Notes for Meeting 4

Symbolic Patterns and Pattern Matching

A Review of Symbol Structures

Newell and Simon (1976) introduce some key concepts that they claim underlie intelligent behavior:

- Symbols are physical patterns that remain stable unless modified.

- Symbol structures (expressions) are organized sets of symbols.

- A physical symbol system includes processes for creating, modifying, copying, and destroying symbol structures that let it:

  - Maintain structures that designate other objects or processes.

  - Interpret expressions that designate such processes.

List structures, like those supported by Lisp, have become a standard means to encode symbols and symbol structures.

However, functional or procedural schemes are not the only approach to interpreting such structures.

Symbolic Patterns and Pattern Matching

Most human knowledge is generic, in the sense that it applies to different instances of the same general situation.

We can state such generic knowledge as symbolic PATTERNS that:

- Describe the structures held in common by these instances.

- Designate the specified class of situations with these structures.

- Omit structures not held in common and use VARIABLES to indicate subelements that vary.

A symbol system interprets a pattern by MATCHING it against one or more symbol structures that describe a situation.

This opens the door to declarative representations of knowledge that support nonprocedural varieties of processing.

Symbolic Patterns as List Structures

We can represent symbolic patterns as lists or list structures that contain pattern-match variables in place of constants. E.g.,

```
     (taller-than ?person1 ?person2)
and
     (on ?block1 ?block2)
```

contain the variables ?person1 and ?person2, and the variables ?block1 and ?block2, respectively.

We can encode complex patterns as lists (sets) of simple patterns that may share pattern-match variables. E.g.,

```
     ((taller-than ?person1 ?person2) (taller-than ?person2 ?person3))
and
     ((on ?block1 ?block2) (wider-than ?block2 ?block1)
      (on ?block2 ?block3) (wider-than ?block3 ?block2))
```

are two complex patterns with two and four patterns, respectively.

Simple Matches and Bindings

We can attempt to MATCH a simple pattern against a ground literal (a list or list structure with no variables).

A match succeeds if there is a consistent set bindings (substitution of constants for variables in the pattern) that gives the ground literal.

- The pattern (taller-than ?person1 ?person2) matches against literal (taller-than Abe Bob) with the bindings

  ((?person1 . Abe) (?person2 . Bob))

- The pattern (on ?block1 ?block2) matches against literal (on A B) with the bindings

  ((?block1 . A) (?block2 . B))

Different variables can bind to the same constant, but each variable must map to a single constant.

List Structures in Working Memory

Symbolic patterns require content against which to match. We will refer to this set of candidate literals as a working memory.

Here are two working memories that contain two and three elements, respectively:

```
((taller-than Bob Cal) (taller-than Abe Bob))
```

```
((on B C) (on C D) (on A B))
```

Although one typically encode working memories as list structures, the order of elements does not matter.

Finding All Simple Matches

Given a simple pattern and the contents of working memory, we can find all matches of the pattern against its elements.

For the simple pattern (on ?block1 ?block2) and the working memory ((on B C) (on C D) (on A B)), there are three matches:

(on A B) with bindings ((?block1 . A) (?block2 . B))

(on B C) with bindings ((?block1 . B) (?block2 . C))

(on C D) with bindings ((?block1 . C) (?block2 . D))

Note that many other possible substitutions or bindings, such as ((?block1 . A) (?block2 . C)) are not legitimate matches.

Complex Matches and Bindings

We can also attempt to match a complex pattern against the elements in working memory.

- The pattern ((taller-than ?p1 ?p2) (taller-than ?p2 ?p3)) matches the literals (taller-than Abe Bob) and (taller-than Bob Cal) with bindings

  ((?p1 . Abe) (?p2 . Bob) (?p3 . Cal))

- The pattern ((on ?block1 ?block2) (on ?block2 ?block3)) matches the literals (on A B) and (on B C) with the bindings

  ((?block1 . A) (?block2 . B) (?block3 . C))

A successful match must bind variables consistently against each of the pattern’s simple subpatterns.

Finding All Complex Matches

Given a complex pattern and the contents of working memory, we can find all matches of the pattern against its elements.

For example, for the complex pattern

```
     ((on ?block1 ?block2) (wider-than ?block2 ?block1)
      (on ?block2 ?block3) (wider-than ?block3 ?block2))
```

and the working memory

```
     ((on B C) (on A B) (on C D) (wider-than B A)
      (wider-than C B) (wider-than D C))
```

there are two matches:

```
     (((on A B) (on B C) (wider-than B A) (wider-than C B)) =>
      ((?block1 . A) (?block2 . B) (?block3 . C)))
and
     (((on B C) (on C D) (wider-than C B) (wider-than D C)) =>
      ((?block1 . B) (?block2 . C) (?block3 . D)))
```

Other bindings are possible, but they do not lead to consistent matches against elements in working memory.

Complex Patterns with Negations

One can also specify NEGATED patterns that designate literals that should NOT be match.

Consider the complex pattern ((on ?X ?Y) (not (on ?any ?X)) and the working memory

```
((on B C) (on A B) (on C D))
```

This pattern produces only one match:

```
((on A B) => ((?X . A) (?Y . B)))
```

since (on ?any ?X) would match (on A B) if ?X were bound to B and it would match (on B C) if ?X were bound to C.

Patterns that include negations add representational power, but they should be used carefully.

Types of Complex Symbolic Patterns

Complex symbolic patterns are used widely within the AI community; they appear in:

- in the conditions of production rules
- in the bodies of Horn clauses
- in the structure of schemas
- in the structure of frames
- in the conditions of planning operators
- in the content of unification grammars

These differ in their syntax and semantics, but they share the use of generalized list structures for pattern matching.

Conceptual Clauses in Icarus

Icarus is a cognitive architecture that encodes conceptual knowledge as relational patterns such as:

```
    ((left-of ?block1 ?block2)
     :percepts  ((block ?block1 xpos ?xpos1)
                 (block ?block2 xpos ?xpos2))
     :tests     ((< ?xpos1 ?xpos2)) )
and
    ((between ?block1 ?block2 ?block3)
     :percepts  ((block ?block1) (block ?block2) (block ?block3))
     :relations ((left-of ?block1 ?block2)
                 (left-of ?block1 ?block2)) )
```

Such conceptual clauses may against PERCEPTS of objects like:

```
    (block A xpos 2 ypos 0 width 2 height 2)
```

and against inferred BELIEFS like:

```
    (left-of A B) and (left-of B C)
```

Patterns in Icarus may also have negations like (not (on ?any ?block)).

Pattern Matching and Unification

Unification is an extension of pattern matching but supports mapping of two patterns, each of which may contain variables.

For example, the simple patterns (on A ?block1) and (on ?block2 B) unify to produce the bindings

```
((?block1 . A) (?block2 . B)
```

whereas the simple patterns (on A ?block) and (on ?block B) do not unify because ?block cannot bind both A and B.

However, the patterns (on A ?block1) and (on ?block2 ?block3) unify with the bindings

```
((?block1 . A) (?block2 . A) (?block3 . A))
```

because different variables can bind to the same constant term.

Complex Unification

One can also apply unification to complex patterns, but it behaves somewhat differently from pattern matching.

For example, consider the two complex patterns:

```
     ((on A ?block2) (on ?block1 B) (on ?block1 ?block2))
and
     ((on ?block3 B) (on ?block3 ?block2))
```

We can unify these two structures, even though they have different numbers of elements, to produce the bindings:

```
    ((?block1 . A) (?block2 . B) (?block3 . ?block1))
or
    ((?block1 . A) (?block2 . B) (?block3 . A))
```

depending on which of the two formats we prefer for bindings.

Assignments for Meeting 5
Deductive Reasoning

Read the article:

- Genesereth, M. R., & Ginsberg, M. L. (1985). Logic programming. Communications of the ACM, 28, 933-941.

- This paper shows how logic programming frameworks like Prolog combine unification with chaining to carry out deductive inference.

- Review notes on pattern matching in preparation for the second exercise.