^block T1)
(6: S1 ^block C1)
(7: S1 ^block B1)
(8: S1 ^block A1)
...
(3: S1 ^io I1)
(10: S1 ^ontop B2)
(11: S1 ^ontop A2)
(9: S1 ^ontop C2)
(4: S1 ^problem-space blocks)
(2: S1 ^superstate nil)
(1: S1 ^type state)
Timetags in Working Memory

The **wmes** command prints individual WME’s, each preceded by an integer timetag.

- Technically, WME’s are quadruples, because each WME has a timetag.
- But many times we don’t need the timetag, so it is ignored.

As Soar runs, a WME may be created, removed, and created again.

But it is not really the same WME when it is recreated — it is a *new* WME.

This is important for matching productions:
- A production will match only once for a given set of WME’s.
- But the match is established using timetags.

*NOTE*: Timetags are determined at runtime, and they will not appear in productions.

---

Multi-Valued Attributes in Working Memory

There may be multiple working memory elements that have the same identifier and attribute, but that each have different values. When this happens, we say the attribute is a *multi-valued attribute*, which is often shortened to be *multi-attribute*.

Multi-attributes are created using parallel preferences — to suggest that the value can exist in parallel with other values.

In the blocks-world, the blocks are examples of multi-attributes.
Syntax of Production Memory

\[
sp \{\text{apply}*\text{move-block}*\text{add-new-clear} \\
\text{(state } <s> \text{ } \uparrow\text{operator } <o> \\
\text{ } \uparrow\text{ontop } <on>) \\
\text{(<o> } \uparrow\text{name move-block} \\
\text{ } \uparrow\text{moving-block } <bl> \\
\text{ } \uparrow\text{destination } <d>) \\
\text{(<on> } \uparrow\text{top-block } <bl> \\
\text{ } \uparrow\text{bottom-thing } \{ <s> <d> <bt> \} ) \\
\text{(<bt> } \uparrow\text{type block} \\
\text{ } \longrightarrow \\
\text{(<s> } \uparrow\text{clear } <bt> + &)}
\]

Name of production is a constant

Conditions and actions use variables, as in <o>
- bindings are not necessarily unique unless forced,
  as in third condition: \{ <s> <d> <bt> \}

Production Syntax, part 2

Production is *matched* (instantiated) only when:
- conditions describe a subset of working memory
- using consistent variable bindings

Production *fires* when matched:
- actions create *preferences* for WME’s
- do not modify working memory directly

There may be many different matches of the same production:
- each distinct match must use different variable bindings
- production will match only once for a specific set of working memory elements (use timetags to distinguish)
- each different match is called an *instantiation*. 
Illustration of Production Instantiations

To get a sense of production instantiations as Soar runs, run the blocks-world again for one elaboration cycle and use the `matches` command to look at the matches of the production that proposes the `move-block` operator:

```
soar> init-soar
soar> source blocks.soar
soar> run 1 e
  0: =>S: S1
Initial state has a, b, and c on the table.
soar> matches
```

Assertions:
blocks-world*propose*move-block (6)
Retractions:

```
soar> matches blocks-world*propose*move-block -wmes
  1 (state <s>
       ~problem-space blocks)
  3 (<s> ~ontop <ontop>)
  3 (<ontop> ~top-block <block1>)
  3 (<block1> ~type block)
 12 (<s> ~clear <clear1>)
  3 (<clear1> ~block <block1>)
  3 (<ontop> ~bottom-block <b1>)
 12 (<s> ~clear <clear2>)
  6 (<clear2> ~block { <> <block1> <> <b1> <block2> })
6 complete matches.
*** Complete Matches ***
(4: S1 ~problem-space blocks)
(10: S1 ~ontop B2)
(26: B2 ~top-block B1)
(18: B1 ~type block)
(14: S1 ~clear C4)
(31: C4 ~block B1)
(27: B2 ~bottom-block T1)
(13: S1 ~clear C5)
(32: C5 ~block C1)
```
Illustration of Production Instantiations

soar> print -depth 2 s1

(S1 ^block C1 ^block B1 ^block A1 ^block T1 ^clear C4 ^clear C5 ^clear C3 ^clear C6 ^io I1 ^ontop C2 ^ontop B2 ^ontop A2 ^problem-space blocks ^superstate nil ^type state)

(I1)
(A1 ^name a ^type block)
(B1 ^name b ^type block)
(C1 ^name c ^type block)
(T1 ^name table ^type table)
(A2 ^bottom-block T1 ^top-block A1)
(B2 ^bottom-block T1 ^top-block B1)
(C2 ^bottom-block T1 ^top-block C1)
(C3 ^block A1)
(C4 ^block B1)
(C5 ^block C1)
(C6 ^block T1)

Linkage in Production Conditions and Actions

One of the conditions in a production must refer to a state (or impasse) in working memory. The convention is to write this as the first condition of the production (we'll call it the first condition from now on).

The first condition also starts with the symbol state or impasse to identify it as the root of the links in the production.

The rest of the conditions must be linked to this condition, but the linkage may be indirect, through other conditions in the production.

The preferences specified in the actions of the production must also be linked to the first condition; this linkage may be indirect, through other conditions, or through other preferences in the actions.
More Production Syntax

Acceptable preferences for operators are the only preferences that appear in working memory, and are, therefore, the only preferences that may appear in the conditions of productions.

You can use predicates, disjunctions, and conjunctions in production conditions.

You can also specify negated conditions, negated conjunctions of conditions, etc.

These follow certain rules, described in Chapter 4 of the manual.

---

Multi-Attributes in Production Memory

For multi-attributes in production conditions, you don’t have to rewrite the attribute name each time. E.g.:

\(<s1> \neg block A B C\)

With variables, this is a little trickier, since variable bindings are not automatically unique — you must force them to be unique if you intend this:

\(<s1> \neg block <a> \{ <> <a> <b> \} \{ <> <a> <> <b> <c> \}\)

The condition above specifies three blocks. The first is bound to \(<a>\); the second cannot bind to \(<a>\) and should bind to \(<b>\); the third cannot bind to \(<a>\) or \(<b>\) and should bind to \(<c>\).

**NOTE:** The order of the conjunctions does not have to be as shown above. For example, the third multi-attribute has three separate tests: "\(<a>\)", "\(<b>\)", and "\(<c>\)"; these three tests could appear in any order with the same effect.
The Most Common Types of Preferences

<table>
<thead>
<tr>
<th>name</th>
<th>symbol</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acceptable</td>
<td>+</td>
<td>this is a candidate for WM</td>
</tr>
<tr>
<td>reject</td>
<td>−</td>
<td>this should be removed from WM</td>
</tr>
<tr>
<td>reconsider</td>
<td>@</td>
<td>this operator has completed</td>
</tr>
</tbody>
</table>

* parallel & this is a multi-valued attribute
* better, best > this value is better than another
* worse, worst < this value is worse than another
* indifferent = this value is no better or worse than another

* Preferences that may be binary or unary:
When binary, these preferences say that the value can exist in parallel with a specific other value, is better than a specific other value, or is indifferent to a specific other value; when unary, these preferences apply to any other competing values.

NOTE: There are other types of preferences, but you probably won’t need them for now.

Preferences in Production Actions

There are a few shorthand conventions for specifying preferences in the actions of a production:

- Multiple preferences for the same identifier and attribute may be strung together without repeating the attribute.
- Multiple preferences for the same value may be strung together without repeating the value.
- The plus symbol + for acceptable preferences may be omitted in most situations.
Preferences in Production Actions

For example:

$<$s$> \ ^{\text{block}} <a> + \ ^{\text{block}} <a> \&$

$^{\text{block}} <b> + \ ^{\text{block}} <b> \&)$

could be written as:

$<$s$> \ ^{\text{block}} <a> + <a> \& <b> + <b> \&)$

or as:

$<$s$> \ ^{\text{block}} <a> + \& <b> + \&$)

But note that the plus sign cannot be omitted from this final form. Though instead, it could be written as:

$<$s$> \ ^{\text{block}} <a> <a> \& <b> <b> \&)$

Other Production Actions

There are a number of other actions that can be executed by productions. (These are usually called RHS actions, for the “righthand side” of the production.)

For example, (halt) is a RHS action that stops Soar’s execution and returns to the user prompt.

Also, write writes its arguments to standard output, e.g.,

(write |I am moving block | <x> |ontop of block | <y>)

(Note that strings are denoted with vertical bars and that spaces are not automatically added.)

Additional RHS functions are documented in Chapter 4 of the manual.

Also, there are a number of RHS functions implemented specifically for Soar/IFOR. For a listing, see:

http://krusty.eecs.umich.edu/ifor/interface/rhsfuns.html
Shorthand: Attribute-Path Notation

Variables are often used in productions just for the sake of linking one condition to another. Attribute-path notation is a shorthand notation that omits these intermediate variables. Eg:

```
sp {blocks-world*propose*move-block
  (state <s> "clear <clear1> <clear2>
  "ontop <ontop>)
  ("clear1" "block <block1>")
  ("clear2" "block { < <block1> <block2> })
  ("block1" "type block")
  ("ontop" "top-block <block1>
  "bottom-block <> <block2>)
  -->
  ...
}
```

```
sp {blocks-world*propose*move-block
  (state <s> "clear.block <block1>
  "clear.block { <> <block1> <block2> }
  "ontop <ontop>)
  ("block1" "type block")
  ("ontop" "top-block <block1>
  "bottom-block <> <block2>)
  -->
  ...
}
```

Soar internally rewrites productions, expanding the attribute-path notation in the process. Therefore, if you print the production shown in attribute-path notation on the previous slide:

```
soar> print blocks-world*propose*move-block-wmes
```

```
sp {blocks-world*propose*move-block
  (state <s> "ontop <ontop>)
  ("ontop" "top-block <block1> "bottom-block <b*1>)
  ("block1" "type block")
  (state <s> "clear <c*1> "clear <c*2>)
  ("c*1" "block <block1>)
  ("c*2" "block { <> <block1> <> <b*1> <block2> })
  -->
  ("s" "operator <o> +
  ("o" "name move-block + "moving-block <block1> + "destination <block2> +})
```
Shorthand: Attribute-Path Notation

Use attribute-path notation cautiously — it is very easy to make mistakes. For example, the `<ontop>` relation from our previous example cannot be written with attribute-path notation:

```
sp {blocks-world*don't-do-this
  (state <s> ~clear.block <block1> ~clear.block { <> <block1> <block2> }
    ~ontop.top-block <block1> ~ontop.bottom-block <> <block2>)
  (<block1> ~type block)
  -->
  ...}
```

```
soar> print blocks-world*don't-do-this
sp {blocks-world*don't-do-this
  (state <s> ~ontop <o*2> ~ontop <o*1> ~clear <c*1> ~clear <c*2>)
  (<o*2> ~bottom-block <b*1>)
  (<o*1> ~top-block <block1>)
  (<block1> ~type block)
  (<c*1> ~block <block1>)
  (<c*2> ~block { <> <block1> <> <b*1> <block2> })
  -->
  ...}
```

Syntax of Preference Memory

Unary preferences are quadruples, e.g., to propose O3 as an operator:

```
(S1 ~operator O3 +)
```

Binary preferences are quintuples, e.g., to say that O3 is a better choice for operator than O4:

```
(S1 ~operator O3 > O4)
```

Like working memory, preferences in preference memory must also be linked to a state or impasse object, or they will be removed (automatically).

Unlike working memory, preference memory is not a set. There may be two or more preferences that contain exactly the same information. (But duplicate preferences do not affect the resolution process.)
A Bit of Tcl Syntax

Soar uses a Tcl front-end, so the Soar program is read by the Tcl parser.

Tcl is case sensitive, so Soar is case sensitive.

Outside of productions, we must use Tcl syntax. In particular, comments:

- are denoted with a pound sign: #
- must be the first non-white character on a new line
- for "inline" comments, can force a newline with a semicolon

```tcl
# this comment is alone on a line
print s1 ;# this is an 'inline' comment
```

Other Tcl code can be included in same file with productions, but stylistically, it's best to do so sparingly.

Comments Inside of Productions

The guts of productions are not parsed by the Tcl parser, but comments follow similar conventions by design.

Specifically, comments inside productions are preceded by a pound sign, # and are terminated by the end of a line.

Example:

```tcl
sp {blocks-world*select*move-block*indifferent
   (state <s> ^operator <o> +) # for a proposed operator
   (<o> ^name move-block) # that is named move-block
   -->
   (<<s> ^operator <o> =)}) # make the operator unary indifferent
```

(Unlike Tcl, # does not have to be the first non-whitespace character on a line.)
2.6: Soar Operators

Peter Hastings
(composed by Clare Bates Congdon)

April 30, 1996, 4:00 – 4:30

Soar/IFOR Tutorial
University of Michigan AI Lab

Repeat: Problem-Solving Functions in Soar

Selecting an operator:
1. Proposing candidate operators P
2. Comparing candidate operators P
3. Selecting a single operator D

Applying an operator:
4. Applying the operator P
5. Terminating the operator P

Four of these functions involve firing productions (P) (and the consequent creation of preferences).

The third function involves making a decision (D), accomplished via the evaluation of preferences.
Operator Proposal and Comparison

An **operator-proposal** production creates an acceptable preference for an operator. (Or a require preference, but that's unusual.)

- Often, the proposal also creates all substructure of the operator, but it doesn't have to. (Substructure of an operator could instead be created with **operator-creation** productions.)

An **operator-comparison** production creates better, best, worse, worst, or indifferent preferences for an operator.

Sometimes, the functions of proposal and comparison are combined into a single production.

---

**Blocks Example: Operator Proposal/Comparison**

```
sp (propose-move-block
 (state <s> `thing <b> `thing (`<b> `<th>)
 `clear `<b> `clear `<th`
 `ontop `<on>)
 (on `<top-block `<b> `bottom-thing `<th>)
)--
 (<s> `operator `<o> +=)
 (<o> `name move-block
 `moving-block `<destination `<th>))
```

This production and this state lead to 6 instantiations, such as:

```
sp (propose-move-block
 (state S1 `thing A1 `thing (`< A1 T1)
 `clear A1 `clear B1`
 `ontop 01)
 (01 `<top-block A1 `<bottom-thing `<B1)
)--
 (S1 `operator `<o> +=)
 (<o> `name move-block
 `moving-block A1 `<destination B1))
```

---

Note: variables in the actions will be bound to new identifiers when the production fires if they do not correspond to variables specified in the conditions.
Blocks Example: Operator Proposal/Comparison

```prolog
example instantiation:
sp (propose "move-block"
    (state S1 "thing A1 "thing (<-> A1 T1)
     "clear A1 "clear B1 "ontop O1)
    (O1 "top-block A1 "bottom-thing <-> B1)
   -->
   (S1 "operator <-> =)
   (<-> "name move-block "moving-block A1 "destination B1))
)
```

Each instantiation proposes an operator that moves a block onto of another block:

```prolog
(04 "name move-block "moving-block A1 "destination B1)
(05 "name move-block "moving-block A1 "destination C1)
(06 "name move-block "moving-block B1 "destination A1)
(07 "name move-block "moving-block B1 "destination C1)
(08 "name move-block "moving-block C1 "destination A1)
(09 "name move-block "moving-block C1 "destination B1)
```

Each operator is proposed with preferences that say that the operator is acceptable for the state and that the operator is indifferent to all other operators proposed:

```prolog
(S1 "operator 04 =) (S1 "operator 04 =)
(S1 "operator 05 =) (S1 "operator 05 =)
(S1 "operator 06 =) (S1 "operator 06 =)
(S1 "operator 07 +) (S1 "operator 07 +)
(S1 "operator 08 +) (S1 "operator 08 +)
(S1 "operator 09 +) (S1 "operator 09 +)
```

```
[Added to preference memory]
[Added to working memory]
Acceptable preferences for operators are the only preferences that ever appear in working memory.
```

The production above proposes candidate operators by creating acceptable preferences and provides comparisons of these operators by making them indifferent to each other.

---

Operator Selection

Selection of an operator is done by Soar's decision procedure in the decision phase of the decision cycle.

The decision procedure gathers all preferences for operators for the state and attempts to select one of the candidates.

Decision Cycle

- **Elaboration Phase**: During the elaboration phase, productions fire and retract until there are no more productions eligible to fire or retract.
- **Decision Phase**: During the decision phase,
  1. all operator preferences are considered
  2. the preferences are evaluated
  3. the current operator is reconsidered
  4. OR an impasse is reached
  OR an impasse is reached
Blocks Example: Operator Selection

Soar’s decision procedure gathers all the relevant preferences and evaluates them, leading to one of six situations:

1. One operator is preferred over others.
2. Multiple operators are tied, but these tied operators are also indifferent.
3. Multiple operators are tied, but these tied operators are not also indifferent.
4. The preferences are contradictory and no operator can be selected as the winner.
5. No operators are candidates.
6. A new operator can’t be selected because the current operator has not been terminated.

Situation 2 holds in our example; Soar will select one of the six operators at random.

NOTE: Situation 1 also leads to the selection of an operator. Situation 3, 4, 5, or 6 lead to an impasse in problem-solving.
Operator Application

An operator-application production proposes changes to the state.

- Typically, there are multiple productions involved in applying an operator.

- These productions look at the operator and the state in their conditions (and substructure of the state and operator).

- Maybe not all of the operator-application productions for an operator will be applied.

- Or maybe they won’t all apply during the same elaboration cycle.

**NOTE:** Note that productions that suggest changes to the state must not only propose new values but also reject the old values or there will be an attribute impasse.

Also note that the proposal of a new value and the rejection of the old value can be done with two different productions.

---

Blocks Example: Operator Application

```
sp (apply-move-block*remove-old-ontop
  (state <s> ^operator <o>
    ^ontop <on>)
  (<o> ^name move-block
    ^moving-block <mb>
    ^destination <d>)
  (<on> ^top-block <mb>
    ^bottom-thing ( <> <d> <bt> ))
  -->
  (<s> ^ontop <ontop> -))

sp (apply-move-block*remove-old-clear
  (state <s> ^operator <o>
    ^clear <b>)
  (<o> ^name move-block
    ^destination <b>)
  (<b> ^type block)
  -->
  (<s> ^clear <b> -))
```

```
sp (apply-move-block*add-new-ontop
  (state <s> ^operator <o>)
  (<o> ^name move-block
    ^moving-block <mb>
    ^destination <d>)
  -->
  (<s> ^ontop <ontop> + &)
  (<ontop> ^top-block <mb>
    ^bottom-thing <d>))

sp (apply-move-block*add-new-clear
  (state <s> ^operator <o>
    ^ontop <on>)
  (<o> ^name move-block
    ^moving-block <bl>
    ^destination <d>)
  (<on> ^top-block <bl>
    ^bottom-thing ( <> <d> <bt> ))
  (<bt> ^type block
  -->
  (<s> ^clear <bt> + &))
```

There are four separate productions in the blocks-world task used to implement the move-block operator.
Blocks Example: Operator Application

These productions apply the move-block operator

```
sp (apply*move-block*remove-old-clear
  (state <s> ^operator <o>)
  <o> ^clear <o>)
  (<<o> ^name move-block
   ^destination <db>)
  (<db> ^type block)
  =>
  (<s> ^clear <o> @))

sp (apply*move-block*remove-old-ontop
  (state <s> ^operator <o>)
  ^ontop <o>)
  (<<o> ^name move-block
   ^moving-block <db>
   ^destination <db>)
  (<db> ^top-block <mb>
   ^bottom-thing {<<db> ^cht}>))
  =>
  (<s> ^ontop <ontop> @))

sp (apply*move-block*add-new-ontop
  (state <s> ^operator <o>)
  (<<o> ^name move-block
   ^moving-block <mb>
   ^destination <db>)
  =>
  (<s> ^ontop <ontop> + @)
  (<ontop> ^top-block <mb>
   ^bottom-thing <db>))
```

NOTE: This illustration leaves out preference memory. Don't forget that productions create preferences.

Wait Operator: Operator Application

It is possible that an operator might have NO productions involved in operator application.

For example, the **wait** operator:

```
sp (default*top-ps*propose*wait
  (state <s> ^problem-space top-ps)
  =>
  (<<s> ^operator <o> <o> @)
  (<o> ^name wait))

sp (default*terminate*wait
  (state <s> ^operator <o>)
  (<<o> ^name wait)
  =>
  (<<s> ^operator <o> @))
```

This operator is proposed as unary worst, and has no application productions. The conditions for termination are just that it is the current operator.
Operators Directly or Indirectly Change the State

The simple blocks-world task does not interact with an external environment (not even a simulated external environment).

Soar programs that do not interact with an external environment make only *direct* changes to the state:

- Often, operator application productions make updates directly to the state.
- This is used mainly for simple programs or during development — not very realistic.

Indirect Changes to the State: Soar I/O

Soar programs that interact with an external environment make both *direct* and *indirect* changes to the state:

- Productions create preferences for an *output structure* (substructure of the state).
- The output structure is created in working memory.
- External code (e.g. C or Tcl) looks for this structure in working memory and reacts accordingly (e.g. executes a command to move a block with a robotic arm).
- External code creates/modifies an *input structure* in working memory (substructure of the state).
- Soar productions can match against an input structure.

*NOTE:* Soar programs that use I/O often also maintain an internal model of the changes expected to result from external actions.
operator Termination

An operator termination production creates a reconsider preference for the operator.

- A reconsider preference signals that the operator has completed and that Soar should reconsider all preferences for the operator of the state. (But remember that operator preferences are evaluated only at quiescence).

- It is possible that Soar may select the exact same operator again. (But the operator must first be terminated.)

sp (terminate^move-block
  (state <s> ^operator <o>
    ^ontop <ontop>)
  (<o> ^name move-block
    ^moving-block <thing1>
    ^destination <thing2>)
  (<ontop> ^top-block <thing1>
    ^bottom-thing <thing2>)
  -->
  (<s> ^operator <o> @))
Elaboration Productions

Productions that are not involved in operator proposal, comparison, application or termination are called *elaboration productions*.

- Elaboration productions make modifications to the state that are independent of the current operator.
- Elaboration productions make simple inferences about the state so that operators do not have to explicitly list all of their side effects.
- An inference is *retracted* when the situation it's based on is no longer true.

Retraction of Productions

As a general rule:

- Productions that create or apply operators do not have their actions retracted when the conditions no longer match.
- All other productions have their actions retracted when they no longer match.

*NOTE:* The two types of preferences are called O-support for operator-supported preferences and I-support for instantiation-supported preferences; the issue of whether a preference retracts or not is called persistence. We won't discuss persistence in detail until day 4.
Blocks Example: Elaboration

sp (elaborate*initial-state
(state <s> ^superstate nil)
-->
(<s> ^thing <a> + &
  ^thing <b> + &
  ^thing <c> + &
  ^thing <t> + &
  ^ontop <o-a> + &
  ^ontop <o-b> + &
  ^ontop <o-c> + &
  ^clear <a> + &
  ^clear <b> + &
  ^clear <c> + &
  ^clear <t> + &)
(<a> ^type block ^name A)
(<b> ^type block ^name B)
(<c> ^type block ^name C)
(<t> ^type table ^name table)
(<o-a> ^top-block <a> ^bottom-thing <t>)
(<o-b> ^top-block <b> ^bottom-thing <t>)
(<o-c> ^top-block <c> ^bottom-thing <t>)
(write (crlf) [Initial state has A, B, and C on the table.]))

This is the only elaboration production in the blocks world example.

NOTE: Although this is an elaboration production, it will never retract because its conditions will always be true.

When an Operator Cannot Be Selected or Applied

When Soar is unable to select a new operator during the decision cycle, it reaches an impasse. From slide 7:

3. Multiple operators are tied, but these tied operators are not also indifferent. This is called a tie impasse.

4. The preferences are contradictory and no operator can be selected as the winner. Two types:
   conflict impasse: better/best/worse/worst/indifferent preferences conflict
   constraint-failure impasse: require/prohibit preferences conflict

5. No operators are candidates. This is called a state no-change impasse.

6. A new operator can't be selected because the current operator has not been terminated. This is called an operator no-change impasse.
Exercise 2.3: Blocks world with two operators

Peter Hastings

April 30, 1996, 4:30 – 5:30

Soar/IFOR Tutorial

University of Michigan AI Lab

The Mini-Blocks Simulator

If you can't use the real world, try a simulation:

The simulated robot suction cup moves blocks, and makes changes to working memory, so the Soar program need not make direct changes to working memory to record the state of the world.
Language of interaction

Soar and the simulator talk to each other with a strictly defined interface, or language.

- **Output commands**
  Soar issues commands to the simulator by calling simulator functions. For example, `move-block 1 3`.

- **Input structure**
  The simulator places working memory elements in Soar, for example:
  - `thing` (block or table) indicates an object in the simulated world
  - `clear` (block or table) indicates that that object has space for another on top of it
  - `ontop` (top-block, bottom-thing) gives the relationships between objects in the world

Effects of External interaction

These aspects of the program are *not* changed:

- Operator Proposal
- Operator Termination
- Goal recognition

These aspects of the program *are* changed:

- Initialization (setting up the world): The simulator does it.
- Operator Application: The Soar program just issues commands to the simulator, and the simulator performs the operation and returns the changes to the state.
move-block.soar Thu Apr 25 15:57:32 1996 1

--- Mode: SDE ---

*******************************************************************************
## File: blocks-ext.soar
## Original author(s): John E. Laird <laird@eecs.umich.edu>
## Organization: University of Michigan AI Lab
## Last Modified By: Peter Hastings <peter@shrike.eecs.umich.edu>
## Last Modified On: 25 Apr 1996, 15:57:32
## Soar Version: 7

### Description: A new, simpler implementation of the blocks world
with just three blocks being moved at random.

### Notes:
- CBC, 6/27: Converted to Tcl syntax
- CBC, 6/27: Added extensive comments
- PMH, 7/3: Changed to graphical blocks world input style. The
  difficulty with this version is that when the gripper is over a
  block, the block does not show up on the input link as being
  clear.
- PMH, 7/12: Changed to a new simpler input format which is on the tostopstate
- PMH, 9/10: External interaction version

*******************************************************************************

# Set user-select mode to random so that Soar will choose randomly from among
# the suggested (indifferent) operators

user-select -random

*******************************************************************************

# Create the output-link
# The conditions establish that the top-level state and the io augmentation
# exist.
# The action creates the output-link as an augmentation of io.

sp (elaborate*state*output-link
 (state <> ~superstate nil
  "io "io")
  <- > output-link <out>))

*******************************************************************************

# Suggest MOVE-BLOCK operators
# This production proposes operators that move a block.
# The conditions establish that:
# 1. The block moved and the destination must be both be clear.
# 2. The block moved is different from the destination.
# 3. The block moved must not already be on the destination (i.e. it is
#    on something else).
# The actions:
# 1. create an acceptable preference for an operator.
# 2. create acceptable preferences for the substructure of the operator (its
#    name, its 'moving-block' and its 'destination').

sp (propose*move-block
 (state <> "thing <thing1>
  "thing (<thing2> <> <thing1>)
  "clear <thing1>
  "clear <thing2>
  "ontop <ontop>
  "ontop <ontop>
  "ontop <ontop>
  "top-block <thing1>
move-block.soar
Thu Apr 25 15:57:32 1996

******************************************************************************
# Reconsider (terminate) the MOVE-BLOCK operator
#
# This production signals that the operator has completed its actions
# The conditions establish that:
# 1. An operator has been selected for the current state
#    a. the operator is named move-block
#    b. the operator has a 'moving-block'
#    c. the operator has a 'destination'
# 2. The state has an 'ontop' relation
#    a. the ontop relation has a 'top-block' that is the same as the
#    'moving-block' of the operator
#    b. the ontop relation has a 'bottom-thing' that is the same as the
#    'destination' of the operator
# The actions:
# 1. create a reconsider preference for the operator

sp (terminate*move-block
    (state <> "operator <>
    "ontop <>ontop">
    <> "name move-block
    "moving-block <thing1>
    "destination <thing2>
    (ontop) "top-block <thing1>
    "bottom-thing <thing2>
    -->
    (<> "operator <> $))

******************************************************************************
# Detect that the goal has been achieved
#
# The conditions establish that:
# 1. The state has three ontop relations
#    a. a block named 1 is on a block named 2
#    b. a block named 2 is on a block named 3
#    c. a block named 3 is on a table named table
# The actions:
# 1. print a message for the user that the 1,2,3 tower has been built
# 2. halt Soar

sp (detect*goal
    (state <> "ontop <12>
    "ontop <23>
    "ontop <3T>
    (<12) "top-block <1>
    "bottom-thing <2>)
    (<23) "top-block <2>
    "bottom-thing <3>)
    (<3T) "top-block <1>
    "bottom-thing <T>
    (<1) "name 1 "type block"
    (<2) "name 2 "type block"
    (<3) "name 3 "type block"
    (<T) "name table "type table"
    -->
    (write (crlf) [Achieved 1, 2, 3])
    (halt))

### Magic incantation to make Soar put the blocks on the output
format-watch -object -add o (ifd (ifdef["v[destination]["ifdef[ "v[move-block.name]] to ifd
sf["v[destination.name]]))
Productions of the Internal and External programs

These are the productions of the internal Soar blocks program, and the external interaction program:

<table>
<thead>
<tr>
<th>Internal productions</th>
<th>External Productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>elaborate*initial-state</td>
<td>elaborate-state*output-link</td>
</tr>
<tr>
<td>propose*move-block</td>
<td>propose-move-block</td>
</tr>
<tr>
<td>apply<em>move-block</em>remove-old-ontop</td>
<td>apply*move-block</td>
</tr>
<tr>
<td>apply<em>move-block</em>add-new-clear</td>
<td>remove<em>move-block</em>command</td>
</tr>
<tr>
<td>apply<em>move-block</em>add-new-ontop</td>
<td>terminate*move-block</td>
</tr>
<tr>
<td>apply<em>move-block</em>remove-old-ontop</td>
<td>detect-goal</td>
</tr>
<tr>
<td>terminate*move-block</td>
<td></td>
</tr>
<tr>
<td>detect-goal</td>
<td></td>
</tr>
</tbody>
</table>

Running the simulator

To start the mini-blocks simulator, do the following steps:

- `cd ~/Common/ex2.3` to change to the right directory
- `soar` to start Soar. This brings up the simulator with the initial world.
- `cd ~/groupname` to change to your directory
- Load the original (move-blocks) productions using the button on the simulator.
- Run the program using the buttons on the simulator by whatever steps you would like, paying special attention to the production firings, and operator selections, applications, and terminations.
Your mission: Convert to pick-up and put-down operators

The previous model assumed a smart robot that knows what it means to move a block. So the Soar program invoked move-block commands, and the robot did it.

The new robot is not so clever. It must be told to pick-up a block, and then to put-down the block it is holding on another block.

Your mission is to create a program which controls this stupid robot by issuing pick-up and put-down commands until the tower is built. For each operator, there must be one or more proposal, application, and termination productions.

How to do it

File pick-put.soar in your group's directory has all of the required productions for this program plus some extras.

Read the descriptions of the productions (and look at them to understand them), and comment out the ones that you don't want in the program.

Then reload the world, clear the working memory, clear production memory, load your program, and run it.

Repeat this process until the program works like it should: picking up a block and putting it down on another block or the table until the goal is met.
3.1: Example Operators in TacAir-Soar

Randolph M. Jones

May 1, 1996, 9:00AM–9:30AM

Soar/IFOR Tutorial

University of Michigan AI Lab

Categories of Operators

This talk presents the different categories of operators that appear in the TacAir-Soar system, and includes examples of each.

There are three general types of operators to be found in the TacAir-Soar code:

- Standard, hierarchical operators
  - Primary tasks and sub-tasks for mission execution
  - Dependencies between standard operators are a strict hierarchy
- Shared operators
  - Tasks that are common to multiple higher-level tasks
Categories of Operators (cont.)

- Floating operators
  - Opportunistic
  - Depend on top state representation, but not on goal hierarchy

A Sample Hierarchy of Standard Operators

*Execute-Mission*

*Intercept*

*Employ-Weapons*

*Get-Missile-Lar*

*Cut-To-Ls*
Example Standard Operators

Employ-Weapons

Propose when:

- The supergoal is *Intercept*
- Contact with the primary threat is achieved
- Intercept geometry has been selected
- No higher priority tasks are being proposed

Subtasks: Achieve-proximity, Fpole, Get-Missile-Lar, Launch-missile, Select-Missile, Sort-Group, Support-Missile

Example Standard Operators (cont.)

Employ-Weapons

sp {intercept*suggest-proposal*employ-weapons
  (state <s> "problem-space.name intercept
  "group <group>
  "threat-agent.contact.achieved *yes*
  "intercept-geometry-selected *yes*)
  -{ (state <s> "operator <o> +)
    (o> "name << execute-tactic change-piece-of-sky
      bug-out evade blow-through
      select-primary-threat >>) }
  -->

Example Standard Operators (cont.)

Cut-To-Ls

Propose when:

- The supergoal is \textit{Get-Missile-Lar}
- The primary target is not "jinking"
- We are not employing a short-range weapon
- We are too close to the primary target’s flight path

Subtasks: Primitive maneuver actions

---

```lisp
(sp {get-missile-lar*suggest-proposal*cut-to-ls
 (state <s> "problem-space.name get-missile-lar
 \^missile <mt>
 \^agent.jinking <aj>
 \^lateral-separation <ls>
 \^target-aspect <ta>
 \^top-state.weapon.missile.parameter <mp>)
 (\<mp\> "type <mt>
 \^range <> short)
 (\<aj\> "value *yes*)
 (\<ta\> "achieved *no* "reason too-close)
 (\<ls\> "achieved *no* "reason too-low)
 -->
```
A Slightly More Complex Example

Push-Fire-Button
Propose when:

- The supergoal is Launch-Missile
- The selected weapon’s launch acceptability region (LAR) is achieved
- The weapons’ steering circle is achieved
- The selected weapon is radar guided
- The radar is in the correct mode for firing

OR ...

A Slightly More Complex Example (cont.)

- The supergoal is Launch-Missile
- The selected weapon’s launch acceptability region (LAR) is achieved
- The weapons’ steering circle is achieved
- The selected weapon is not radar guided

Subtasks: Primitive button-press action
A Slightly More Complex Example (cont.)

Push-Fire-Button proposals

sp {launch-missile*suggest-proposal*push-fire-button*normal-radar-weapon
(state <s> "problem-space.name launch-missile
  "steering-circle-achieved *yes*
  "missile <mt>
  "agent <a>
  "top-state <ts>)

# Compare the following four lines to the next slide
(<ts> "weapon.missile.parameter <mp>
  "radar.mode stt-pd)
(<mp> "type <mt>
  "guidance radar)
(<a> "weapon <aw>)
(<aw> "missile <am>)
(<am> "type <mt>
  "lar-achieved *yes*)

-->
A Slightly More Complex Example (cont.)

Push-Fire-Button application

sp {push-fire-button*apply
  (state <s> ~top-state <ts> ~operator <o>)
  (~o> ~name push-fire-button ~missile <mt>
      ~agent.weapon.missile <am> ~fire-missile *yes*)
  (~ts> ~radar.mode <rm> ~weapon.missile.parameter <mp>
      ~current-time.value <t>)
  (~mp> ~type <mt> ~name <mn> ~range <range> ~launch-to-eject <lte>)
  (~am> ~type <mt>)
-->
  (write (crlf) | Push button to launch | <mt> | ( | <mn> | |))
  (write (crlf) | [ | <range> | range missile].|)
  (~o> ~fire-missile *yes*
      ~missile-instance <ami>)
  (~am> ~instance <ami> + &)
  (~ami> ~waiting-to-clear *yes*
      ~launched-radar-mode <rm>
      ~time-to-clear (+ <lte> <t>))
}

An Example of a Shared Operator

Achieve-Proximity

Propose when:

- The supergoal is Intercept
- Contact with the primary threat is achieved
- Intercept geometry has been selected
- No higher priority tasks are being proposed

OR ...
An Example of a Shared Operator (cont.)

- The supergoal is *Employ-Weapons*
- No higher priority tasks are being proposed

*OR*

- The supergoal is *Get-Missile-Lar*
- The primary target is "jinking"
- We are not employing a short-range weapon
- We are not behind the primary target

*OR ...*

An Example of a Shared Operator (cont.)

- The supergoal is *Follow-Leader*
- We are very far away from our group partner

Subtasks: Primitive maneuver actions
Example Floating Operators

Identify-Contact

Propose when:

- We are executing a mission
- There is a blip on our radar
- The blip has not been assigned to a "mental agent"

Subtasks: Find or create mental agent corresponding to radar blip

Example Floating Operators (cont.)

Identify-Contact

sp {default*propose*identify-contact*radar
   (state<s> `top-state <ts>)
   ~(state <ss> `superstate <s>)
   (<ts> `mission.executing *yes*
       `radar <r>
       ~`agent.info.radar <ri>
       `parameter.radar-sense-delay <sd>)
   (<r> `blip <ri>
       ~`ignore-blip <ri>)
   (<ri> `status *normal*)
   ~(<s> `operator <o1> +)
   (<o1> `name << identify-by-call-sign identify-my-assignment
         identify-by-group-id >> )}
   -->
Example Floating Operators (cont.)

Activate-Command-Group

Propose when:

- We belong to a section command group (flight of two)
- The command group is not active
- We are instructed to activate the command group when contact is achieved
- We have achieved contact with the other member of the group

Subtasks: Mark the command group as activated

---

Example Floating Operators (cont.)

Activate-Command-Group

sp {any-ps*propose*activate-command-group*contact
 (state <s> ~top-state.command <c>)
 (<<c> ^group <sg>)
 (<<sg> ^type section
   ^joining *contact*
   ^active *no*
   ~mission.member.agent.contact <contact>)
 (<<contact> ^active *yes*
   ~achieved *yes*)

-->
3.2: Impasses and Subgoals in Soar

Peter Hastings
(words and music by Clare Bates Congdon)

May 1, 1996, 9:30 – 10:00

Repeat: When an Operator Cannot Be Selected

When Soar is unable to select a new operator during the decision cycle, it reaches an impasse.

3. Multiple operators are tied, but these tied operators are not also indifferent. This is called a tie impasse.

4. The preferences are contradictory and no operator can be selected as the winner. Two types:
   - conflict impasse: better/best/worse/worst/indifferent preferences conflict
   - constraint-failure impasse: require/prohibit preferences conflict

5. No operators are candidates. This is called a state no-change impasse.

6. A new operator can't be selected because the current operator has not been terminated. This is called an operator no-change impasse.
Impasses in Problem-Solving

- An impasse signifies that knowledge is not directly available; Soar must do problem solving to resolve the impasse.
- Resolving the impasse is cast as a subtask of the original task — this is called a subgoal.
- The subgoal is represented as a new state, sometimes called a substate.
- The state is linked to the substate through the superstate attribute. (Note that the superstate is not necessarily linked to the state.)
- Problem-solving progress can occur in the state and the substate in parallel. (That is, productions may fire.)
- However, once Soar is able to select an operator in the superstate, the impasse is resolved, and the subgoal is terminated (removed from working memory).
- An impasse can arise in a subgoal, leading to a stack of subgoals.

Simplified Illustration of a Subgoal Stack

- `nill` = state and operator objects
- `o` = other objects
- `o` = operator decisions that have not yet been made
- `+` = acceptable preferences for operators

This subgoal was created because Soar didn't know how to apply operator O2 in state S1.
No operator has been selected yet for S2.
Impasses in Problem-Solving

Example of an impasse:

- Two operators are suggested as acceptable, but neither is better than the other, and the two are not indifferent. (This is a tie impasse.)
  - Soar creates a subgoal to find out if one operator is better, or if they are mutually indifferent, or if one can be rejected.
  - In the subgoal, Soar can do a lookahead search: it tries out the operators “in its head” to see what might happen.
  - The subgoal is terminated when Soar is able to generate a preference that resolves the impasse.

**NOTE:** The creation of a subgoal is the only time that a new state is created in Soar. Each state can have at most one substate.

**NOTE:** Impasses can also occur for non-operator decisions. These are called attribute impasses; we'll discuss them later.

---

An Operator Tie Impasse

In the blocks world, 6 different move-block operators are initially proposed. If they are not made indifferent, then Soar does not know how to choose between them, and reaches an operator tie impasse.

![Operator Tie Impasse Diagram]

<table>
<thead>
<tr>
<th>Operator tie impasse</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1: move-block 1 to 2 +</td>
</tr>
<tr>
<td>O2: move-block 1 to 3 +</td>
</tr>
<tr>
<td>O3: move-block 2 to 1 +</td>
</tr>
<tr>
<td>O4: move-block 2 to 3 +</td>
</tr>
<tr>
<td>O5: move-block 3 to 1 +</td>
</tr>
<tr>
<td>O6: move-block 3 to 2 +</td>
</tr>
</tbody>
</table>

Subgoal
Lookahead Search: High level view

In the subgoal, Soar randomly picks an operator to evaluate and checks if it makes progress toward the goal. If not, it makes that operator undesirable by passing a worst preference to the superstate. This preference is called a result because it is put on the superstate by a production in the substate.

Resolution of Impasses

An impasse is resolved when the conditions that cause it no longer hold.

- An operator tie impasse, for example, is resolved when Soar can choose a unique operator.
- An operator no change impasse is resolved when Soar can terminate the current operator.
- A state no change impasse is resolved when an operator is proposed.
- And so on...

Usually an impasse is resolved by a result preference from the substate, but it can also be resolved by input from the external world.
What happens when an impasse is resolved

When an impasse is resolved:
- the subgoal (substate) is automatically removed (by Soar)
- all substructure not also attached to superstate is also removed

Operator Tie Impasse Resolved

If the subgoal finds an operator that makes progress toward the goal state, it posts a best preference on the superstate, which allows Soar to choose that operator, and it can thereby make progress, and the impasse is resolved.
**Justifications**

Justification corresponds to a temporary production instantiation:

- A justification serves to hold the result in working memory.
- The action of the justification: the result
- The conditions of the justification: determined via backtracing

The justification remains as long as its conditions are met and is then removed.

**Backtracing**

Trace back through production instantiations that fired to find the WME's that allowed the result to be created:

- The result was formed due to the presence of certain working memory elements.
- These, in turn, were formed due to the presence of other working memory elements.
- Keep working backwards until entire trace is grounded in the superstate: These WME’s form the conditions of the justification. (This could be a lot of conditions.)
Chunks

Learning is not currently used for Soar/IFOR

When learning is ON, chunks are created when subgoals are resolved.

- Chunks are variabliized versions of justifications (created via backtracing, etc.)
- A chunk is added to production memory and will fire when its conditions are matched.
- A chunk will not fire with the same set of WME's that created it.
- But it may fire immediately with a slightly different set of WME's.

Subgoal Stacks

The subgoal is structurally similar to top-level problem solving:

- Selection and application of operators
- Impasses may also occur in the subgoal, leading to additional subgoals
  - Subgoals form a "stack"
  - problem-solving activity can occur at multiple levels of the stack simultaneously — but will typically focus on the most recently created subgoal
- etc.

The substate is created with minimal structure; it is up to the Soar programmer to write productions that add the structure to substates. E.g.

- Might copy part of the structure from the superstate.
- Might use entirely different structure.
**Attribute impasses**

Soar can also reach an impasse for non-operator attributes; this is called an attribute impasse, and does not lead to a subgoal.

- Instead, an *impasse* object is created in working memory.
- Productions can monitor for attribute impasses, or they may be ignored.
- Often, an attribute impasse is a bug in the Soar program
  - Forgot to use parallel preferences for multi-attributes
  - Forgot to reject the old value when proposing a new value

So monitoring for attribute preferences is a good idea.

---

**Impasses in Working Memory**

When Soar reaches an impasse, it creates a new object in working memory to represent the impasse. Regardless of the type of impasse, there are several attributes that are created with this new object:

- **type** Either *state* or *impasse*, depending on whether it is an operator impasse or an attribute impasse.
- **impasse** Contains the impasse type: *tie*, *conflict*, *no-change*, or *constraint-failure*.
- **choices** Contains *multiple* (for tie and conflict impasses), *none* (for no-change impasses), or *constraint-failure* (for constraint-failure impasses).
- **superstate** Contains the identifier of the state in which the impasse arose (not for attribute impasses).

(continued on next slide...)
Impasses in Working Memory, continued.

attribute For multi-choice and constraint-failure impasses, this contains the attribute for which the values were competing. For no-change impasses, this contains the attribute of the last decision with a value (state or operator).

item For multi-choice and constraint-failure impasses, this contains all values involved in the tie, conflict, or constraint-failure. If the set of items that tie or conflict changes during the impasse, the architecture removes or adds the appropriate item augmentations without terminating the existing impasse.

quiescence States are the only objects with quiescence t, which is an explicit statement that quiescence (exhaustion of the elaboration cycle) was reached in the superstate. If problem solving in the subgoal is contingent on quiescence having been reached, the substate should test this flag.

---

Examples of Impasses in Working Memory

The following is an example of a substate that is created for a tie between three operators:

(s12 `type state `impasse tie `choices multiple `attribute operator `superstate s3 `item o9 o10 o11 `quiescence t)

The following is an example of a substate that is created for a no-change impasse to apply an operator as well as part of the superstate:

(s12 `type state `impasse no-change `choices none `attribute operator `superstate s3 `quiescence t)

(s3 `operator o2)

The following is an example of an attribute impasse that is created for a tie between two values of the color augmentation:

(i12 `type impasse `impasse tie `choices multiple `attribute color `superstate s3 `item o9 o10 `quiescence t)
Impasses in Production Memory

When there is an operator tie impasse, the production below gives the substate a problem-space, and names it choose-operator.

\[
\text{sp } \{ \text{blocks-world\_name-choose-problem-space} \\
\quad \text{(state } \langle s \rangle \text{ } \wedge \text{impasse tie} \\
\quad \quad \wedge \text{attribute operator} \\
\quad \quad \wedge \text{choices multiple}) \\
\quad -->
\quad \langle s > \text{ } \wedge \text{problem-space } \langle p \rangle \rangle \\
\quad \langle p > \text{ } \wedge \text{name choose-operator})\}
\]

Impasses in Production Memory, cont.

The following production checks for the choose-operator and for a tied item, and proposes an evaluate-operator operator for it.

\[
\text{sp } \{ \text{propose\_evaluate\_move\_block} \\
\quad \text{(state } \langle s \rangle \text{ } \wedge \text{problem-space}\_name \text{ choose-operator} \\
\quad \quad \wedge \text{item } \langle \text{super-op} \rangle \\
\quad \quad \quad \wedge \text{failure } \langle \text{super-op} \rangle \rangle \\
\quad -->
\quad \langle s > \wedge \text{operator } \langle o \rangle \text{ } + = \rangle \\
\quad \langle o > \wedge \text{name evaluate-operator} \\
\quad \quad \wedge \text{eval-operator } \langle \text{super-op} \rangle)\}
\]
3.3: Overview of Micro-TacAir-Soar

Randolph M. Jones

May 1, 1996, 10:15AM–10:30AM

Soar/IFOR Tutorial

University of Michigan AI Lab

Introduction

Micro-TacAir-Soar (μTAS) was developed as a teaching aid, and as a vehicle for presenting and explaining the general design of the larger TacAir-Soar system.

μTAS provides a bare-bones intelligent agent that functions in the tactical air domain. It contains many of the types of capabilities that are found in TacAir-Soar, but at a much more rudimentary level.

The system contains about 600 productions, implementing about 30 operators and 8 problem spaces.
Missions

The tutorial version of $\mu$TAS incorporates the knowledge necessary to fly only one mission: BarCAP.

Capabilities

In order to fulfill its mission, a $\mu$TAS agent has a number of core capabilities, which are a subset of the capabilities in TacAir-Soar.

Here is a general categorization of the capabilities in $\mu$TAS:

- Fly an FA-18 or MiG-29
- Employ AMRAAM, Sparrow, Sidewinder, Alamo, or Archer missiles
- Use radar and vision sensors
- Fly to a waypoint
- Fly a racetrack
- Keep track of one other friendly agent and one other (potential) enemy agent
Capabilities (cont.)

- Intercept and shoot down a single agent
- Fly in a section (flight of two) in a single formation
- Communicate two messages within section
- Switch section tactical lead if necessary

Implementation of Capabilities

The capabilities of a $\mu$TAS agent can also be presented in the form of the problem spaces, operators, and data structures that make up the agent's knowledge.
Problem Spaces and Operators

Top-PS

- Init-Agent
- Create-Agent
- Init-Plane
- Execute-Mission (BarCAP)
- Terminate-Agent

Execute-Mission

- Racetrack
- Intercept
- Follow-Leader

Problem Spaces and Operators (cont.)

Racetrack

- Fly-To-Racetrack
- Fly-Inbound-Leg
- Fly-Outbound-Leg

Intercept

- Achieve-Proximity
- Employ-Weapons
Problem Spaces and Operators (cont.)

Follow-Leader

- Achieve-Proximity
- Fly-To-Formation

Employ-Weapons

- Select-Missile
- Get-Missile-Lar
- Launch-Missile
- Support-Missile

Get-Missile-Lar

- Achieve-Proximity

Launch-Missile

- Get-Steering-Circle
- Lock-Radar-For-Missile
- Push-Fire-Button
Floating Operators

μTAS also includes the following "floating" operators, which can be proposed in many problem spaces:

- Adjust-Radar-Elevation
- Communicate-Intercept
- Create-Agent
- Focus-Attention
- Give-Up-Lead
- Group-Agent
- Identify-Contact
- Missile-Time-Is-Low
- My-Missile-Is-Stupid

Floating Operators (cont.)

- Record-Agent-Position
- Select-Primary-Threat
- Store-Position
Data Structures

To support all of this reasoning, μTAS maintains data structures for the following types of information:

- Agents (mental representation)
- Command groups
- Commit Criteria
- Communication
- Formations
- Missions
- Radar
- Target groups
- Vehicles

Data Structures

- Vision
- Waypoints
- Weapons
Exercise 3.1: Moving from Problem Spaces to Operators

Randolph M. Jones

May 1, 1996, 10:30AM-11:15AM

Soar/IFOR Tutorial

University of Michigan AI Lab

Laying Out a Problem Space

How do we design and implement the sub-tasks in a problem space?

Example from TacAir-Soar and Micro-TacAir-Soar:

- Employ-Weapons
Analyze the Super-Operator

What are the functions of the Employ-Weapons operator?

- Consider operator proposal and termination conditions
Analyze the Super-Operator

What are the functions of the Employ-Weapons operator?

• Considerations
  – What can we assume from the super-operator (Intercept)?
  – What should be the parameters of Employ-Weapons?
  – Under what conditions should we employ weapons?
  – Under what conditions would we be doing an intercept, but should not employ weapons?
  – When are we done employing weapons?

Analyze the Super-Operator

What are the functions of the Employ-Weapons operator?

• Proposal
  – We are intercepting a group
  – We have identified the primary threat in the group
  – We have contact with the primary threat (we know where the threat is)
  – The primary threat is known to be hostile
  – We are not about to reconsider the primary threat

• Termination
  – Proposal conditions are not true
Analyze the Super-Operator

Which knowledge structures are involved in employing weapons?

- Target groups
- Mental agents
  - ID
  - Contact
- Weapons (?)
Constructing the Problem Space

What do we want to achieve?
How can we achieve it?
What are the specific steps to take?
What local data structures support reasoning in this problem space?

Initial Data Structures

Common local data structures provide easy access to elements of the supergoal.
Initial Data Structures

Common local data structures provide easy access to elements of the supergoal.

- The name of the problem space
- A pointer to the parameters of the super-operator (the target agent in this case)

Choosing the Sub-Tasks

What do we want to achieve?
Choosing the Sub-Tasks

What do we want to achieve?

Whatever will terminate the super-operator.

- No more group to intercept
- No more primary threat in the group
- Others?
  - We no longer have contact with the primary threat
  - The primary threat is no longer hostile
  - We want to pick a new primary threat
Choosing the Sub-Tasks

What steps can be taken to employ weapons, in order to achieve the goals?

- Launch a missile
Choosing the Sub-Tasks

What steps can be taken to employ weapons, in order to achieve the goals?

- Achieve launch envelope for missile
- Launch a missile

- Select the best missile to use
- Achieve launch envelope for missile
- Launch a missile
Choosing the Sub-Tasks

What steps can be taken to employ weapons, in order to achieve the goals?

- Select the best missile to use
- Achieve launch envelope for missile
- Launch a missile
- Support missile

Specifying the Sub-Tasks

What are the knowledge dependencies between these tasks?

Select-Missile:
Get-Missile-Lar:
Launch-Missile:
Support-Missile:
Specifying the Sub-Tasks

Select-Missile:

- Must know which missile is currently best to use.
  - Use a local computation to make this decision and update it dynamically
- Must not be busy supporting a missile!

Specifying the Sub-Tasks

Get-Missile-Lar:

- A missile has been selected
- The target is not in the weapons envelope for the selected missile
## Specifying the Sub-Tasks

### Launch-Missile:

- A missile has been selected
- The target is in the weapons envelope for the selected missile
- We have radar contact with the target
- We are not already supporting a missile

### Support-Missile:

- A missile has been launched at some target
- The missile needs support
Specifying the Sub-Tasks

What do the sub-operators do?

- Select-Missile
  - Output command to cockpit simulator
- Get-Missile-Lar
  - Multiple possible maneuvers...a new problem space!
- Launch-Missile
  - Multiple activities...a new problem space!
- Support-Missile
  - Simple maneuvers to keep target illuminated on radar
Specifying the Sub-Tasks

What knowledge structures support the sub-operators?

- Select-Missile
  - Weapons, missiles
- Get-Missile-Lar
  - Weapons, missiles, agent
- Launch-Missile
  - Weapons, missiles, missile “instance”, radar
- Support-Missile
  - Weapons, missiles, missile instance
3.4: The Persistence of Preferences

Clare Bates Congdon

May 1, 1996, 11:15-11:30

Soar/IFOR Tutorial

University of Michigan AI Lab

When Is a Preference Removed?

A preference remains in preference memory until one of the following:

1. The I-supported production instantiation that created it retracts.

2. The preference is an acceptable preference (either I-supported or O-supported), and an O-supported reject preference is added to preference memory.

3. The preference is no longer linked to the subgoal stack.

**O-support** (operator support): provided by operator-application (and operator-creation) productions

**I-support** (instantiation support): provided by operator-proposal, operator-comparison, operator-termination, and elaboration productions.
How O-Support is Determined

O-support is given to operator-application and operator-creation production instantiations.

**Operator-application production**: tests the current operator (and state) and modifies the state. (Remember, current operator is also substructure of the state.)

**Operator-creation production**: tests the state and modifies an operator object — perhaps an operator that has been proposed, but not yet selected.

I-support is given to instantiations that do not meet the criteria for O-support.

---

Operator-Application Productions

Determination of operator-application instantiation:

- At least one condition tests the state (true of ALL productions).
- At least one condition must refer to an augmentation of the current operator.
- At least one action must create a preference for the state or substructure of the state.

It may not be possible to determine that an instantiation is an operator-application production just from looking at it:

- The production may create preferences for substructure of the state, but that substructure might not yet be linked to the state.
- Such preferences receive I-support until they are linked to the state.
Operator-Application Production in TacAir-Soar

sp {activate-division*apply*active*yes
  (state <s> ^operator <o>
    ^top-state.command.group <cg>)
  (<o> ^name activate-division
    ^group <cg>)
  (<cg> ^active *yes*)
  -->
  (<cg> ^active *yes*) }

---

Operator-Creation Productions

Determination of operator-creation instantiation:

- At least one condition tests the state (true of ALL productions).
- At least one action creates a preference to modify an operator (or substructure of an operator) that has been proposed for the state.

O-support must be given to operator-creation productions because operator substructure must be maintained while the state is changing during operator application.

It may not be possible to determine operator-creation until runtime.

In other words, it's not quite correct to say that a production is an O-supported production.
Operator-Creation Production in TacAir-Soar

\[
\text{sp \{} \text{any-ps*propose*activate-division} \\
(\text{state} \ <s> \ \text{^top-state.command.group} \ <cg>) \\
(<cg> \ \text{^joining-division} \ *yes*) \\
\text{-->} \\
(<s> \ \text{^operator} \ <o> \ + >, =) \\
(<o> \ \text{^name activate-division} \ \text{^group} \ <cg>) \ \text{)} \ \text{}\}
\]

If a single production does both operator-proposal and operator-creation functions, operator-proposal preferences will receive I-support, while operator-creation preferences receive O-support.

Calculation of Support for Results

*Recall*: a result is a preference created in a subgoal that resolves an impasse.

The subgoal is terminated when the result is created, so the result loses the support it was created with.

The result receives support from a temporary production instantiation, called a justification.

Whether a result received I-support or O-support is determined by the justification that supports the result.

- The support the result received in the substate is irrelevant.
- Support is based on the function the result plays in the superstate.
When to Use O-Support

If the conditions of an I-supported production test a WME that one hopes to change by the actions, the production will loop: it will repeatedly fire and retract on consecutive elaboration cycles.

O-supported productions should be used to make "sticky" changes to the state.

Use negated conditions cautiously with I-supported productions. In particular, do not use a negated condition to test for the absence of the same working memory element you would like your I-supported production to create; this would lead to an "infinite loop" in your Soar program, as Soar would repeatedly fire and retract the production.

An Example of an "I-Support Infinite Loop"

### NOTE: This is an INCORRECT production

sp {top-ps*elaborate*division*active
   (state <s> ^problem-space.name top-ps
     ^command.group <cg>)
   (<cg> ^joining-division *yes*
     ^active *yes*)
   -->
   (<cg> ^active *yes*)
}

### An Example of an "I-Support Infinite Loop"

```plaintext
### NOTE: This is an INCORRECT production
sp {top-ps*elaborate*division*active
   (state <s> ^problem-space.name top-ps
     ^command.group <cg>)
   (<cg> ^joining-division *yes*
     ^active *yes*)
   -->
   (<cg> ^active *yes*)
}
```
Persistence of Input/Output Structures

The substructure of the input-link will remain until the input function that created it removes it, and may act as cues for productions as long as it persists in working memory.

Note that since input functions may change the input link during any elaboration cycle, it is often prudent to copy important features from the input link to another attribute of the state.

Also note that this copying should be done with O-support.

The output link persists according to the production instantiations that created it.
Exercise 3.2: Moving from Operators to Productions

Randolph M. Jones

May 1, 1996, 11:30AM–12:30PM

Soar/IFOR Tutorial

University of Michigan AI Lab

Implementing an Operator

Now that we have decomposed part of the domain to the operator level. How can we realize these operators in an actual Soar implementation?

We will return to our example from Exercise 3.1 and look first at the Employ-Weapons operator.
Stages of Implementation

Recall the different stages involved in the use of an operator:

1. Propose the operator
2. Compare operators
3. Select an operator
4. Apply the operator
5. Terminate the operator

Step 3 is done by the Soar architecture, and we do not usually do step 2 in TacAir-Soar (that is, we usually propose only the correct operators, so there is nothing to compare). Thus, we only need to worry about writing productions for proposal (1), application (4), and termination (5).

Defining the Semantics of the Operator

The three stages of operator use tell us which functions we have to implement as Soar productions. Before doing that, we must decide the semantics of the operator:

- When should Employ-Weapons be used? (proposal)
- What should Employ-Weapons do? (application)
- When should Employ-Weapons go away so that a different operator can be selected? (termination)
Defining the Semantics of the Operator (cont.)

Recall that we specified these conditions in the previous exercise:

- Propose Employ-Weapons when
  - We are intercepting a group
  - We have identified the primary threat in the group
  - We have contact with the primary threat (we know where the threat is)
  - The primary threat is known to be hostile
  - We are not about to reconsider the primary threat
- Application will be done in a separate problem space (called Employ-Weapons, curiously enough)
- Terminate Employ-Weapons when
  - Any of the proposal conditions become untrue

Implementing the Operator

Now that we know the semantics of the operator, we can write the actual productions that Soar can execute.

Let us start with a production for proposing the operator.

Here is a template for the production:

```
sp { <production-name>
  (state <s> ... )
  <other conditions>
  -->
    <actions>
  }
```

We will start with a relatively simple step...what should the name of this operator be?
Operator Proposal

\[
\text{sp \{} \text{intercept}\ast \text{propose}\ast \text{employ-weapons} \\
\quad \text{(state <s> \ldots)} \\
\quad \text{<other conditions>} \\
\quad \text{-->} \\
\quad \text{<actions>} \text{\}}
\]

The next relatively simple step is to determine what the action side of the operator should do. Any guesses?

---

Operator Proposal

The action side simply needs to propose the operator, so the Soar decision mechanism can select it. The way you propose operators in Soar is to create an \textit{acceptable} preference for an operator augmentation on the state:

\[
\text{sp \{} \text{intercept}\ast \text{propose}\ast \text{employ-weapons} \\
\quad \text{(state <s> \ldots)} \\
\quad \text{<other conditions>} \\
\quad \text{-->} \\
\quad \quad \text{(<s> \sim \text{operator <o> +})} \\
\quad \quad \text{(<o> \sim \text{name employ-weapons})} \\
\text{\}}
\]

Now comes the more tricky part. How do we translate the semantics of the proposal conditions for Employ-Weapons into Soar syntax?
Knowledge Structure

Of course, the solution depends heavily on the form of the data structures we have chosen to represent the agent’s knowledge.

For now, assume the following structures:

- \(^{\text{threat-agent}}\) on the local state is the agent structure for the primary threat
- The local state also has a \(^{\text{problem-space}}\) with a \(^{\text{name}}\) attached to it (in this case, \textit{intercept})
- Agent structures have an \(^{\text{id}}\), which in turn has a \(^{\text{value}}\) of either \textit{friendly}, \textit{unknown}, or \textit{bandit}
- Agent structures have a \(^{\text{contact}}\), which in turn has a flag for \(^{\text{achieved}}\) with values of either \textit{*yes*} or \textit{*no*}

Operator Proposal

Here is (approximately) what the production looks like in the Micro-TacAir-Soar system:

```
sp {intercept*propose*employ-weapons
 ;## We are intercepting a group
 ;## (i.e., we are in the intercept problem space)
 (state <s> "problem-space.name intercept
 ;## We have identified the primary threat
 ;"threat-agent <a>>)
 ;## The primary threat is known to be hostile
 (<a> "id.value bandit
 ;## We have contact with the threat
 ;"contact.achieved *yes*"
 ;## We are not about to reconsider the primary threat
 ;## That is, the select-primary-threat operator is
 ;## not currently proposed
 -{(state <a> "operator <a> +)
 (<<o> "name select-primary-threat))}
 -->
 (<<s> "operator <a> +)
 (<<o> "name employ-weapons)
}
Operator Proposal

There is one more small detail involved in the proposal. We would actually like to remember which agent was the primary threat at the time of operator proposal, so the Employ-Weapons problem space can be sure of who to employ weapons against!

We can add this information to the operator’s representation to end up with the following proposal production:

```
sp {intercept*propose*employ-weapons
  (state <s> `problem-space.name intercept
   `threat-agent <a>)
  (<a> `id.value bandit
   `contact.achieved *yes*)
  -(state <s> `operator <o> +)
  (<o> `name select-primary-threat)}
-->
  (<s> `operator <o> +)
  (<o> `name employ-weapons
   `agent <a>)
}
```
Operator Termination

Employ-Weapons applications will occur in a sub-goal. So for now, the only other productions we need to write will terminate the operator.

Under what conditions should the operator terminate?

When any of the proposal conditions are untrue.

How can we implement these termination conditions in Soar productions?

Operator Termination, Solution 1

The proposal for Employ-Weapons consists of a conjunction of conditions.

The operator should be terminated when any one of these conditions becomes untrue (a disjunction of conditions).

One method is to write a termination production for each condition that can become untrue. But this can lead to messy code.
Operator Termination, Solution 2

Another method would be to create a single termination production that tests for the negation of all the conditions in the operator proposal:

```lisp
sp {employ-weapons*terminate
     (state <s> "operator <o>"
     (<<o> "name employ-weapons "agent <a>"
     -{( <s> "threat-agent <a>"
          (<<a> "id.value bandit"
            "contact.achieved *yes*"
          -{(state <s> "operator <o> +"
               (<<o> "name select-primary-threat})
          )
     -->
     (<<s> "operator <o> @)
} }

With this solution, we must remember to change the termination production any time we change the proposal production.
```

Operator Termination, Solution 3

When the proposal conditions are satisfied, an acceptable preference for the Employ-Weapons operator will appear in working memory.

To terminate, we can simply check whether the acceptable preference is still there:

```lisp
sp {employ-weapons*terminate
     (state <s> "operator <o>
     -"operator <o> +
     (<<o> "name employ-weapons"
     -->
     (<<s> "operator <o> @)
}
```
Solution 3 is very close to what we do in the TacAir-Soar system. However, to make things a little prettier, we introduce a new term, called \$suggest-proposal$, which we can test instead of the acceptable preference for the operator. Thus, for example, we end up with three productions in the Micro-TacAir-Soar code:

\begin{itemize}
  \item \texttt{intercept\#suggest-proposal\#employ-weapons}
    \begin{itemize}
      \item If the proposal conditions for employ-weapons are true, build a \$suggest-proposal$ for Employ-Weapons and the threat-agent on the lcoal state
    \end{itemize}
  \item \texttt{intercept\#propose\#employ-weapons}
    \begin{itemize}
      \item If there is a suggest-proposal for Employ-Weapons, propose the operator
    \end{itemize}
  \item \texttt{employ-weapons\#terminate}
    \begin{itemize}
      \item If the current operator is Employ-Weapons against a particular agent, and the corresponding suggest-proposal is gone, terminate the operator
    \end{itemize}
\end{itemize}
Implementing the Sub-Tasks of Employ-Weapons

We will look at just one of the sub-operators in Employ-Weapons: Select-Missile

We have the same questions for this operator as we had for Employ-Weapons:

- When should Select-Missile be used? (proposal)
- What should Select-Missile do? (application)
- When should Select-Missile go away so that a different operator can be selected? (termination)

Specifying the Semantics of Select-Missile

- When should Select-Missile be used? (proposal)
  - If the currently selected missile is not the same as the “best” missile
  * We will make the “best missile” decision into a knowledge structure, so we can implement it later
  - Not if we are currently supporting a missile!
- What should Select-Missile do? (application)
  - Issue an output command to select the desired missile
- When should Select-Missile go away so that a different operator can be selected? (termination)
  - When the selection has been made (the proposal conditions are no longer true)
Supporting Knowledge Structures

Here are the knowledge structures to use in order to write your productions:

- The local state has a `best-missile`, indicating the type of the best missile to use in the current situation
- The `top-state` has a `weapon` structure
- The `weapon` structure has a `type`, indicating the type of the currently selected missile
Select-Missile Application

If we want to print an informative message while selecting the missile, we might access some extra information from the missile's knowledge structure:

```lisp
sp {select-missile*apply
  (state <s> ~operator <o>)
  (~name select-missile
   ~missile <mt>)
  -->
  (write (crlf) |Push button to select | <mt> | (| <mn> |)|)
  (write (crlf) | [| <range> | range missile]|)
  (<o> ~select-missile <mt>)
}
```
Select-Missile Termination

This allows us to terminate when the missile selection is complete, but also if something goes wrong, for example, and the best-missile changes.

```
sp {select-missile*terminate
   (state <s> °operator <o>)
   (<o> °name select-missile
   °missile <mt>)
   -{(°s assume-choice <p>)
    (<p> °name select-missile
    °missile <mt>)}

   ->
   (°s °operator <o> @)
}
```

Creating the Supporting Knowledge Structure

Our final task is to implement the semantics for determining the best missile, so Select-Missile can be proposed at the appropriate time.

What should be the conditions for determining the best missile?
Creating the Supporting Knowledge Structure

What should be the conditions for determining the best missile?

- We have achieved the launch envelope for one of our missiles
- We have not achieved the launch envelope for any missile, so pick the longest range missile (the one that will achieve the launch envelope first)
- For multiple missiles, choose the fastest one

Specifying the Best Missile

What should be the precise conditions for selecting the best missile?
Specifying the Best Missile

What should be the precise conditions for selecting the best missile?

Choose an available missile whose launch envelope is achieved AND there is no faster available missile whose launch envelope is achieved.

OR

If there is no available missile whose launch envelope is achieved, choose the fastest of the longest range missiles.

Supporting Knowledge Structures for the Best Missile

In order to write the two productions for determining the best missile, assume the following existing knowledge structures:

- The local state specifies the ^target-agent
- The top-state has a ^weapon structure, which in turn has multiple ^missile structures
- A ^missile structure has a ^type, and a ^speed. It also has an ^available flag with a value of *yes* or *no*, and a ^lar-achieved flag for every agent in that missiles weapons envelope
Supporting Knowledge Structures for the Best Missile (cont.)

- A \text{missile} structure has multiple \text{lar} structures, each in turn having a \text{quarter} (for weapons quarter) and a \text{slant-range-high} (specifying maximum range).

- An agent structure has a \text{relative} structure, which in turn has a \text{weapon-quarter}, specifying which weapon-quarter the agent is in

\begin{verbatim}
Productions Determining the Best Missile

sp {employ-weapons*elaborate*state*best-missile*in-lar
   (state <s> ~problem-space.name employ-weapons
      ~target-agent <a>
      ~top-state.weapon <w>)
   ;## There is an available missile with a particular speed
   (<w> ~missile <m>)
   (<m> ~type <mt>
      ~available *yes*
      ~speed <sp>
      ;## The target-agent is in the missile's LAR
      ~lar-achieved <a>)
   ;## There is no available missile with LAR achieved and faster speed
   ;## than the chosen missile
   -((<w> ~missile <m2>)
    (<m2> ~available *yes*
     ~speed > <sp>
     ~lar-achieved <a>))
   -->
   (<s> ~best-missile <mt>)}
\end{verbatim}
Productions Determining the Best Missile

sp {employ-weapons*elaborate*state*best-missile*not-in-lar
  (state <s> "problem-space.name employ-weapons
       ^target-agent <a> ^top-state.weapon <w>)
  (<a> ^relative.weapon-quarter <wq>)
  ;## There is an available missile with a particular speed and high
  ;## slant-range for the agent's weapon quarter, but the LAR
  ;## is not achieved
  (<w> ^missile <m>)
  (<m> ^type <mt> ^available *yes* ^lar <l> ^speed <sp>
      ^lar-achieved <a>)
  ;## The target agent is not in the missile's LAR
  ;## Also look up the missile's range for that weapons quarter
  (<l> ^quarter <wq> ^slant-range-high <srh>)
  ;## There is no available missile with an achieved LAR
  -{(<w> ^missile <m2>)
    (<m2> ^available *yes* ^lar-achieved <a>)}

continued...

## There is no available missile with a greater range
-{(<w> ^missile <m3>)
  (<m3> ^available *yes* ^lar <l3>)
  (<l3> ^quarter <wq> ^slant-range-high > <srh>)}
  ;## There is no available missile with the same high slant range but
  ;## greater speed
  -{(<w> ^missile <m4>)
    (<m4> ^available *yes* ^lar <l4> ^speed > <sp>)
    (<l4> ^quarter <wq> ^slant-range-high <srh>)}
  -->
  (<s> ^best-missile <mt>)
}
3.5: Soar I/O

Clare Bates Congdon and Randolph M. Jones

May 1, 1996, 1:30pm-2:00pm

Soar/IFOR Tutorial

University of Michigan AI Lab

Repeat: I/O in Soar

Tcl or C input function creates WME's on the input structure

Tcl or C output function monitors WME's on the output structure

Production Memory

MATCH

FIRE

Working Memory

Preference Memory

DECIDE

input structure

output structure
I/O in Working Memory and the Decision Cycle

Input functions (= sensor input):
- create WME's (which then trigger productions)
- are called synchronously with Soar agents
  - Beginning of each decision cycle in TacAir-Soar
  - Beginning of each elaboration cycle in regular Soar

Output functions (= motor commands):
- respond to WME's (which are created via productions)
- are called synchronously with Soar agents
  - At the end of each decision cycle in TacAir-Soar
  - At the end of each elaboration cycle in regular Soar
- ....but only if changes to working memory warrant it.
Tradeoffs Between C and Tcl

Input and Output functions are written by the Soar programmer in C or Tcl.

I/O functions written in C:
- require recompilation of Soar.
- but generally run faster than Tcl.

I/O functions written in Tcl:
- do not require recompilation of Soar.
- are relatively quick to implement.
- but generally run slower than C.

Sometimes the appropriate approach is to do development with Tcl and later switch to C.

A Note on Using Soar I/O

We don’t have time to teach you how to write input and output functions in this tutorial.

Most likely, you won’t need to write your own anyway.

You may need to modify existing TacAir-Soar I/O code:
1. You’ll have our code to use as an example.
2. Ask us (Clare) for the Advanced Applications Manual.
3. Send us email to ask for help.
I/O in Working Memory

The \(^\text{io}\) attribute is added to the top-level state automatically by Soar (when the state is created), but has no substructure.

Input Functions (written in C or Tcl) create an *input link* in working memory as substructure of the \(^\text{io}\) attribute.

Output Functions (written in C or Tcl) look for specific *output links* in working memory as substructure of the \(^\text{io}\) attribute.

![Diagram of input and output links](image)

Naming Conventions

The input link can use any name, but the convention is to use simply \(^\text{input-link}\).

- If there are multiple types of sensors, they usually appear as separate substructures attached to the input link.

- The programmer must enforce that the same names are used in the input functions and in the Soar productions.

The output link can use any name, but the convention is to use simply \(^\text{output-link}\).

- If there is more than one type of effector for the agent to use, the commands to invoke each one can appear as a structure attached to the output link.

- The programmer must enforce that the same names are used in the output functions and in the Soar productions.
Example Blocks-World Input-Link

For example, the initial state of a blocks-world done with real blocks might have an input-link from a vision system, such as:

(S1 ^io I6)
(I6 ^input-link I9)
(I9 ^block B1)
(I9 ^block B2)
(I9 ^block B3)
(B1 ^x-location 1)
(B1 ^y-location 0)
(B1 ^color red)
(B2 ^x-location 2)
(B2 ^y-location 0)
(B2 ^color blue)
(B3 ^x-location 3)
(B3 ^y-location 0)
(B3 ^color yellow)

TacAir-Soar Input-Link

The amount of information that comes in on the input link in TacAir-Soar is much larger than the blocks-world task.

There are a number of different structures on the input link, corresponding to two different types of information sources.

- One group of information sources give exercise-related information, which would generally be pre-briefed to a pilot before flying the mission.

- The second group of information sources correspond more closely to what we would term sensors.

See http://ai.eecs.umich.edu/ifor/interface/input-link.html
TacAir-Soar Input-Link: Exercise-Related

These input features are generally static: The inputs are received at the start of the run, but do not change.

- battle-areas
- commit
- division
- event
- exercise
- flight-plan
- mission
- roe
- section
- waypoints

TacAir-Soar Input-Link: Sensors

These inputs are generally dynamic, and will change frequently.

- ccip
- detonation
- pgm-detonation
- radar-blip
- radar-warning
- radio
- signal
- speaker
- terrain
- vehicle
- visual-object
- waypoint-computer
TacAir-Soar Input-Link: Sensors, Detonation

Here is an example of the information the comes in attached to the `io.input-link.detonation` structure. (see http://ai.eecs.umich.edu/ifor/interface/detonation.html)

`^munition` The type of munition (a string) `mk84`
`^result` A "DIS result" `vehicle-hit`
`^time` The time on the agent’s stopwatch at which the detonation occurred (in seconds) `1348`
`^x coordinate of detonation` `35417`
`^y coordinate of detonation` `2512`
`^z coordinate of detonation` `0`

x, y, and z are in meters, which are ModSAF coordinates.

Example Output in Working Memory

For example, the initial state of a blocks-world done with real blocks might have an output-link to send commands to a program that controls a robotic arm, such as:

(S1 `^io I6`)
(I6 `^output-link O3`)
(O3 `^name move-block`)
(O3 `^moving-block B1`)
(O3 `^x-destination 2`)
(O3 `^y-destination 1`)
(B1 `^x-location 1`)
(B1 `^y-location 0`)
TacAir-Soar Output Structures

A number of different types of commands can be put on the output link in TacAir-Soar. The output commands can be divided into seven categories (see http://ai.eecs.umich.edu/ifor/interface/output-link.html):

- CCIP
- Ordnance
- Radar
- Radio
- Terrain Analysis
- Vehicle
- Waypoint Computer

TacAir-Soar Output Structures

The following are the output augmentations that fall into the "Radar" category. When these structure are attached to the ^io.output-link, it causes corresponding radar manipulation functions to be sent to the ModSAF simulator (see http://ai.eecs.umich.edu/ifor/interface/radar-control.html).

- ^desired-radar-mode
- ^desired-radar-range-minimum
- ^desired-radar-azimuth
- ^desired-radar-elevation
- ^break-lock
- ^desired-acquisition-mode
- ^desired-acquisition-stt-mode
- ^desired-acquisition-target
Input Links in Production Memory

Productions involved in input will test for specific attributes and values of an input-link. For example, a production that responds to the vision input for the blocks task might look like this:

```lisp
sp {blocks-world-get-input-block
  (state <s> ^io <io> ^operator <o>)
  (<o> ^name move-block)
  (<io> ^input-link <in>)
  (<in> ^block <ib>)
  (<ib> ^x-location <x> ^y-location <y>)
  -->
  (<s> ^block <b> +)
  (<b> ^x-location <x> + ^y-location <y> +))
}
```

This production is done with O-support so that the copied information will not retract.

Input Links in Production Memory (TacAir-Soar)

Here is an example of a production from TacAir-Soar, checking for a radar blip on the input link, and copying a pointer to the information to a radar structure on the top state (if it is an airborne blip):

```lisp
sp {top-ps-elaborate-radar*blip
  (state <s> ^problem-space.name top-ps
  ^radar <r>
  ^parameter.airborne <arb>
  ^io.input-link.radar-blip <rb>)
  ;## Ignore non-flyers
  (<rb> ^object-type <arb>)
  -->
  (<r> ^blip <rb> + &)
}
```
I-Support or O-Support

When you copy information from the input link, when should you give it I-support and when should you give it O-support?

Using I-Support from the Input Link

I-support can be considered a form of "reactive" support. If you want the agent's internal representation to change as fast as the world changes, you should use I-support.

In the previous example from TacAir-Soar, we want to create a structure for the radar blip only as long as the blip actually appears on the radar (as signified by its presence on the input link). If the blip goes away or a new blip appears, we want to retract or build structures immediately in response to the changes.
Using O-Support from the Input Link

O-support is what gives an agent a short-term memory. If you want to remember that something was on the input link even after it disappears, you should use O-support.

In the previous example from the blocks world, we wanted to record and remember the initial location of the block, even if the position would change later.

Output Links in Production Memory

Productions involved in output will create preferences for specific attributes and values of an output link. For example, a production that creates the output-link for the blocks task might look like this:

```prolog
sp {blocks-world*apply*move-block*send-output-command
  (state <s> ^operator <o> ^io <io>)
  (<o> ^name move-block ^moving-block <b1>
    ^destination <b2>)
  (<b1> ^x-location <x1> ^y-location <y1>)
  (<b2> ^x-location <x2> ^y-location <y2>)
  (<io> ^output-link <out>)
-->
  (<out> ^name move-block ^moving-block <b1>
    ^x-destination <x2> ^y-destination (+ <y2> 1))}
```
TacAir-Soar Output Links

Here is a production from TacAir-Soar that creates a preferences for a \texttt{^desired-radar-elevation} value on the top link. This production invokes the simulated radar steering mechanism to keep the radar pointed in the right place even as the pitch of the airplane changes (e.g., during a climb or dive).

\begin{verbatim}
sp {top-ps*elaborate*output*desired-radar-elevation
  (state <s> \texttt{^problem-space.name top-ps}
   \texttt{^io.output-link <ol>}
   \texttt{^position.pitch.value <pitch>}
   \texttt{^radar.desired-radar-elevation <v>})
  -->
  (<ol> \texttt{^desired-radar-elevation (- <v> <pitch>))
 }
\end{verbatim}

Note that this production creates an acceptable preference for the \texttt{^desired-radar-elevation} attribute, but does not reject the old value. This is because it is not an O-supported production; the old value will automatically be retracted at the same time that the new value is made acceptable.
Exercise 3.3: Debugging a Soar Program

Randolph M. Jones and Clare Bates Congdon

May 1, 1996, 2:00pm-3:00pm

Soar/IFOR Tutorial

University of Michigan AI Lab

<table>
<thead>
<tr>
<th>function</th>
<th>summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>load productions</td>
</tr>
<tr>
<td>run</td>
<td>run Soar (run decisions)</td>
</tr>
<tr>
<td>exit, quit</td>
<td>quit Soar</td>
</tr>
<tr>
<td>help</td>
<td>get help on Soar commands</td>
</tr>
<tr>
<td>excise</td>
<td>remove productions</td>
</tr>
<tr>
<td>init-soar</td>
<td>clear working memory</td>
</tr>
<tr>
<td>log</td>
<td>save Soar run to a file</td>
</tr>
<tr>
<td>print</td>
<td>print productions and WME's</td>
</tr>
<tr>
<td>wmes</td>
<td>print WME's (alternate command)</td>
</tr>
<tr>
<td>preferences</td>
<td>print preferences</td>
</tr>
<tr>
<td>matches</td>
<td>look at matchset and partial matches</td>
</tr>
<tr>
<td>watch</td>
<td>control how much info is printed as Soar runs</td>
</tr>
<tr>
<td>pwatch</td>
<td>watch specific productions fire and retract</td>
</tr>
</tbody>
</table>
Bugs and Soar

Debugging a Soar program involves many of the same types of techniques and problem-solving skills as debugging a computer program in some other language.

On the other hand, the Soar language is quite different from procedural programming languages, so some of the debugging techniques to use are very different.

Standard Debugging Techniques

As with any other programming language, Soar systems should be designed in such a way as to make debugging as easy as possible. Here are some of the debugging techniques that most computer programmers should be familiar with:

- Try to avoid bugs in the first place:
  - Use good, simple, programming style
  - Minimize complexity and “hacks” as much as possible
- Try to trace through code at a logic level
- Try to trace through code at a code level
- Make liberal use of the write command:
  - On action side of existing productions
  - In separate “monitor” productions
Soar-Specific Debugging Utilities

Soar provides some facilities for debugging that you may not find in other programming environments.

Because Soar is in some sense an interpreted programming language, you can set options to watch what is happening while Soar is processing (you should have already seen these commands while getting familiar with running Soar).

When you are looking for bugs, watching for certain production firings or working memory changes can be invaluable.

Four Very Useful Soar Commands

Besides using commands and techniques for watching a Soar system as it runs, there are four particularly useful commands that allow you to inspect various parts of the system’s memory, in order to attempt to determine what is going wrong:

print — Allows you to look at the contents of working memory, or at a production’s definition.

preferences — Allows you to look at the contents of preference memory.

matches — Allows you to see which productions have matched, or why a production is not matching when you think it should.

pf — Provides a fast way to find productions that may be causing problems.
Start the Exercise!

In order to get a feel for how you might debug a simple Soar program, we will try to find and fix a number of bugs in a Soar program you are provided with.

First, go into your group’s directory (e.g., cd Group2)

Next, start up Soar by typing soar7

Finally, load in the productions for the “buggy” program: source buggy-blocks.soar

What is the first thing you notice?

---

Load-time Errors and Warnings

When loading the file, you should have seen one error message and one warning message. These are usually the easiest kinds of bugs to fix, because the Soar system tells you where they are. Most bugs will prove harder to find than that.

You should treat the warning as an error. Even though Soar will allow the production to be loaded, a warning means at least that you are not using “good” Soar programming style, and at worst means you have made some kind of mistake.

The messages tell you which productions are in error.

Try to find the errors and fix them.

Use emacs or vi (in a separate window) to edit the buggy-blocks.soar file.
Fixing the Load-Time Errors

When blocks-world*elaborate*initial-state is loaded, the error message indicates that there is a variable on the right-hand side of the production that does not link up with anything on the left-hand side of the production.

If you look carefully, you will find that the variable is <clear-X>. This variable is a typographical error, and should be changed to <clear-Y>.

When blocks-world*select*move-block*indifferent is loaded, the warning message indicates that there is a variable on the left-hand side of the production that does not link up with the state variable (<s>) on the left-hand side of the production. In this case, the variable is <a> and should be changed to <o>.

Try Again, No Load Errors

Save the file you have edited, and reload it into Soar by typing

```
source buggy-blocks.soar
```

If you edited the file properly, there should be no load errors this time.

Now let's run the system, and see if it does what it is supposed to do.

Type run d at the Soar prompt (to run for one decision) and see what happens.
Finding Run-Time Errors: State No-Change

The system starts okay, and says it is firing the production that builds the initial blocks-world state. However, you then see something new, called a state no-change impasse.

When you are an advanced Soar programmer, there may actually be times when you want to see state no-change impasses (because they provide opportunities for certain types of learning). However, in this example (as well as in TacAir-Soar) a state no-change is always an indicator of some kind of bug in the code.

A state no-change means that no operator was proposed, so the system does not know what action it should take next.

How can we figure out what this means in terms of how our program may be broken?

Debugging the State No-Change

From our previous experience with this program, we know what is supposed to happen here: a collection of move-block operator instantiations are supposed to be proposed, with one of them getting selected. Apparently, however, this is not happening.

A good first step to take is to check whether the things that you think should be proposed are in fact being proposed. You can do this by typing preferences s1 operator. This will list all the preferences for the operator slot on state S1. (This is a particularly useful thing to do in a domain with many operators, where you think the wrong ones are being proposed and you want to find out why.)
Debugging the State No-Change (cont.)

In this case, the preferences command shows us that there are no preferences for the operator, which means that the production that should have created the preference did not fire for some reason.

*How can we find out which production should be firing in this situation?*

---

Using pf to Find a Production

One way to find the production would be to look in the buggy-blocks.soar file and search for the operator proposal. However, you can also use Soar’s pf command (which stands for Production Find).

To find an operator proposal, we want to find all productions that create acceptable preferences for operators on their right-hand sides.

*Try to figure out what you should type to do this.*

Use help pf if necessary.
Using pf to Find a Production

To find all productions that propose operators, type:

```
pf -rhs {(<s> ^operator <o> +)}
```

This should tell us that `blocks-world*propose*move-block` is the offending production.

*How can we figure out why this production is not firing?*

Using Matches to Find Production Errors

The `matches` command tells you if a production matches, how many times it matches (i.e., the number of instantiations), or why it does not match. Type

```
matches blocks-world*propose*move-block
```

to find out why the production is not matching.
Using Matches to Find Production Errors

The matches command prints the production's conditions in a particular order (Soar's matcher tries to find the most efficient order for doing matching, but you can consider the ordering essentially random).

To the left of each condition is a number telling how many instantiations the matcher has found for the production up to that point. The >>>> indicates the first point at which the matcher can find no instantiations to match the production.

In this case, this points us right to the offending condition of the production. This is another typographical error, where black should be changed to block.

Make that change in your copy of the file now.

Using Matches to Find Production Errors (cont.)

Note that Soar picks only one of many possible orderings on the conditions in attempting to match the production. Thus, the output of the matches command does not alway point you exactly at the condition you want to find, like it did in this case. Sometimes you still have to do a little work to discover which condition is really causing the production not to match.
Run the System Again

Type `init-soar` to reset the system. Then reload the new productions. Now we will try running the system again.

Type `run d` to start the system again.

This time the system has successfully selected an operator. So far so good. Now let us see if we can keep going without any errors.

Type `run d` one more time.

Soar has reached another impasse. This time it is an `operator no-change impasse`. Once again, in some Soar systems this would not necessarily be a signal that there is a bug, but in this system it is.

What kind of error would cause an `operator no-change impasse`?

Errors That Lead to Operator No-Change

An operator no-change impasse indicates that the selected operator did not terminate. This in turn means that the termination production did not fire.

This could mean that the termination production is incorrectly written.

It could also mean that something has gone wrong during operator application (e.g., one of the operator application productions is wrong), so the termination production's conditions never become true.

What should we do to find the source of the impasse?
Checking the Termination Production

We can do the same kinds of things we did for the previous bug. First, find the termination production by typing:

`pf -rhs {(<s> ^operator <o> @)}`

Then find out why this production is not matching by typing:

`matches blocks-world*terminate*move-block`

Look carefully at the output of the matches command, and try to figure out why it is not firing.

Operator No-Changes Can Mean Many Things

It turns out in this case that there is actually nothing wrong with the termination production. Something else must be preventing it from firing.

The matches command indicates that there is something amiss with the ^ontop structure on the state. There are in fact two ^ontop structures that the production can match, but neither has the correct substructure.

Let us verify this by actually inspecting working memory.

Type `print -depth 2 s1` to look at the top state, and examine the ^ontop structures.

The termination production is looking for an ^ontop structure whose components match the blocks on the current operator. We can find out what the current operator expects by also printing that.
Searching For the Proper Match (cont.)

This series of commands should show you things in working memory that look something like the following:

(S1 ^block C1 ^block A1 ^block B1 ^block T1
 ^clear C3 ^clear C4 ^clear C6 ^io I1
 ^ontop C2 ^ontop B2 ^operator O2 ^operator O3 +
 ^problem-space blocks ^superstate nil
 ^type state)

(C2 ^bottom-block T1 ^top-block C1)

(B2 ^bottom-block T1 ^top-block B1)

(O2 ^destination C1 ^moving-block A1
 ^name move-block)

Searching For the Proper Match (cont.)

The termination production would match against an ^ontop structure that has ^top-block A1 and ^bottom-block C1 attached to it.

However, no such structure exists. We are fairly certain that the termination production is correct. Another possibility is that there is an application that should be creating a new ^ontop structure, with the proper substructure.

Try to figure out how to find such a production and determine why it is not firing.
Checking for an Incorrect Application

First, try to find a production with the pattern we are looking for:

```
pf -rhs {(<s> ^ontop <o>)}
```

There are two such productions. One is the state initialization production, which we have already fixed once (but you can check again to make sure it is okay).

More likely, the problem lies with the other production:

```
blocks-world*apply*move-block*add-new-ontop
```

*How can we figure out if something is wrong with this production?*

---

Checking for an Incorrect Application

Typing `matches` blocks-world*apply*move-block*add-new-ontop indicates that there is only one condition preventing the production from firing.

*What (if anything) needs to be fixed in this production?*
Fixing the Incorrect Application

The moving-block condition should use the variable <block2> rather than <block1>.

Make this change to your program file. Then reinitialize everything, reload, and try running the program again.

This time everything should work fine.

Bonus Exercises

Look at the complete and correct blocks-world code.

Can you think of some other types of bugs that might arise when writing a program like this?

What would you do to fix them?

Can you think of any bugs that could only be found using techniques other than the ones we have discussed so far?

Can you think of some really insidious bugs that would be very difficult to track down?
Other Common Sources of Soar Bugs

The following is not meant to be an exhaustive list of problems; it's just what we could think of as we're putting these slides together.

1. Productions that never fire:
   set watch level to 3 or higher, and use `matches` and/or `firing-counts` to check

2. Productions that fire too often:
   set watch level to 3 or higher to find out when, and `matches -wmes <prod-name>` to find out why

3. Negated conditions cause many problems
   • Only because most of us aren’t so good with the logic
   • …especially with negated conjunctions
   • …or attribute-path notation
   Go slowly when writing negations.

Other Common Sources of Soar Bugs

1. Attribute-path notation can cause problems for beginners
   • Better to avoid it when writing your own productions
   • …but if you must, `print` the production after you load it and check that it expands to what you intend.

2. I-support and O-support issues cause many problems:
   • Could be that a production should have been O-supported, but wasn’t (retracts when you would rather it didn’t).
   • Could be that a production should be I-supported, but one of the conditions that support it retract prematurely (maybe `that` preference should have had O-support (i.e., the bug is further back).

3. Forgetting to reject an value for an attribute when you propose a new one.
A Few Tclisms...

Tcl provides command completion; you can type a unique string (different from any other Soar command) instead of the whole command.

The Tcl interpreter hides some errors; you have to inspect the Tcl variable "errorInfo" to see the last error:

`soar> set errorInfo`

If you make a typo when you enter that, your new last error will be the typo, and you won't know what the problem was. (It may make sense to alias "set errorInfo" to something easier to type accurately.)

Remember that Tcl is case-sensitive.
SDE: Soar Development Environment

- SDE is a special mode within Emacs that provides special functions for development (and running) Soar programs.
- Current SDE only supports functions for helping in development.
  - Justifying Productions (meta-Q).
  - Balancing Parentheses.
  - Move to end of production, etc.
  - Templates.
SDE Template Magic

- Define templates for adding new operators and new goals (operators with subgoals).
- Automatically creates files, links in for loading and documentation, and creates templates of necessary productions.
- Abracadabra! (^C ^T)
  - User fills in the blanks for all parameters.
  - Parameters are preloaded with old values.
- Presto! (^C ^C)
  - Completed productions and files are generated.

SDE Templates

<table>
<thead>
<tr>
<th>Template</th>
</tr>
</thead>
</table>
| `sp ([OP]*apply
  (state <s> ^operator <o>
   [$])
  (<o> ^name [OP])
  -->
  (<s> ))` |

<table>
<thead>
<tr>
<th>User Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP: Turn-Left</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Buffer Contents</th>
</tr>
</thead>
</table>
| `sp {turn-left*apply
  (state <s> ^operator <o>
   )
  (<o> ^name turn-left)
  -->
  (<s> ))` |
SDE Templates

super-op.soar
super-op.html
super-op
load.soar
state.html
elaborations.soar
operator1.soar
operator1.html

super-op.soar
super-op.html +
super-op
load.soar +
state.html
operator1.soar
operator1.html
operator2.soar +
operator2.html +
operator2
load.soar +
elaborations.soar +
state.html +

super-op.soar
cd super-op
load load.soar
cd ..

SDE Templates

User Input
PS: implement-super-op
OP: operator2

Modifications, New Files and Directories
super-op.html +
super-op
load.soar +
operator2.soar +
operator2.html +
operator2 +
load.soar +
elaborations.soar +
state.html +
Exercise 3.4: Subgoaling in the Blocks World

Peter Hastings

May 1, 1996, 3:45 – 5:30

Soar/IFOR Tutorial

University of Michigan AI Lab

Back to the blocks world

Now we turn back to the blocks world for a programming exercise: to turn the previous example into a subgoaling program.

The goals of this exercise are to:

- enforce the fundamentals of subgoaling
- teach the basics of writing productions:
  - matching and linking on the left hand side
  - creating preferences on the right hand side
- create another Soar program
Recap: Previous blocks world task

- Used the simulator to show the actions, and to describe the state of the world.
- Language of Interaction:
  - Pick-up and Put-down commands are issued to the simulator
  - Simulator puts thing, clear, ontop, and holding attributes on the state.
- We chose Proposal, Application, and Termination productions for each of the new operators.

New Task: Hierarchical Operators

The same as the previous task except:

- Use a hierarchy of operators: At the top level, the move-block operator is selected. This is implemented in a subgoal using pick-up and put-down operators.
- You write the program.
  - Look at the previous programs.
  - Look at the comments.
  - Feel free to copy, but it's best if you type it in.
  - Switch coders so both get the experience of programming Soar.
Hierarchical Operators: How to do it

- There is a new `pick-put.soar` file in your directory.
- The old one is included in the handouts.
- Load `pick-put`, read the comments, and write the code. Use the original `move-block` and `pick-put` files to guide you.
- When the productions have been written, start the simulator:
  - `cd ~/Common/ex2.3`
  - `soar`
  - `cd ~/groupname`
- Load your program, and run it, and debug it.
- Ask lots of questions.
- Try out different Soar commands.
- Good luck!!!
Suggest pick-up operators

This production proposes operators that pick up a block. The conditions establish that:
1. The thing to be picked up must be clear.
2. The gripper is empty.
3. The thing moved must have type block.
The actions:
1. create an acceptable preference for an operator.
2. create acceptable preferences for the substructure of the operator (its name, and its 'moving-block').

```sp
(propose-pick-up
  (state <= "thing <thing1>
    "clear"
    "holding nothing")
  ("thing1" "type block")
  (<= "operator <= +")
  ("name pick-up
    "moving-block <thing1>"))
)
```

Apply a PICK-UP operator

This production is part of the application of a PICK-UP operator. The conditions establish that:
1. An operator has been selected for the current state
   a. the operator is named pick-up
   b. the operator has moving-block BLOCK1
2. BLOCK1's name is NAME1
3. There is an output-link
   The actions:
   1. Put the pick-up command and the name of the block on the output-link.

```sp
(apply-pick-up
  (state <= "operator <=
    "io-output-link <out>)
  ("name pick-up
    "moving-block <block1>"
    "block1" "name <name1>
    "arg1 <name1>"
    ("out" "command pick-up
      "arg1 <name1>"))
)
```

Reconsider (terminate) the PICK-UP operator

This production signals that the operator has completed its actions
The conditions establish that:
1. An operator has been selected for the current state
   a. the operator is named PICK-UP
   b. the operator is moving block THING1
2. The gripper is holding THING1 as indicated by the holding attribute on the top-state.
The actions:
1. create a reconsider preference for the operator

```sp
(terminate-pick-up
  (state <= "operator <=
    "holding <thing1>
    "name pick-up
    "moving-block <thing1>"
    ("<thing1" "name pick-up
      "moving-block <thing1>"))
)
```
The state has an 'ontop' relation
a. the onttop relation has a 'top-block' that is the same as the
   'moving-block' of the operator
b. the onttop relation has a 'bottom-thing' that is the same as the
   'destination' of the operator
The actions:
1. create a reconsider preference for the operator

sp (terminate*put-down
 (state <> "operator <>
   "onttop <onttop>
 (<> "name put-down
   "moving-block <thing1>
   "destination <thing2>

top-block <thing1>
  "bottom-thing <> <thing2>)
-->
(<s> "operator <> <>)
(<> "name put-down
  "moving-block <thing1>
  "destination <thing2>
))

Apply a PUT-DOWN operator
The conditions establish that:
1. An operator has been selected for the current state
a. the operator is named put-down
b. the operator has moving-block BLOCK1
2. BLOCK1's name is NAME1
3. There is an output-link
The actions:
1. Put the pick-up command and the name of the block on the
   output-link.

sp (apply*put-down
 (state <> "operator <>
  "output-link <out>
 (<> "name put-down
   "destination <block1>
 (block1) "name <name1>
   <=>
 (out) "command put-down
   "arg1 <name1>
)

Reconsider (terminate) the PUT-DOWN operator
This production signals that the operator has completed its actions
The conditions establish that:
1. An operator has been selected for the current state
a. the operator is named PUT-DOWN
b. the operator is moving block THING1 to THING2

Detect that the goal has been achieved
The conditions establish that:
1. The state has three onttop relations
a. a block named 1 is on a block named 2
b. a block named 2 is on a block named 3
c. a block named 3 is on a table named table
The actions:
1. print a message for the user that the 1,2,3 tower has been built
2. halt Soar

sp (detect*goal
 (state <> "onttop <12>
   "onttop <23>
   "onttop <3T>
 (<12) "top-block <1>
   "bottom-thing <2>
 (<23) "top-block <2>
   "bottom-thing <3>
 (<3T) "top-block <3>
   "bottom-thing <T>
 (<1) "name 1 "type block"
 (<2) "name 2 "type block"
 (<3) "name 3 "type block"
 (<T) "name table "type table"
 <=>
 (write {crlf} {Achieved 1, 2, 3})
 (halt))

Magic incantation to make Soar put the blocks on the output
format-watch -object -add o (%id %ifdef[v[name]]%ifdef[ %moving-block.name]%ifdef[ o n %v[destination.name]])