

Python ACT-R: A New Implementation and a New Syntax

Terrence C. Stewart <tcstewar@connect.carleton.ca>

Robert L. West <robert_west@carleton.ca>

Carleton Cognitive Modelling Lab, Institute of Cognitive Science, Carleton University
1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada

We present a re-implementation of ACT-R and a new syntax for the creation of ACT-R models. This allows for easier development of new sorts of modules and a more gradual learning curve. This short (800 line) implementation provides for all of the core functionality of ACT-R (including production compilation, optimized and non-optimized declarative memory learning, and basic environment interaction). This process has also allowed us to investigate the distinction between the theory of ACT-R and the details of the standard Lisp implementation.

ACT-R is the most extensively developed, widely used, and carefully examined architecture for modelling human cognition that we have. Its successes are broad and startling, and it is an exemplar of the sort of theory that cognitive science strives toward. However, due to various factors, ACT-R has a rather harsh barrier to entry. Some of this difficulty is due to its complexity: learning any system that attempts to describe human cognition is naturally going to involve a certain degree of effort. However, depending on the user's background, the architecture itself can have an impact on how understandable it is. In particular, people with a strong background in Lisp typically have a much easier time, especially when running experiments with ACT-R, or collecting data from multiple runs of an ACT-R model. Thus, one way to make ACT-R accessible to a wider audience is to implement it in other languages.

Our project is a complete functional reimplementation of ACT-R with this in mind. We have the following goals: (1) To confirm that ACT-R is doing what we think it is doing. A complete reimplementation of software is often used in industry to confirm functionality in this way. (2) To investigate the distinction between the theory of ACT-R and an implementation of ACT-R. It is possible that certain aspects of ACT-R are due more to the implementation choices than to the theoretical commitments. (3) To make it easier for ACT-R researchers to investigate modifications and additions to it. This is one of the goals of ACT-R 6 but it still requires an extensive knowledge of Lisp. Making ACT-R available in more languages will help this process.

In creating this new version, we also have an opportunity to change the syntax of ACT-R. The current syntax is heavily embedded in its Lisp roots, and this can be a significant barrier to entry for new users. Therefore, we are taking this opportunity to develop an alternate syntax which fits well within the new implementation, and will be more familiar to people with a procedural/object-oriented programming background (e.g. C++, Java).

Implementation

Our reimplementation is in the Python language. It was chosen due to the first author's success using it in a graduate course to teach non-programmers to develop connectionist

and evolutionary computational models of cognitive systems (Stewart, 2004). The language is often described as 'executable pseudo-code' due to its goal of having a syntax that is as clear as possible both for writing and reading. Significant effort has gone into making it suitable for both beginner and expert programmers, and it supports an elegantly subtle transition from procedural to functional programming. It is also freely available, Open Source, widely ported, and has a comprehensive built-in library.

Importantly, Python "supports all of Lisp's essential features except macros" (Norvig, 2000). This gives the full power of Lisp, but a constrained syntax. This syntax has been carefully designed for fast development, clarity, and ease of learning. For a full discussion and comparison between the two languages, see (Norvig, 2000). Our experience has been that academics with no programming background are able to read and understand Python code, and that this accessibility makes them more likely to develop computational models within their own research.

Syntax

Unlike the jACT-R project (Harrison, 2005), which uses Java to process models written in the standard ACT-R syntax, we write ACT-R models as normal Python code. This makes it seamless to interact with other Python software for defining experiments, new modules, and other models. It also leads us to a different, but identically expressive, syntax.

As a sample, here is the Lisp version of the increment-sum production from the addition model in ACT-R Tutorial 1, followed by the Python ACT-R version:

```
(P increment-sum
=goal>
  isa      add
  count    =count
  sum      =sum
=retrieval>
  isa      count-order
  first    =sum
  second   =newsum
==>
=goal>
  sum      =newsum
+retrieval>
  isa      count-order
  first    =count
)

def incrementSum(goal='add ?count ?sum',
                 retrieval='count-order ?sum ?newsum'):
    goal(sum=newsum)
    retrieval('count-order ?count ?')
```

The first two lines of the Python version define the LHS, which matches on both the goal buffer and the retrieval buffer. The last two lines are the RHS, and show the modification of a slot in the goal buffer and a retrieval request. From the Python point of view, this is a function definition, and the arguments and default values for the function are used as the LHS, while the body of the function is used as the RHS. Our Python ACT-R system extracts this information from the production and uses the matching rules to determine when it will fire.

Python ACT-R supports the same functionality as the ‘-’ and ‘=’ (and the new ACT-R 6 ‘?’) matching on the LHS, and the ‘-’, ‘=’, and ‘+’ commands on the RHS.

Chunk Syntax

The most controversial change is that chunks in Python ACT-R do not have named slots. The main reason for this is to reduce a confusion we have frequently encountered. Many people learning ACT-R have difficulty remembering that slot names do not have semantic value. Furthermore, determining a good name for a slot can be difficult if the slot is used in different ways in different productions.

Since slot names are not part of the theory of ACT-R, but are rather there for the convenience of the modeller, we have chosen to investigate what happens if we identify slots by position, rather than by a slot name. We have found two interesting positive benefits of this approach. First, it eliminates the need to keep track of both slot names and variable names (which is confusing in the common situations where the slot name and the variable name are identical, as in something like ‘first =first’). Instead, we use bound variables as temporary slot names within a particular production.

Second, it makes the creation of chunks with many slots much less convenient. Although it has been argued that there should be only 7 ± 2 slots in a chunk, it is still common to see models with a larger numbers. This is not easily noticeable when one examines just the productions, since each production may only use a few slots. However, in Python ACT-R, since slots are identified by position, you must explicitly say in the production that certain slots should be ignored. This seems to form an interesting soft constraint on the number of slots it is practical to use.

Chunks in Python ACT-R also do not have a chunk-type, or a name. There is thus no need for the (chunk-type ...) declarations. Instead, chunks are an ordered list, usually represented as a line of text with spaces separating the elements (although they can be any arbitrary list).

Lisp ACT-R:

```
(chunk-type count-order first second)
(a ISA count-order first 0 second 1)
```

Python ACT-R:

```
'count-order 0 1'
```

Left-Hand-Side Syntax

Where Lisp ACT-R uses the ‘=buffername>’ syntax to identify what buffer we are referring to on the LHS, Python ACT-R uses a series of default function arguments.

Lisp ACT-R

```
=goal>
.....
=retrieval>
.....
=whatever>
.....
```

Python ACT-R

```
goal='.....'
retrieval='.....'
whatever='.....'
```

To specify a matching pattern for a buffer, we use a similar syntax as when we defined the chunks. This is a text string that uses spaces to separate the slots. Since there are no slot names, order matters. Python ACT-R uses a ‘?’ to indicate variables (much like Lisp ACT-R uses ‘=’).

Lisp ACT-R

```
=goal>
isa add
arg1 =num1
arg2 =num2
sum nil
```

Python ACT-R

```
goal='add ?num1 ?num2 nil'
```

Since order matters, slots that are unimportant for the match cannot be left out. Instead, a ‘?’ is used to indicate that this slot is unimportant.

Lisp ACT-R

```
=goal>
isa add
arg1 =num1
sum nil
```

Python ACT-R

```
goal='add ?num1 ? nil'
```

If a variable is used in two (or more) slots, then that forces both slots that use the variable to have the same content (exactly as in Lisp ACT-R).

Lisp ACT-R:

```
=goal>
ISA          add
sum          =sum
count        =count
=retrieval>
ISA          count-order
first        =sum
second       =newsum
```

Python ACT-R:

```
goal='add ?sum ?count',
retrieval='count-order ?sum ?newsum'
```

To indicate that a slot should not match, Python ACT-R uses a ‘!’’. This can be combined with the ‘?’ for variables, giving the following possibilities:

Lisp ACT-R

```
- sum seven
- sum =sum
```

Python ACT-R

```
!seven
!?sum
```

To do more than one match on the same slot, you can combine these together.

Lisp ACT-R	Python ACT-R
sum two	
sum =sum	two?sum!three!eight!?other
- sum three	
- sum eight	
- sum =other	

Right-Hand-Side Syntax

In Python ACT-R, the RHS of a production is implemented as the body of the function being defined. This means that any arbitrary Python code can be written for the RHS. However, this would be the equivalent of abuse of Lisp ACT-R's !eval! command. So, for normal models, the RHS should be restricted to commands which are all buffer and module requests.

Inside the RHS, you have access to all of the bound variables from the LHS. These are treated exactly like normal Python variables.

Lisp ACT-R	Python ACT-R
!output! (=num1)	print num1

Modifying a particular slot in a buffer is done using temporary slot names. Any bound variable can also be used to refer to the slot that the variable is bound to. By setting a new value for this variable, we actually modify the slot itself.

Lisp ACT-R	Python ACT-R
(p set-sum	def setSum(
=goal>	goal='add ?count ?sum'):
isa add	goal(sum=count)
count =count	
sum =sum	
==>	
=goal>	
sum =count	
)	

To put a new chunk into a buffer, or to make a request of a module, the chunk is specified in the same manner as in the LHS. Bound variables can be used, and a '?' in a request indicates slots that are not important to match on.

Lisp ACT-R	Python ACT-R
+goal>	goal('add ?num1 0')
isa add	
sum =num1	
count 0	
+retrieval>	retrieval('order ?count ?')
isa count-order	
first =count	

One open question is whether this syntax should also be used for modifying slots in buffers. If so, it may be possible

to consolidate buffer changes ('=') and module requests ('+') into a single type of RHS request.

Creating a Full Model

The following is the full source code for the ACT-R Tutorial Unit 1 counting model. We start by importing our ACT-R library, which gives Python access to the ACT-R system we have written.

```
import actr
```

The model definition is contained within a single Python class. This class can then be used to create multiple instances of that model. It explicitly specifies what modules exist within this model, and what they are called. Note that it is completely possible to have multiple declarative memory systems, if desired. Adding a new buffer/module is as simple as adding a new line in these declarations. It is also possible to design new modules and add them here, as long as the module conforms to a basic set of rules (it must be able to indicate the contents of its buffer, and it must be able to respond to requests). Note that we have chosen to have exactly one buffer per module.

```
class Count:
    goal=actr.Buffer
    retrieve=actr.BasicMemory
    production=actr.BasicProduction
```

To simplify the initialization of declarative memory, a large set of chunks can be created like this:

```
memory="""count 0 1, count 1 2,
count 2 3, count 3 4,
count 4 5, count 5 6,
count 6 7, count 7 8,
count 8 9, count 9 10"""
```

Next, the productions are defined. They make use of the named modules created previously. For each production, we specify its name, the LHS matching rules, and the actions to take on the RHS. Note that the RHS is simply the body of a normal Python function, which means that any valid Python code can be used (such as the print command).

```
def start(
    goal='count-from ?start ?end starting'):
    retrieve('count ?start ?next')
    goal('count-from ?start ?end counting')

def increment(goal='count-from ?x !?x counting',
    retrieve='count ?x ?next'):
    print x
    retrieve('count ?next ?nextNext')
    goal(x=next)

def stop(goal='count-from ?x ?x counting'):
    print x
    goal('count-from ?x ?x stop')
```

Now that the model has been defined, we can create it by giving it to the ACT-R system. Multiple models can be created, as can multiple instances of the same model. All communication between these models is done outside of the Python ACT-R system.

```
model=actr.ACTR(Count)
```

Once the model has been created, it is possible to use any RHS command to explicitly set buffer values. This is most useful to set the goal, but can also be used to add chunks into declarative memory.

```
model.goal('count-from 2 5 starting')
```

Finally, we can run the model. A model can be run for a set period of virtual time, a certain number of steps, or until there are no pending productions.

```
model.run()
```

The system can also create log files as expected.

```
0.000 focus='count-from 2 5 starting'
0.000 'start' Selected
0.050 'start' Firing
0.050 focus='count-from 2 5 counting'
0.100 retrieve='count 2 3'
0.100 'increment' Selected
0.150 'increment' Firing
2
0.150 focus='count-from 3 5 counting'
0.200 retrieve='count 3 4'
0.200 'increment' Selected
0.250 'increment' Firing
3
0.250 focus='count-from 4 5 counting'
0.300 retrieve='count 4 5'
0.300 'increment' Selected
0.350 'increment' Firing
4
0.350 focus='count-from 5 5 counting'
0.350 'stop' Selected
0.400 'stop' Firing
5
0.400 retrieve='count 5 6'
0.400 focus='count-from 5 5 stop'
0.400 No events left to process
```

All of the system parameters are available for modification at any time, or can be set when instantiating the model.

```
model=actr.ACTR(Count, retrievalThreshold=-0.5)
model.params.retrievalThreshold=-0.5
```

Current Status

Since the Python ACT-R modules are explicitly added to a particular model, it is easy to develop new models and make modified versions of existing models. We have also chosen to break the standard ACT-R modules up. This means that the optimized declarative memory module is actually a different module than the non-optimized version (although the one is, of course, based on the other). We currently have working versions of the following modules:

Declarative Memory

BasicMemory: no chunk activations, matching requests return the first chunk found. Same as setting (:ESC F) in Lisp ACT-R

FastMemory: optimized activation learning. Same as setting (:ESC T :OL T) in Lisp ACT-R

FullMemory: non-optimized activation learning. Same as setting (:ESC T :OL F) in Lisp ACT-R

Productions

BasicProduction: no learning of production weights. Same as setting (:PL F) in Lisp ACT-R

PGCProduction: learning of production activations using the PG-C rule. Same as setting (:PL T) in Lisp ACT-R

CompilingProduction: PG-C learning along with basic production compilation. Same as setting (:PL T :EPL T) in Lisp ACT-R.

Other Modules

Buffer: A simple buffer with a module that does nothing. Same as the Goal buffer in Lisp ACT-R.

SOS: An implementation of the Simple Object System used for creating environments for ACT-R (West, Emund, and Tacoma, 2005).

The system does not currently support partial matching, spreading activation, or direct linking of chunks, but these are all planned additions. We are also evaluating the possibility of an ACT-R/PM module.

Open Design Questions

As this project has developed, a number of questions have arisen as to the approach we should take. These are situations where the theory of ACT-R may be being influenced by the standard Lisp implementation. For example, it was easier in the Python implementation to have any chunk that was cleared from any buffer to automatically be merged into declarative memory, rather than that happening for just the goal buffer. We changed this to conform to Lisp ACT-R, and then discovered that in ACT-R 6 there is strong consideration given to having this happen to all buffers. We therefore think it is worthwhile for us to describe similar design decisions that arose during our work on this project, so as to get feedback from the ACT-R community as to whether these are legitimate possibilities.

Production Compilation

Our current production compilation system creates a new production with a RHS that is simply the two previous RHSs put together. This means that, in the standard case of

a retrieval request and a match on that request, the retrieval request is still made, instead of being bypassed, as in Lisp ACT-R. For the most part, this works out to be functionally the same as the more elaborate compilation system, and has the side effect of automatically dealing with the situation where a retrieval request is used by two different productions. Note that this is very different from the approach in ACT-R 6 of having a retrieval buffer automatically cleared (unless it is told not to) when it is made use of.

Goal Buffer

The goal buffer in Python ACT-R is not a special buffer. As already described, all buffers automatically add their contents to declarative memory. Furthermore, the goal buffer is not required on the LHS of Python ACT-R productions. Interestingly, this allows for *reactive* productions, which respond to information in a non-goal-driven manner.

Furthermore, since the names of buffers are not fixed in Python ACT-R, we have sometimes found it clearer to have a 'focus' buffer instead of a 'goal' buffer. This is accomplished simply by giving it a different name when creating the model.

It is also possible to have multiple buffers of this sort, although we have not yet found practical uses for this.

Chunks

Python ACT-R chunks are much simpler than Lisp ACT-R chunks. Lisp ACT-R chunks are full objects in their own right, allowing for inheritance and arbitrary numbers of slot values, whereas our chunks are simple ordered lists. This is a very different way of looking at chunks, but seems consistent with the core ACT-R theory. It does, however, mean that the elements within the slots of our chunks are not currently guaranteed to also be chunks themselves (they could be text strings, for example), although they can behave as chunks if needed.

Availability

The source code is freely available under the GNU General Public License, and can be downloaded from the project website at <http://ccmlab.ca/actr>. It is currently a total of 800 lines (22kB).

References

- Harrison, A.M. (2005) jACT-R. Available at <http://simon.lrdc.pitt.edu/~harrison/jactr.html>.
- Norvig, P. (2000) Python for Lisp Programmers. Available at <http://www.norvig.com/python-lisp.html>.
- Stewart, T.C. (2004) Teaching Computational Modelling to Non-Computer Scientists. Sixth International Conference on Cognitive Modelling.
- West, R.L., Emund, B., and Tacoma, J. (2005) Simple Object System (SOS) for creating ACT-R environments: A usability test, a test of the perceptual system, and an ACT-R 6 version. 12th Annual ACT-R Workshop.