Parsing, representing and translating AKT statements in webAKT

Robert Muetzelfeldt

April 2023

These notes describe three inter-related topics:

1. How we check that statements entered by the user (or provided in a knowledge-base file) are correct.

Currently, 'correct' only applies to their syntax - i.e. that they conform to AKT's grammar. The process for doing this is called parsing. In the future, the checks on the statements will be extended to their semantics - i.e. that the ame of an of an object appears where we expect an object.

2. How statements are represented internally in webAKT.

In AKT5, there is no distinction between the formal text form of a statement (i.e. as entered by the a researcher in AKT's formal notation) and the internal representation of that statement - they are both Prolog. It would be highly inefficient to do this in JavaScript, since JavaScript does not natively use Prolog's symbolic representation, so we would need to manipulate the statement held as a text string every time we wanted to process it. Instead, we used a notation called JSON, which allows all the terms (words) in a statement to be represented individually in a data structure which is essentially a nested array. This greatly facilitates any processing of statements.

3. How we convert between the various representations of a statement.

We need to be able to convert to and from AKT's formal (Prolog) syntax and the internal (JSON) notation. We also need to be able to convert statements into other forms - e.g. the stylised English used to present statements to non-AKT-trained users, and to and from a diagrammatic form for causal statements. In webAKT, the JSON format is the core one, with all conversions being done into or out of this one.

EXAMPLE

Here is a simple attribute-value from the Atwima KB, in AKT's formal (Prolog) syntax:

att\_value(part(asase\_kokoo,abuo),presence,erratic)

which we can read as

"the presence of the abuo part of asase\_kokoo is erratic"

where:

- assase\_kokoo and abuo are objects,

- presence is an atribute, and

- erratic is a value.

The corresponding JSON form for this statement is:

["att\_value",["part","asase\_kokoo","abuo"],"presence","erratic"}

To get this, we have done 2 simple operations:

- enclosed every word in double-quotes, the make it into a JSON (and JavaScript) string; and

- changed every Prolog clause, with a predicate and n arguments, into an array, enclosed in [...], with n+1 elements.

So:

- part(asase\_kokoo,abuo) becomes ["part","asase\_kokoo","abuo"], and

- att\_value(X,presence,erratic) becomes ["att\_value",X,"presence","erratic"], where X is the 'part' array.

Don't worry, the user of webAKT does not have to read or write this format - it is purely for internal use. But I mention it here to convey the very close mapping between the two formats.

**Parsing**

A mentioned at the start, we need to ensure that statements entered by the user (in AKT's formal Prolog syntax) conform to AKT's grammar. This grammar is specified precisely in Chapter 4 of the AKT5 Manual, in Table 4.1, and is included below for completeness.

**The AKT Grammar**

***Table 4.1*** *The definite clause grammar. Terms in bold are reserved terms in the grammar*

*(i.e. words reserved for use by the system); terms starting with a capital letter are variables;* ⇒ *means ‘can take the form of’.*

|  |
| --- |
| FormalSentence ⇒ Statement **if** FormalConditions |
| FormalSentence ⇒ Statement |
| Statement ⇒ Cause **Causes** Effect  where Causes is an element of the set:{**causes1way**,**causes2way**} |
| Statement ⇒ AttributeStatement |
| Statement ⇒ **not**(AttributeStatement) |
| Statement ⇒ **link**(influence,Thing,Thing) |
| Statement ⇒ **link**(Link,Object,Object) |
| Statement ⇒ **link**(Link,ProcessBit,ProcessBit) |
| Statement ⇒ **link**(Link,ProcessBit,Object) |
| Statement ⇒ **comparison**(Attribute,Object,Comparison,Object) |
| FormalConditions ⇒ FormalConditions **and** FormalConditions |
| FormalConditions ⇒ FormalConditions **or** FormalConditions |
| FormalConditions ⇒ Statement |
| FormalConditions ⇒ ActionBit |
| FormalConditions ⇒ ProcessBit |
| AttributeStatement ⇒ **att\_value**(Object,Attribute,Value) |
| AttributeStatement ⇒ **att\_value**(ProcessBit,Attribute,Value) |
| AttributeStatement ⇒ **att\_value**(ActionBit,Attribute,Value) |
| Cause ⇒ AttributeStatement |
| Cause ⇒ ProcessBit |
| Cause ⇒ ActionBit |
| Cause ⇒ Object |
| Cause ⇒ **not**(Cause) |
| ActionBit ⇒ **action**(Action,Object,Object) |
| ActionBit ⇒ **action**(Action,Object) |
| Effect ⇒ AttributeStatement |
| Effect ⇒ ProcessBit |
| Effect ⇒ ActionBit |
| Effect ⇒ **not**(Effect) |
| Process\_bit ⇒ **process**(Process) |
| Process\_bit ⇒ **process**(Object,Process) |
| Process\_bit ⇒ **process**(Object,Process,Object) |
| Thing ⇒ Object |
| Thing ⇒ ProcessBit |
| Attribute ⇒ atom |
| Process ⇒ atom |
| Link ⇒ atom |
| Object ⇒ atom |
| Object ⇒ **part**(Object,Object) |
| Action ⇒ atom |
| Comparison ⇒ Atom where Atom is an element of the set:{**greater\_than**, **less\_than, same\_as**, **different\_from**} |
| Value ⇒ Atom  Where Atom is an element of the set:{**increase**, **decrease**, **change**, **no\_change**} |
| Value ⇒ Atom |
| Value ⇒ Number  Where Number is either a floating point number or an integer |
| Value ⇒ **range**(Value,Value) |

In order for a grammar to be of any use, it needs to be accessed by a parser, which is capable of checking whether a particular statement (or set of statements, actually conforms to the grammar. There are numerous parsing strategies, each of which will have various parsers implemented. I have used a parsing strategy call PEG (Parsing Expression Grammar), and a software tool called PEG.js, which is a avaScript implementation of the PEG strategy. It is somewhat more restrictive than a parser written in Prolog (for example, it does not support bcktracking), but is still a powerful and widely-used tool. It also has the major benefit that, as well as being used for parsing, it can generate an alternative representation of the statements - in or case, JSON. More on that later.

[Aside: Technically, PEG.js is a "parser generator", in that it generates code which is actually used to do the parsing, but we'll skip over that for now.]

**The current version of the webAKT grammar (parser15.peg, April 2023)**

**Statement** = statementHead:**StatementHead** " if " conditions:**Conditions** *{ return ["if", statementHead, conditions] }*

/ **StatementHead**

**StatementHead** = cause:**CauseX** \_ causeType:("causes1way" / "causes2way") \_ effect:**EffectX** *{ return [causeType, cause, effect] }*

/ **AttributeStatementX**

/ "comparison" "(" attribute:**Attribute** \_\_ object1:**Object** \_\_ comparison:**Comparison** \_\_ object2:**Object** *")" { return ['comparison',attribute,object1,comparison,object2]}*

**Conditions** = condition1:**Condition** " and " condition2:**Condition** *{return ["and",condition1,condition2]}*

/ condition1:**Condition** " or " condition2:**Condition** *{return ["or",condition1,condition2]}*

/ **Condition**

**Condition** = **ActionBit**

/ **StatementHead**

/ **ProcessBit**

**AttributeStatementX** = "not" "(" attribute\_statement:**AttributeStatement** ")" *{ return ["not",attribute\_statement]}*

/ **AttributeStatement**

**AttributeStatement** = "att\_value(" object:**Object** "," attribute:**Attribute** "," value:**Value** ")" *{return ['att\_value',object,attribute,value]}*

/ "att\_value(" process:**ProcessBit** \_\_ attribute:**Attribute** \_\_ value:**Value** *")" {return ['att\_value',process,attribute,value]}*

/ "att\_value(" action:**ActionBit** \_\_ attribute:**Attribute** \_\_ value:**Value** ")" *{return ['att\_value',action,attribute,value]}*

**CauseX** = "not" "(" cause:**Cause** ")" *{ return ["not",cause]}*

/ **Cause**

**Cause** = **AttributeStatement**

/ **ProcessBit**

/ **ActionBit**

/ **Object**

**ActionBit** = "action(" action:**Action** \_\_ object1:**Object** \_\_ object2:**Object** ")" *{ return ['action',action,object1,object2]}*

/ "action(" action:**Action** \_\_ object1:**Object** *")" { return ['action',action,object1] }*

**EffectX** = "not" "(" effect:**Effect** ")" *{ return ["not",effect]}*

/ **Effect**

**Effect** = **AttributeStatement**

/ **ProcessBit**

/ **ActionBit**

**ProcessBit** = "process" "(" object1:**Object** \_\_ process:**Process** \_\_ object2:**Object** *")" {return ['process',object1,process,object2]}*

/ "process" "("object:**Object** \_\_ process:**Process** ")" *{ return ['process',object,process]}*

/ "process" "(" process:**Process** *")" {return ['process',process]}*

**Attribute** = attribute:**Atom** *{return attribute}*

**Process** = **Atom**

**Object** = "part(" object1:**Atom** "," object2:**Atom** ")" *{ return ["part",object1,object2] }*

/ **Atom**

**Action** = **Atom**

**Comparison** = ("greater\_than" / "less\_than"/ "same\_as" / "different\_from")

**Value** = ("increase" / "decrease" / "change" / "no\_change")

/ "range" "(" value1:**Value** \_\_ value2:**Value** ")" *{return ['range',value1,value2]}*

/ **Atom**

/ **Number**

**Atom** = chars:[a-zA-Z\_0-9<>%.]\* *{return chars.join("")}*

**Number** = [0-9]\*

**\_** = [ \t\n\r]\* // Zero or more whitespace characters

**\_\_** = "," \_ {return null} // A comma followed by zero or more whitespace characters

**Using PEG.js**

The aim of these notes is to enable anyone else to update the grammar and re-generate the JavaScript file which actually does the parsing. This file needs to replace the current version of it which ships with webAKT. Note that these notes assume you are working with a local version of webAKT (i.e. one where you can edit the HTML and JavaScript files and run webakt.html in you browser as a local file, without an internet connection.

The following is just one pattern of working, using the online version of PEG.js. This is particularly good for testing changes you make to the grammar, since you can immediately see whether the changes work properly. However, it does require you to carefully copy back the grammar into an updated file. Once you get the hang of it, you might find alternative approaches that work well for you.

1. Open PEG.js online version in a new browser tab

<https://pegjs.org/online>

I suggest you keep this tab open as you cycle between editing the grammar, testing it in the online version, and testing it in webAKT itself.

1. Open up the current version of the grammar in your favourite text editor.  
   Currently, this is held in [webakt]/js/parsers/parserNN.peg, where NN is the highest version number.
2. Copy and paste the complete parser code into the left-hand panel of PEG.js online version.  
   Do Select All then Copy for the contents of parserNN.peg, delete the contents of the sample grammar in the window labelled “1. Write your PEG.js grammar”, and paste the copied text in. Don’t worry about the comment lines – these are simply ignored.
3. Check that the message beneath the left-hand panel is green, and says “Parser built successfully”.
4. Enter a syntactically correct test statement into the right-hand panel (the one labelled “2. Test the generated parser with some input”).  
   For example, **att\_value(a,b,c)**
5. Check that the message beneath the right-hand panel is green, and says “Input parsed successfully”
6. Replace the test statement with one that exemplifies the particular grammatical feature you want the parser to handle. i.e. which prompted you to modify the parser in the first place.   
   You should get an orange error message under the right-hand panel. This is to be expected.
7. Edit the grammar text in the left-hand panel to correct or extend the existing grammar rules!!  
   In general, this needs a good understanding of the PEG.js formalism for expressing grammars, as described below. If it’s a simple change, you might be able to work from the pattern of existing rules, but it helps to have a more general understanding.
8. **When you are happy that the parser is working properly**, change the text “module.exports” to “window.parser” in the bottom right-hand panel, labelled “3. Download the parser code”,  
   **This is important.**  The change can be made by editing the generated JavaScript code, but is easier to do at this point.
9. Click on the button labelled “Download parser”.  
   Your browser will show a message for the downloaded file.  
   Save this file in [webakt]/js/parsers, rename it parser.js, to replace the existing parser.js file.
10. **VERY IMPORTANT**. Save the modified grammar code back in [webakt]/js/parsers, as parserNM.peg.  
    … where NM is one more than the NN version you started with.

**Otherwise, your changes will be lost forever**!