Abstraction Techniques

One way to describe LISP programming (in particular) and AI (in general) is

the art of successive abstraction

That is, LISP programming is writing low level functions, then writing higher level programs which use the lower level functions as basic building blocks, then writing languages which use the high level programs as building blocks, etc.

A primary example of abstraction is the UNIFY algorithm, which takes concrete facts (written in a relational form of predicate calculus) and abstracts them into arbitrary patterns which relate sets of facts. UNIFY does this by introducing pattern variables. The ELIZA program uses pattern variables to turn normal English sentences into an apparently therapeutic dialog. So, for example, the three declarative facts

```
(John likes Mary)
(John likes Tom)
(John likes Sarah)
```

can be abstracted into a single idea of "the people John likes":

```
(John likes ?x)
```

A pattern matching engine collects the basic facts that match the pattern query, saving the user from having to ask "Does John like Mary?", "Does John like Tom?", "Does John like Sam?", etc. Pattern variable abstraction is entirely general, if you know the form of the facts in the database, you can ask for any particulars, including, for example, the general query about all facts relating three labels:

```
(?x ?y ?z)
```

Another example of abstraction is the GENERIC-SEARCH algorithm, which takes concrete state spaces (written as sets of state descriptions in a form of predicate calculus) and concrete state transformations (written as changes to the current set of state descriptions) and abstracts both of them into arbitrary patterns of traversal of state space. GENERIC-SEARCH does this by treating traversal functions as next-state priority queues, and introducing state evaluation functions to decide the traversal priorities. ELIZA, PARCIL and GPS can each be easily modified to search their respective state spaces more intelligently by using the techniques of GENERIC-SEARCH.

Our LISP code distribution contains examples of many different tools for abstraction of programming objectives. The distribution emphasizes several programming techniques for saying more with less. The philosophy is that when the problems are very difficult, it is more important for the program to be understandable than to be efficient. Another deep programming technique, however, is that code optimization itself is an automated transformation. In compiling code, the programmer is not expected to read and debug the resulting machine code. Similarly, in LISP programming, the programmer can express the desired functionality at the highest level of abstraction *without paying a performance cost*. In the language of SE553, the

programmer should focus on expressing the mathematical model of the conceptualization of the problem, building abstraction tools rather than implementing low level optimizations.

AI emphasizes conceptual understanding by providing abstract implementation tools and languages. The LISP code distribution includes the following packages for applying abstraction to programming style.

I. FILTERS AND STREAM FUNCTIONS WITH DELAYED EVALUATION

This package illustrates the techniques of *dataflow programming*. A program is conceptualized as a stream of data which flows through filters that remove unwanted data structures. The "results" flow out the other end of the stream. This technique is incorporated deeply in the Mathematica language, and is essential to the mathematical (or functional) programming style. New programming tools include BACKQUOTE and DEFMACRO.

BACKQUOTE is just like QUOTE in that the form following the backquote is not evaluated, being treated as literal. However inside BACKQUOTE, any forms preceded by the function COMMA are evaluated, in the context that the backquote form is in. This permits passing structured forms, including functional forms, into unknown environments and having them behave appropriately.

DEFMACRO is just like DEFUN in that it defines a function. However, the macro-function being defined is expanded inline; when a function which uses a macro is loaded, the macro code is automatically written in place of the macro name. Macros permit naming and abstraction of code components without the overhead of introducing new function calls.

To make stream objects which generate data items only when needed, BACKQUOTE and DEFMACRO are combined. For instance the macro-function DELAY:

```
(defmacro delay (exp) `(function (lambda () ,exp)))
```

places a function "shell" inline wherever the function DELAY is used. The arguments of DELAY form the body of the function shell. Thus CONS-STREAM

```
(defmacro cons-stream (new stream) `(cons ,new (delay ,stream)))
```

provides the function body STREAM to DELAY. (CONS-STREAM itself is a macro) Like CONS, CONS-STREAM adds a new element to the front of a stream, but the stream itself does not yet exist as data, only as potentia. When CONS-STREAM occurs in a function body, it turns into

```
(cons <new> (function (lambda () <stream>)))
```

with `<new>` and `<stream>` being provided by the function which uses CONS-STREAM. Since CONS-STREAM is a macro, whenever it is used, LISP builds an "anonymous" function in place of the code `(function (lambda () <stream>))`, which makes the code look like

```
(cons <new> #<Anonymous Function #x267F5A6)
```

Finally the function FORCE, called by REST-STREAM, actually evaluates the anonymous function which has been defined in the context of the constructor CONS-STREAM:

```
(defun force (function-closure) (funcall function-closure))
(defun rest-stream (stream) (force (rest stream)))
```

Thus the set of constructor and accessor functions (CONS, FIRST, REST) turn into *generators*. CONS-STREAM builds a "list" of data consisting of a first element and a function which generates the next element. When REST-STREAM is called in the course of a computation, the generator function is evaluated by FUNCALL to build the next data element. Exactly what data element it builds is determined by the function which uses CONS-STREAM. Notice the subtlety that the anonymous generator function itself is not a static entity, it contains within it the code to build the next generator function, not just the next data element.

The stream abstraction permits programs to accumulate or process collections that we do not know how to count at compile time. The code example, for instance, asks for the first N Fibonacci numbers which meet an arbitrary test function, such as the first 20 odd Fibonacci numbers, or the first 8 Fibonacci numbers containing the digit 5 at least three times.

To accumulate items from a stream, we place a filter function in the stream. The final abstraction is the function ACCUMULATE-FILTERED-STREAM, which takes as parameters a stream generating function, a filter function, and a desired number of items which pass the `filter`.

II. LOGIC PROGRAMMING

This package illustrates *metalinguistic abstraction*, or declarative programming. A program is expressed in a descriptive language (in predicate calculus), and the language interpreter applies pattern matching to provide solutions. The abstraction is at the level of language, in that the interpreter is not sensitive to the form or the vocabulary of the language used to represent and describe the problem domain.

Yet another UNIFY algorithm is included in this package. Note that variables are expressed as a list `(var x)`. Of course the `x` can be any label, since the program takes care of providing unique names for all variables

To use the program, type `(logic-shell)` to LISP. A new top-level interpreter is called, which recognizes logical queries. Be sure to load an assertions database before entering the LOGIC-SHELL.

Note: when loading new *ASSERTIONS*, be sure to exit the logic-shell first.
Note: The *DONE* signal alone is returned when there are no solutions.

III. OBJECT-ORIENTED PROGRAMMING

Objects, messages, and inheritance are an alternative declarative organizational structure for programming. By seeing the internal mechanisms of object-oriented programming, you can understand that object-orientation is simply a different way of organizing and accessing a database. The abstraction is in the *programming metaphor*. Our culture is object-oriented (as

opposed to process oriented), so we think more efficiently in terms of objects and their interactions, than in terms of pure functionality.

The model of process as *messages passing between discrete objects* is at the foundations of Shannon's theory of information, which greatly influenced the origins of Computer Science. Object-orientation is excellent for digital systems which have been designed as discrete objects sharing communication lines, but it is weak for process-oriented models (such as a lighthouse sweeping a beam across the ocean and the waves breaking upon the shore) and for all biological and human systems.

The LISP code for object-oriented programming makes extensive use of symbol databases. Every symbol in a LISP environment has its own database. The function

```
(get <object-name> '<property>)
```

retrieves the value of a property associated with the object-name, whether or not the object is an instance or a class or a data-structure. The function

```
(setf (get <object-name> '<property>) <value>)
```

assigns a value to a property of a given object-name. Although we think of objects and classes, the LISP system treats all "things" as symbols, that is, as names. The function

```
(symbol-plist <name>)
```

displays an object-name's properties and their current values.

Methods are treated the same way, as properties associated with symbols. The difference is that the value of a method is a lambda form, that is, an abstract function waiting to have its formal parameters bound to values.

The primary tools of abstraction in object systems are polymorphism and inheritance. Inheritance is a search of each symbol's database, following the ISA property to the database of classes and superclasses. Polymorphism is having a separate name-space for different databases of different symbols.

The function BUILD-ROOT provides some elementary methods (i.e. lambda forms organized under property names) for the root-object ROOT These are

ROOT	
SHOW	<print the ISA, VARIABLES and METHODS properties>
SHOW-PARENTS	<print the ISA values>
SHOW-VALUE	<evaluate the name given as the argument to show-value>
SHOW-ENV	<build the environment of SELF>
SET-VALUE	<given property and value arguments, set the value>

To use an object environment, build the environment, then send messages to the objects about their properties.

IV. SEMI-NATURAL LANGUAGE PARSING

PARSE analyzes an English sentence in the same way that the English classes teach (or at least taught 30 years ago) students how to diagram a sentence. The recursive descent parser uses a technique similar to the one PARCIL uses for the C language. It is an example of top-down programming, each component can be written without knowledge of what the other components do.

English sentences are defined to follow a *grammar*, that is, a theory of structure. The code disassembles its input (i.e. sentences) by following the rules of the grammar. This implementation assumes that an acceptable sentence can be decomposed into compound fragments. The compound fragments can be composed of other compound fragments, but eventually everything is composed of simple words. The words are listed in a *lexicon*, a dictionary which classifies words into their types. (Obviously natural English is far more complex. The code sample generalizes to the construction of arbitrary grammars.)

COMPOUND FRAGMENTS

```
<S>   -->  <NP> <VP>
<NP>  -->  [<detr>] <adjv>* <noun> <PP>*
<VP>  -->  <vaux>* <verb> [<NP>] <PP>*
<PP>  -->  <prep> <NP>
```

`*` means the part may occur zero or more times
`[...]` means the part may occur zero or one time

Abstraction	*Part-of-grammar*	*Function which analyzes the fragment*
`<S>`	Sentence	`(sentence <s>)`
`<NP>`	Noun Phrase	`(noun-phrase <np>)`
`<VP>`	Verb Phrase	`(verb-phrase <vp>)`
`<PP>`	Prepositional Phrase	`(prep-phrase <pp>)`

SIMPLE WORDS

Abstraction	*Part-of-grammar*	*Function which accesses lexicon*
`<noun>`	Noun	`(noun <w>)`
`<detr>`	Determiner	`(detr <w>)`
`<verb>`	Verb	`(verb <w>)`
`<prep>`	Preposition	`(prep <w>)`
`<vaux>`	Verb Auxiliary	`(vaux <w>)`
`<adjv>`	Adjective	`(adjv <w>)`

It is tempting to put the input sentence into a global variable which is used up from left to right by the parser. However the parser needs to access each fragment twice, once to determine its type and once to build the parse tree. Since the type of the fragment is unknown when it is first accessed, this strategy requires a host of other supporting global variables, leading to difficult

to debug dependencies between parsing functions. Failures in guessing the correct type lead to backtracking, that is to unwinding previous results, a nightmare for data structure management. But the major problem is that compound fragments can contain other compound fragments recursively, so that a function which assigns a value to a global variable might be called again within the same sentence, losing the values assigned in the outer context. For example, our grammar permits NP to call PP which may have an NP in it.

Assignment statements (that is, SETQ) work differently each time you use them because the calling context (the binding of local variables) is different. This undermines modularity by creating dependencies and interactions between program segments. The solution is to use LAMBDA forms which do not bind function arguments until the appropriate time (similar to the way STREAMS work, but simpler.) A LAMBDA has the form

```
(lambda (formal-parameters) <body>)
```

It can be conceptualized as an *abstract operator*, independent of its arguments, with the operation defined by the body of the LAMBDA. A *function-call* is the form

```
((lambda (formal-parameters) <body>) <contextual-values>)
```

The evaluation of a functional form consists of evaluating the arguments, then substituting the values of the arguments for their formal parameters in the body of the LAMBDA, then evaluating the LAMBDA body itself. For example, the function PARSE-SENTENCE has the form

```
((lambda (s-pair) <body-which-uses-s-pair>) (sentence s))
```

First the function SENTENCE processes the input sentence, then that result is substituted into the LAMBDA body and the result is evaluated.

Similarly, the function SENTENCE has the form

```
((lambda (np-pair) <body>) (noun-phrase s))
```

This calls NOUN-PHRASE to process the beginning of the sentence. Whatever the result (for instance the sentence may start with a verb phrase and not with a noun phrase), it is handed in an appropriate form to the next function to continue parsing.

The parsed results of a fragment processing function are conveyed to the next processing function simply by CONSing them onto the sentence being processed. This is the meaning of "s-pair", "np-pair", etc. The FIRST of any current fragment is the results of prior processing, the REST is the parts yet to be processed. Thus the output functions can build the results before all of the results are known, and all backtracking is eliminated. And the output functions are precisely the body of each LAMBDA form. Finally, all fragment processors use the same technique, thus all the functions are similar.

The only remaining complexity is to handle the grammar modifiers `*` and `[...]`. The functions FIND-* and FIND-? are abstracted to handle any type of sentence fragment. In the current grammar, `*` applies only to the set {adjective, prepositional-phrase, verb-auxiliary}, while `[...]` applies to the set {determiner, noun-phrase}. Note that `[...]` is renamed `?` since LISP already has an assigned meaning for square brackets. Changing the grammar is easy when the

language features are abstracted away from the particular instances they are used for. Of course, FIND-* uses the same LAMBDA technique as the rest of the parser. FIND-?, which identifies zero or one occurrence, calls the type being tested. If the parse fails, it CONSes NIL to indicate that no occurrence was found.

In summary, here are some things to notice about the PARSE code:

1. No assignment statements, so no variables which you have to track how they are being used.

2. No special treatment for any parsing categories, code uniformity = code maintainability/readability/modifiability

3. No backtracking or storage of temporary results, thus easy tracing and debugging.

4. The * and [...] types are abstracted into FIND-* and FIND-? functions, making extensibility easy.

5. No global variables, so functions have no interaction and can be modified modularly.

Yet more abstraction is available, but not fully implemented. Our grammar has three occurrence types,

`<...>`	for simple words and compound forms
`<...>*`	for sequences of zero or more forms
`[...]`	for zero or one occurrences of a word or form

The function PARSE-FIND removes the code dependency of each special occurrence form, by managing the metalanguage of the grammar. This leads to easy extensibility, not of the grammar, but of the grammar's metalanguage. You may, for example, wish to establish an occurrence operator `!` which calls for exactly one occurrence of a type, zero and more than one occurrence being trapped as ungrammatical. Sentence punctuation requires this occurrence rule.

Finally (finally!) a function which is not difficult to write, but not included, is GENERIC-PARSE, which takes a sentence and a grammar. The grammar data structure may look like

```
(defvar current-grammar
    '((SENTENCE (NP VP))
      (NP   ((? DETR) (* ADJV) NOUN (* PP)))
      (VP   ((* VAUX) VERB (* PP) (? NP)))
      (PP   (PREP NP))
      (NOUN (member-of *nouns*))
      (DETR (member-of *determiners*))
      (VERB (member-of *verbs*))
      (PREP (member-of *prepositions*))
      (VAUX (member-of *verb-auxiliaries*))
      (ADJV (member-of *adjectives*)) ))
```

GENERIC-PARSE is a fully abstract tool for exploring types of parsing grammars. Here is where AI research begins, since the metalinguistic abstractions provided by the code make it easy to experiment with a variety of grammars and parsing metalanguages. A researcher would build a database of sentences, and then build grammars which parsed the database. These parsings would be passed to a statistics collection program which would report of the success of each grammar. The (happy) researcher spends all his/her time at the conceptualization and modeling levels, and before long, a book of results is collected.

V. MISCELLANEOUS

1. Here is the recursive REVERSE which I did not do in class:

```
(defun reverse553 (lst)
      (if (atom lst) lst
      (append (reverse553 (rest lst))
      (list (reverse553 (first lst))))))
```

2. Here is how to build a semantic net in LISP. You might try drawing the graph of this birds and animals example (the spacing is for readability only. LISP ignores whitespace.):

```
(setf (get 'animal  'covering) 'skin)
(setf (get 'bird    'covering) 'feathers)
(setf (get 'bird    'travel)   'flies)
(setf (get 'bird    'isa)      'animal)
(setf (get 'fish    'isa)      'animal)
(setf (get 'fish    'travel)   'swims)
(setf (get 'ostrich 'isa)      'bird)
(setf (get 'ostrich 'travel)   'walks)
(setf (get 'penguin 'isa)      'bird)
(setf (get 'penguin 'travel)   'walks)
(setf (get 'penguin 'color)    'brown)
(setf (get 'opus    'isa)      'penguin)
(setf (get 'canary  'isa)      'bird)
(setf (get 'canary  'color)    'yellow)
(setf (get 'canary  'sound)    'sings)
(setf (get 'tweety  'isa)      'canary)
(setf (get 'tweety  'color)    'white)
(setf (get 'robin  'isa)       'bird)
(setf (get 'robin  'sound)     'sings)
(setf (get 'robin  'color)     'red)
```

3. Here is planning example that is not blocks world. The idea is to find a plan that allows the robot to get the key and take it to the exit. You should be able to program the solution to this in either GPS or the logic engine.

```
(caves c1..c18)          ;a listing of all 18 caves
(tunnel c1  c7)          ;the tunnels between caves
```

```
(tunnel c1  c2)
(tunnel c2  c8)
(tunnel c3  c9)
(tunnel c4  c10)
(tunnel c6  c12)
(tunnel c8  c9)
(tunnel c8  c14)
(tunnel c9  c10)
(tunnel c9  c15)
(tunnel c11 c12)
(tunnel c10 c16)
(tunnel c11 c17)
(tunnel c12 c18)
(tunnel c13 c14)
(tunnel c15 c16)

(robot in 3 at 0)        ;initial locations of robot, key and exit at time 0
(key   in 4 at 0)
(exit in 16 at 0)

(move ci cj)             ;movement action
(grab  x ci)             ;pick up action
(drop  x ci)             ;put down action is not needed in simple game
(open ci)                ;use key action
```

This is the framework of a planning game called WUMPUS. To make a more interesting game, you could extend this framework by

1. have a ladder that is necessary for some (vertical) tunnels
2. make the robot put down the key when holding the ladder
3. add new tunnels (digging) at selected or random times;
   delete some tunnels (cave-ins) at selected times.
4. have another creature (the WUMPUS) moving around the tunnel maze, which the robot can hear when in an adjacent cave. If the WUMPUS is in the same cave, it eats the robot, or steals the key/ladder (putting it in some random place).
5. add gold in certain caves and require the robot to collect an amount of gold before exiting.
6. add teleportation from a specific cave to a random cave
7. add a second robot and have a contest to see who exits first.
8. have an automated robot and a robot moving under user control
9. add friendly natives in some caves which give hints; add some natives which give false hints.

The idea, of course, is that with a free week, class members have sufficient skills to model these extensions and implement WUMPUS within the context of either GPS, or LOGIC ENGINE, or OBJECTS. By combining PARSE and ELIZA, you could even have the robot talking with the natives in the caves.