School of Informatics, University of Edinburgh

Institute for Communicating and Collaborative Systems

The JAPE riddle generator: technical specification

by

Graeme Ritchie

Informatics Research Report EDI-INF-RR-0158

School of Informatics
http://www.informatics.ed.ac.uk/

February 2003

The JAPE riddle generator: technical specification

Graeme Ritchie

Informatics Research Report EDI-INF-RR-0158

SCHOOL *of* INFORMATICS
Institute for Communicating and Collaborative Systems

February 2003

Abstract :
Although the JAPE riddle generator has attracted significant attention and there are published accounts of its performance, there has been no detailed technical statement of its internal workings. This paper remedies that, by providing formal definitions of the program's data structures, rules and procedures. The most important rules, the schemata, are listed in full in an appendix.

Keywords : humour, riddles, puns, generation

The JAPE riddle generator: technical specification

Graeme Ritchie

School of Informatics, University of Edinburgh

Abstract

Although the JAPE riddle generator has attracted significant attention and there are published accounts of its performance, there has been no detailed technical statement of its internal workings. This paper remedies that, by providing formal definitions of the program's data structures, rules and procedures. The most important rules, the schemata, are listed in full in an appendix.

1 What is JAPE?

JAPE is a computer program, originally constructed (in Prolog (Clocksin and Mellish, 1981)) by Kim Binsted under the supervision of Graeme Ritchie and Helen Pain, in the Department of Artificial Intelligence at the University of Edinburgh. It produces short texts which are intended to be punning riddles (see Section 3 below for more details).

There have been several versions of **JAPE**. **JAPE**-1 was a pilot version, which Binsted wrote in 1993 as part of her MSc (Binsted and Ritchie, 1994, 1997). **JAPE**-2, written mainly during 1995, was the central program in Binsted's PhD thesis (Binsted, 1996). **JAPE**-3 is a revised version of **JAPE**-2, produced by Ritchie in 1999. The motivation for **JAPE**-3 was that **JAPE**-2 had certain limitations in terms of clarity and structure. Although **JAPE**-3 generates riddles in the same way as **JAPE**-2, using largely the same rules, its internal interfaces and data-structures are simpler, clearer and more systematic, and the task of connecting the program to an alternative source of lexical information should be more straightforward (a point verified by Griffiths (2000) in constructing **JAPE**-4, which interfaces **JAPE**-3 to a more flexible dictionary module). That is, **JAPE**-3 is to some extent a *rational reconstruction* (Bundy, 1990; Campbell, 1990) of **JAPE**-2. Most of the account given here is an accurate description of both **JAPE**-2 and **JAPE**-3, but where these systems differ, **JAPE**-3 is described.

The various rule types (described below) are sufficiently well-defined and stable that it is worthwhile distinguishing between the rule-sets and the Prolog code which manipulates them. The former could vary (with addition or modification of schemas, for example), while the latter remains stable. For example, schema `VN` and its supporting rules were added some time after the main development of **JAPE**-3, but required no changes to the underlying rule-interpreters. Hence it may be useful in the future to consider different version-numbering for code and for rules.

2 Motivation for this document

The aim here is to set out a formally precise, implementation-independent account of how **JAPE** generates punning riddles. The reason for doing this is that experimental AI programs are usually under-documented, making it difficult for other researchers to replicate the work, or to know what theoretical claims are actually embodied in the implementation (Ritchie and Hanna, 1984).

The account here presents all the mechanisms which **JAPE** uses in the construction of a riddle, but many of these devices are of no interest to a humour theorist, or to a linguist. All these stages are included because the formalisation aims to be comprehensive and explicit about the **JAPE** model.

Although we are describing an implemented system, we will work at a fairly abstract level, glossing over how some of the more basic mappings (e.g. lexical look-up) might be performed. All the mappings are described here in a direction which follows the flow of processing within **JAPE** (from underlying structures to surface text). Ideally, these mappings should be bijective (two-way), allowing recovery of the semantic structures from the surface forms, but the definitions here are simply one-way.

In the actual implementations, all rules are stated in a notation suitable for use by the program. It can be quite hard to grasp their true import and possible interactions from that representation. We will present an abstract, non-programming, version of these rule-types, in a way which stays as close to the central ideas of the actual **JAPE** program as possible. In defining the various rules, two aspects will generally be stated: the overall functionality of the rule will be stated, and an outline will be given of the internal details of how that functionality is achieved.

An example will be used as a running illustration.

3 What did JAPE do?

JAPE produced short texts which were attempts at punning riddles, such as the following:

(1) How is a nice girl like a sugary bird?
Each is a sweet chick.

(2) What is the difference between leaves and a car? One you brush and rake, the other you rush and brake

(3) What is the difference between a pretty glove and a silent c at? One is a cute mitten, the other is a mute kitten.

(4) What do you call a naked bruin?
A grizzly bare.

(5) What do you call a strange market? A bizarre bazaar.

The best of these were found to be comparable to those published in joke books for children (Binsted, 1996, Chapter 5),(Binsted et al., 1997).

4 Some linguistic constructs assumed

The **JAPE** system assumes a very simplified model of language processing, which can be summarised in the following definitions.

Alphabets and strings. The textual strings created by **JAPE** are orthographic rather than phonetic forms, but the internal computations by the rules make explicit use of both orthographic and phonetic representations (as well as lexical entries). The similarity of sound that is critical to puns is embodied in a number of predicates which relate these levels of representation (see Section 7.1 below). All of these computations occur within the rules, so that the only data-types passed between modules are lexemes and orthographic forms. Therefore, our abstract specification (which does not describe the internal workings of rules) will not mention phonetic forms. We will abstractly assume an abstract alphabet of symbols, with strings of these symbols being called *text strings*.

Syntax. There is a set SYN of *syntactic categories*; these are, informally, labels such as ‘noun’, ‘verb’, etc.

Lexicon. A *lexeme* is a triple $\langle$String, Syn, Sense $\rangle$ where $String$ is a text string and $Syn \in SYN$ is a syntactic category. $Sense$ is simply a unique identity marker to distinguish between separate lexemes with the same string and syntactic category (e.g. *chick* as a small bird or as a slang term for a young woman). This allows individual senses with the same syntax and surface form to enter into different lexical relationships. A *lexicon* is a set of lexemes, together with a set of relations between lexemes (e.g. synonymy, antonymy, hyponymy).

A lexicon L induces a (partial) mapping LEX_L from text strings to sets of lexemes, where $LEX_L(w) = \{E \in L | E = \langle w, Syn, Sense \rangle\}$. The mapping is partial because not every textual fragment corresponds to a word with a syntactic label. Where the lexicon maps a string to a non-singleton set, that string is deemed to have several lexical entries (i.e. to be ambiguous).

Given a lexicon L, a text string T *realises* a sequence $E_1, \ldots, E_n$ of lexemes if T can be segmented into substrings $T_1 \ldots T_n$ such that for each i, $E_i = \langle T_i, Syn_i, Sense_i \rangle$.

Given an alphabet and a lexicon for strings over that alphabet, there are various possible conditions that can be true or false of a lexeme, a string or some combination of these. For example, if the syntactic label NOUN occurs in the lexicon, then there is an associated predicate *noun* which can be applied to a lexeme; if the lexicon contains a relation SYNONYM, then there is an associated predicate *synonym* which can be applied to two lexemes; if a string T realises a lexical sequence $E_1, \ldots, E_m$, then there is a predicate, which we can call *written-form*, such that *written-form*($\langle E_1, \ldots, E_m \rangle, T$). In this way, choice of an alphabet and lexicon indirectly defines a collection of predicates (unary and binary) which can be used to state conditions on text strings and lexemes. We shall call these the *lexically definable predicates*.

This version of the lexicon is less sophisticated than that typically discussed within linguistics. It represents an association between information and text strings, but the manner of computing this association is of no interest. Therefore, it is not essential to include any form of morphological processing to ensure that various forms of a verb are all related to the same lexical entry in some way. In our abstract version, *walk* and *walks* have separate entries.

5 An overview of JAPE's mechanisms

Before presenting mathematical definitions of structures and rules, we will briefly summarise the overall picture (see Figure 1).

JAPE's rules were of four different types, performing different tasks in the overall model: *schemata* (defining configurations of lexemes underlying riddles), *sentence forms* (patterns of fixed text with slots for further text strings to be inserted), *templates* (defining conditions for particular items to be inserted into sentence forms), and *SAD generation rules* (which create abstract linguistic structures from lexemes). The term "SAD", originally short for *small adequate description*, is used here for compatibility with Binsted's terminology. **JAPE**-3 contains 15 schemata, 18 SADrules, 9 templates and 9 sentence forms.

There is a *lexicon* which contains *lexemes*, where a lexeme is a cluster of linguistic information about one sense of a noun, verb, or adjective (see Section4 above).

A schema contains some *lexical preconditions*, which describe, in terms of their properties and relationships, a configuration of lexemes which would form the central underlying structure of a riddle. A schema also contains an *output specification*, which indicates which of these lexemes are to be used in the surface form, very roughly how they are to be expressed, and any abstract relations which will hold between the constituents which express these lexemes.

The SAD generator acts on the information supplied by a schema, fleshing out the lexemes (or sequences of lexemes) into SADs, which are more detailed descriptions of

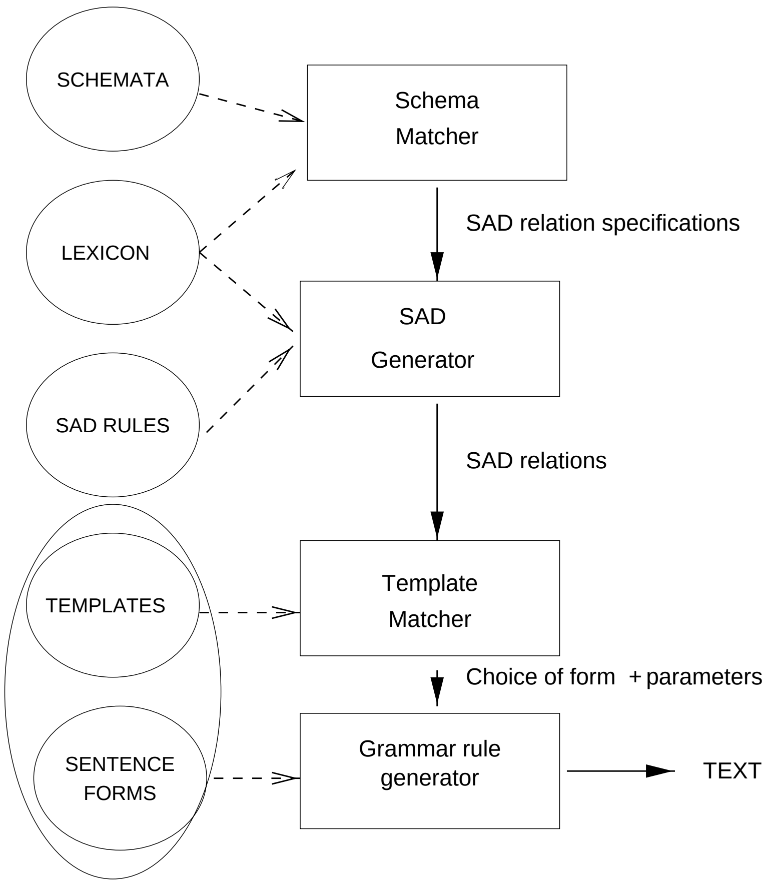

Figure 1: An overview of JAPE

linguistic structures. For this purpose, it contains various *SAD rules*, which are rules of thumb for creating linguistic data based on the outline information given by the schema. Each rule has some pre-conditions (similar in form to the preconditions in a schema) which take the lexemes provided by the schema as parameters. The remaining part of a SAD rule shows how, using some of the items mentioned in its preconditions, a linguistic structure is to be formed.

The information passed to the SAD generator from the schema processing indicates not only what lexemes are to be used to create constituents, but also what the relations are to be between these constituents once created. The **JAPE** system made use of only a few such relations, principally one which can be glossed as "describes the same concept". These equation-like structures – *SAD relations* – are passed on to the template stage.

A template contains some conditions describing the internal structure of the SAD relations it receives, and a computation to extract information (typically lexemes and text strings) from the items within SAD relations; these items are the constituent structures drafted by the SAD-generator. This extracted data is then paired with the name of a sentence form (a pattern of pieces of text and slots to be filled with linguistic constituents) and passed to a generator which carries out the last stage of text creation.

The reason that there are so many schemata is that different lexical preconditions must be available for each variation on how the comparable strings are to be constructed. For example, jokes (6), (7) and (8) use, intuitively, the same punchline structure (the initial part of an established word or phrase is replaced by a phonetically similar item), but because grammatically different forms (compound noun or word) and different notions of phonetic similarity (homophony or rhyming) are involved, separate schemata are used.

(6) What kind of murderer has fibre?
A cereal killer.

(7) What do you call a bath tour?
A tub crawl.

(8) What do you call a depressed train? A lowcomotive.

Similarly, output specifications vary according to the broad linguistic structure of the set-up question. With hindsight, it might be possible to generalise across similar schemata, and delegate the finer details of the punchline-construction to another module or rule-set.

6 A formalisation of JAPE

6.1 Types of objects and structures

6.1.1 Basic units

For the more basic structures (alphabet, text strings, lexicon, etc.) we shall rely on the definitions in Section 4 earlier. Simple words will be used here to indicate lexemes, but abstractly a lexeme is a structure within a lexicon, not a printed word. Notions such as *rhyming* and *alliteration* can be regarded as lexically definable predicates, based on identity of pronunciation of substrings of the text strings being compared.

In the examples below, we shall write lexically definable predicates in standard predicate notation, indicating the syntactic category of a lexeme as a function applied to it (e.g. NOUN(`bazaar`)) and lexical relations as predicates applied to pairs of lexemes (e.g. SYNONYM(`bizarre`,`strange`)).

6.1.2 SADs

A *SAD* is a relatively abstract characterisation of a text string, not explicitly showing the actual text to be used but indicating the important attributes of that constituent. For example, for a text string which (informally speaking) represents a transitive verb phrase, a SAD might indicate the verb and the head noun of the object. These are a rather crude and **JAPE**-specific form of linguistic structure, whose essential import is defined solely by how they are manipulated by the various rules. That is, they have no empirical justification beyond their role as an internal construct within the model.

Example: The following (from the **JAPE** program) could be glossed as "that which can be brushed and can be raked", with `NP` indicating a parameter to be instantiated.

```
joint_object(NP, brush, rake)
```

6.1.3 SAD specifications

The information from which the description generator works is a *small adequate description specification*, or "SAD-spec" for short. A SAD-spec consists of two parts: a *tag* from a small built-in set of symbolic labels, and a tuple of lexemes.

Example:

```
(shared_object,[brush, rake])
```

Informally, the tag is used by the generator to select a rule for the computation, and the lexemes are the arguments on which the rule is to work.

6.1.4 SAD relations

Another non-primitive object used within **JAPE** is the *SAD relation*. This consists of a triple of items: the first is a label indicating one of a small set of relation-names used within **JAPE**, and each of the other two items can be either a text string or a SAD. Informally, this means that later rules are to treat the two items as if the relation named by the label held between them. (For almost all of the jokes generated by **JAPE**, the relation label is the mnemonic `same`). In all the riddles covered by **JAPE**, it happens that all such pairings consist of an item which is to appear in the question part of the text paired with an item which is to appear in the answer part. As with SADs, these abstract constructs are meaningful only as internal representations within the system, and are not directly accessible to intuitive judgements about appropriateness.

Example: The following (from the **JAPE** program) indicates that a phrase expressing the lexeme `leaves` directly describes roughly the same concept as *that which can be brushed and raked*.

```
<same, same_lexeme(NP,leaves), joint_object(NP,brush,rake)>
```

6.1.5 SAD equations

A *SAD equation* is a sketch of a SAD relation, in that it has a similar overall structure (a relation-name and two other items), but instead of SADs it contains SAD-specs. A SAD equation acts as a blueprint for the building of a SAD relation, since the SAD generator (see below) creates a SAD relation by replacing each of the SAD-specs with the result of applying the generation rules to that spec.

Example:

```
<same, (shared_object, [brush, rake]), (conjoin_inacts, [brush, rake])>
```

These structures are used in the output conditions of schemata (below).

6.2 Types of rule

6.2.1 Schemata

A *schema* consists of two parts. The first is a set of *preconditions*, stated in terms of lexically definable predicates. In all the **JAPE** schema so far, the conditions are of the logical form

$$\exists X_1, \ldots, X_n : P_1 \ \wedge \ P_2 \ldots \ \wedge \ P_m$$

where the various P_i are propositions involving some or all of the items X_j. That is, the preconditions always test for the *existence* of a set of lexemes and text strings related in particular ways. The preconditions take account of information encoded in the lexicon in order to determine their results. If the preconditions are satisfied by some particular set of values $A_1, \ldots, A_n$, then these values are available for use in the second part of the schema, the *output specification*. (That is, the formal mechanism borrows heavily from the treatment of variables in Prolog, rather than relying on pure first-order logic.) Thus the preconditions are formally a mapping from the supporting information (the lexicon) to a sequence of values (for the various variables mentioned in the existential quantifiers at the start). In the case where the preconditions are not satisfied by any items, the mapping yields $false$.

The second part of a schema is a mapping from the values which satisfied the preconditions (i.e. in effect the parameters of the schema) to a set of k SAD equations (usually $k = 1$ or $k = 2$).

Example: The schema known mnemonically as `brushrake` has the following preconditions (where all the capitalised words indicate variables which are implicitly existentially quantified and the commas indicate logical conjunction of the propositions they separate):

```
spoonerizes([WordA, WordD],[WordB,WordC]),
written_form([LexA], [WordA]),
written_form([LexB], [WordB]),
written_form([LexC], [WordC]),
written_form([LexD], [WordD]),
verb(LexA),
verb(LexB),
verb(LexC),
verb(LexD)
```

This schema has the output specification:

```
<same, (shared_object,[LexA, LexD]),
       (conjoin_inacts, [LexA, LexD])>,
<same, (shared_object,[LexB, LexC]),
       (conjoin_inacts, [LexB, LexC])>
```

In the output specification, the uninstantiated variables (`LexA`, etc.) embody the mapping from schema parameters (the values found by the preconditions) to SAD equations: instantiating these variables forms the actual SAD equations.

6.2.2 The SAD generator

The SAD generator is a mapping from SAD-specs to SADs. Informally, the resulting SAD should be made up of linguistic properties and relations that a entity might have if it was in some way created from the lexemes in the tuple. The tag is a way for the schema (which prepares the input for the generator) to guide the building of the SAD, as there might be more than one way to construct a SAD from the lexemes; for example, given a noun and a verb, the noun could be the head of the verb's subject or of the verb's object.

The main part of the SAD generator is a set of *SAD-generation rules*. Each rule is a mapping from a sequence of lexemes to a SAD. Internally, a rule consists of a set of preconditions and an output specification. (That is, it has the same broad structure as a schema, but differs in the way that it receives its input and in the type of output structures it creates). The SAD generator also contains a mapping from the set of guiding tags to a set of sets of these SAD rules. That is, given a particular symbolic tag (which originates in a schema's output specifications) there is a candidate set of rules for carrying out the generation.

The preconditions (unlike those of a schema) start with some values already selected, in the form of the lexeme sequence supplied in the SAD-spec. The existence (typically in the lexicon) of other related values will be tested, and (as with schema preconditions) all the values found are available for use by the rule's output specification. Hence the preconditions are a mapping from the input lexemes, together with the lexicon, to a sequence of lexemes (or possibly text strings, although this does not happen in the **JAPE** rules at present).

The rule's output specification maps some subset of the sequence of lexemes which satisfied the preconditions into a SAD.

Formally, an n-m-ary SAD rule is a pair (C, S). C represents the preconditions, a mapping from a sequence of n lexemes (and the lexicon) to either $false$ or a sequence of m lexemes. S is a mapping from m lexemes to a SAD.

The whole SAD generator SG is then of the form $(select, \{(C_i, S_i)\})$ where $select$ is the mapping from tags to subsets of $\{(C_i, S_i)\}$, the set of rules. Regarded as a mapping from SAD-specs to SADs (relative to a lexicon LEX), $SG((tag, \langle l_1, \ldots, l_n\rangle), LEX) = V$ iff there is a rule (C, S) in SG such that $(C, S) \in select(tag)$ and $C(\langle l_1, \ldots, l_n\rangle, LEX) = \langle v_1, \ldots, v_m\rangle$ and $S(\langle v_1, \ldots, v_m\rangle) = V$.

Example: The $select$ function operating on the tag `conjoin_inacts` will yield a set of SAD-rules which includes the following one.

```
Input Parameters:  [V1, V2],
Preconditions:
   inact_verb(X,V1),
   inact_verb(X,V2)
Output specification:
  [joint_object(NP, V1, V2)]
```

6.2.3 Templates

A template is a mapping from one or more arguments, each of which is a SAD relation (see above), to a sentence form and a set of parameters for that sentence form. It achieves this by checking its arguments for suitability, then manipulating the arguments (e.g. extracting subparts, or combining arguments) so as to form items suitable for use by the particular sentence form. The written text can then be created directly by this sentence form.

More formally, an k-ary template (over an alphabet A) consists of three parts: k conditions, $P_1, \ldots, P_k$, to be applied to the template's k arguments; an n-ary sentence form sf over A; a function F which maps the template's k arguments into a tuple of n values suitable as input to sf.

Thus, a template of the form $(P_1, \ldots, P_k, sf, F)$, given arguments $A_1, \ldots, A_k$ such that $P_i(A_i)$ is true for each $1 \leq i \leq k$, will pass on the n-tuple of values $F(A_1, \ldots, A_k)$ to the sentence form sf.

The conditions in a template test only the internal structure of the arguments and are independent of the lexicon or any other information source outside the parameter values supplied. That is, they perform local structure-matching, rather than any form of inference or verification against a database.

Example: The following is an English summary of the computation carried out by one of the templates, which is coded in Prolog within **JAPE**:

```
Inputs: SAD-Reln1, SAD-Reln2
Conditions:
    Both SAD-Reln1, SAD-Reln2 should have relation ‘same’.
    Each of these relations should have an ‘answer’ side
     which is a SAD of the general form:
        joint_object(X, V1, V2).
Extraction of information:
    Extract suitable lexemes for making noun phrases from
     the SADs on the ‘question’ sides of the SAD-relations;
     call these NPQ1 and NPQ2 respectively.
    Extract the verbs (2nd and 3rd arguments) from the SADs
     in the ‘answer’ sides;
     call these  V11, V12 (from SAD-Reln1), V21, V22 (from
     SAD-Reln2).
Results:
    Sentence form : vvcompare
    Arguments for sentence form:
     NPQ1, NPQ2, V11, V12, V21, V22
```

6.2.4 Sentence forms

Informally speaking, a *sentence form* is a pattern of written strings and slots where material can be filled in to create partly “canned” text. The data on which the “fillers” are to be based are regarded as parameters of the sentence form, and the number of parameters might differ from the number of blank slots in the sentence form’s text, since, for example, two parameter values might combine to make up the filler for a single slot, or one parameter value might be duplicated into more than one slot. The possible parameter types are those appropriate for the rest of the **JAPE** architecture, which happen to be sequences of lexemes or text strings. Thus a sentence form is a mapping from a tuple of input arguments to a surface string.

More precisely, an *n-ary sentence-form* over an alphabet A consists of a tuple of m strings from A (fragments of text), and for each of the $(m - 1)$ gaps between these surface texts (and for the beginning and end of the tuple) there is a mapping from n parameters to a surface text. Informally, the jth function f_j indicates what surface text (computed from the n parameter values) should be inserted between fragments j and $(j + 1)$ of the tuple, with the 0th function f_0 defining anything that is to be put at the start, f_m defining any text to be placed at the end.

In this way, a sentence form $(\langle s_1, \ldots, s_m\rangle, \langle f_0, \ldots, f_m\rangle)$ defines a mapping from n items $\langle p_1, \ldots, p_n\rangle$ (where each item p_i is either a sequence of lexemes or a surface text) to a surface text of the form:

$$f_0(p_1, \ldots, p_n)s_1 f_1(p_1, \ldots, p_n) \ldots s_m f_m(p_1, \ldots, p_n)$$

Example:

```
Sentence form : vvcompare

Input parameters: NP1, NP2, Verb11, Verb12, Verb21, Verb22

Strings: ‘What is the difference between’,
         ‘and’
         ‘?’
         ‘One you’
         ‘and’
         ‘, the other you’
         ‘and’
Functions: f0 = empty string;
           f1 = a noun phrase string from NP1;
           f2 = a noun phrase string from NP2;
           f3 = empty string;
           f4 = surface verb form of Verb11;
           f5 = surface verb form of Verb12;
           f6 = surface verb form of Verb21;
           f7 = surface verb form of Verb22.
```

This is one case where the actual program code – a Definite Clause Grammar rule (Pereira and Warren, 1980) in Prolog notation – may be more perspicuous than the version above:

```
vvcompare(NP1, NP2, Verb11, Verb12, Verb21, Verb22) -->
   [’What’, is, the, difference, between],
   np(NP1), [and], np(NP2), [’?’],
   [’One’, you], verb(Verb11), [and], verb(Verb12), [’,’],
   [the, other, you], verb(Verb21), [and], verb(Verb22).
```

6.3 The whole system

Combining all the formal entities from the past few pages, we have a statement that a *JAPE riddle system* is 6-tuple consisting of:

An alphabet A. See Section 4 above.

A lexicon LEX. See Section 4 above.

A set SF of sentence forms over A.

A set T of templates. Each is of the form $(P_1, \ldots, P_k, sf, F))$ (for some k, F and some $sf \in SF$).

A set S of schemata. These will have preconditions stated in terms of the lexically definable predicates.

A SAD generator SG. (of the form $(select, \{(C_i, S_i)\}))$ The preconditions of each rule will be stated in terms of the lexically definable predicates.

6.4 Characterising riddles

If we assume the various formalisations given above for the structures and rules involved in a **JAPE** riddle system, then we can now consider a formal definition of what counts as a well-formed riddle according to such a rule set.

Given a **JAPE** riddle system:

$$\langle (A, \approx), LEX, SF, T, S, (select, \{(C_i, S_i)\}) \rangle$$

a text σ is a *JAPE riddle* if there is an n-ary schema $s \in S$ of the form $(CONDS, OUTPUT)$ and an m-ary template $t \in T$ of the form $(\langle P_1, \ldots, P_m \rangle, sf, F)$ such that there are values $v_1, \ldots, v_n$ such that:

- $CONDS(LEX) = \langle v_1, \ldots, v_n \rangle$
- $OUTPUT(v_1, \ldots, v_n) = \{(Reln_j, LHS_j, RHS_j) \mid 1 \leq j \leq m\}$
- there are m SAD relations $\langle E_1, \ldots, E_m \rangle$ such that for $1 \leq j \leq m$

  $$E_j = (Reln_j, SG(LHS_j, LEX), SG(RHS_j, LEX)),$$

  and such that

  - $P_i(E_i)$ is true for all $1 \leq i \leq m$
  - $F(E_1, \ldots, E_m) = \langle w_1, \ldots, w_k \rangle$
  - $sf(w_1, \ldots, w_k) = \sigma$.

7 JAPE schemata

7.1 Predicates used

Here is a list of the Prolog predicates used in the Lexical Preconditions of **JAPE**-3 schemata, together with informal definitions. ("`TString`" indicates a "text string" in the sense of Section 4 earlier.)

`noun(Lexeme):` The syntactic category of `Lexeme` is `noun`. Similar definitions hold for `verb`, `adj` and `noun_phrase` (where the latter means "noun-noun compound").

`homophone(TStringA, TStringB):` These two surface strings sound the same.

`written_form(LexemeList, TString):` The lexemes in `LexemeList` are realised at the surface by `TString`.

`synonym(Lexeme1, Lexeme2):` These two lexemes mean the same.

`component_lexemes(PhraseLexeme, LexemeA, LexemeB):` The written form of `PhraseLexeme` is the same as the concatenation of the written forms of `LexemeA` and `LexemeB`.

`rhyme(TString1, TString2):` These text strings end in phonetically similar/identical material.

`alternate_meaning(Lexeme1, Lexeme2):` These two lexemes are different senses for the same text string.

`spoonerize([TStringA, TStringD],[TStringB, TStringC]):` The initial parts of text strings `TStringA` and `TStringC` are phonetically similar, as are those of `TStringB` and `TStringD`.

`phon_form(TString, PhonForm):` `TString` has a phonetic form `PhonForm`.

`match(TString1, PhonForm1, PhonForm2, TString2):` `PhonForm1` is the pronunciation of `TString1`, `PhonForm2 is` an initial substring of `PhonForm1` such that `TString2` is the corresponding initial substring of `TString1`.

`subspell(TString1, TString2, TString3):` `TString1` is the concatenation of `TString2` and `TString3`.

`subphons(PhonForm1, PhonForm2, PhonForm3):` Phonetic form `Phonform1` is the concatenation of `PhonForm2` and `PhonForm3`.

8 In conclusion

We have tried to ensure that the workings of **JAPE** are stated explicitly, so that other workers can assess or replicate them.

Any queries should be directed to `Graeme.Ritchie@ed.ac.uk`.

Acknowledgements

The conversion of **JAPE**-2 to **JAPE**-3 and the first drafting of the abstract specification were carried out while the author was a Visiting Researcher in the Alfa Informatica group at the University of Amsterdam. The final version of this specification was prepared while the author was supported by a research fellowship from the Leverhulme Trust.

Appendix: Schema definitions

Here we present cleaned up and restructured versions of the schemata listed in Appendix J of Binsted (1996), copied directly from the **JAPE**-3 program. The names in Binsted's thesis have been kept, for ease of cross-referencing. Annotations at the right hand side, preceded by a semi-colon, are not part of the schema, but are a sketch of how the variables in the schema would be bound in order to generate the illustrative example given directly below the schema.

Prolog notation is used throughout: upper-case identifiers represent variables to be instantiated, and `\+` represents negation ("not").

The `Lexical Preconditions` are grouped within parentheses, with propositions separated by commas denoting conjunction. The notation used in the `Output specification` field is as follows. The field contains a list (delimited by square brackets), where each element in the list is a term representing an (uninstantiated) SAD equation (see Section 6.1.5 above). This term (of which the functor is `sad_reln`) contains takes three arguments: the first is the name of a SAD-relation (usually `same`) and the other two arguments are terms (with functor `sad_from`) each of which will, once its variables are instantiated, act as input to the SAD-generator. That is, the functor `sad_from` is used just to wrap up the tag and data for the SAD-generator to work on.

In the implementation, there was a third field to a schema, after the Lexical Preconditions and Output Specification, namely the Keywords. These were a list of the variable names from the Lexical Preconditions that would become instantiated and whose values would appear (in their written forms) in the surface text. This information was used for various testing and evaluation purposes, but did not affect the generation of the jokes, and so is omitted here.

Schema `VN` is rather odd. **JAPE**-2 generated jokes of this type, although in a slightly unprincipled fashion. In the conversion to **JAPE**-3, generation of this class of joke required the inclusion of this schema, three SAD-generator rules, one template and one sentence form. These jokes are different from others handled by **JAPE**, in that a particular word (*sea*, *sale*) must appear in *both* question and answer, and no synonym can be substituted. Hence the schema must stipulate rather more than usual about the surface form, and the work of the text generator is then highly constrained by the very specific brief it is given.

Lotus

```
Lexical preconditions:
  (noun_phrase(NPLex),                   % NPLex = serial killer
   component_lexemes(NPLex, LexA, LexB), % LexA = serial, LexB = killer
   written_form([LexA], WordA),          % WordA = 'serial'
   homophone(WordA, HomWord),            % HomWord = 'cereal'
   written_form([HomLex], HomWord)       % HomLex = cereal
  )
Output specification:
  [sad_reln(same,
             sad_from(share_properties, [HomLex, NPLex]),  % in Q
             sad_from(make_phrase, [HomLex, LexB]))        % in A
  ]
```

What kind of murderer has fibre? A cereal killer.

Bazaar

```
Lexical constraints:
  (noun(LexB),                        % LexB = bazaar
   written_form([LexB], WordB),       % WordB = ‘bazaar’
   homophone(WordA, WordB),           % WordA = ‘bizarre’,
   written_form([LexA], WordA)        % LexA = bizarre%
  )
Output specifications:
  [sad_reln(same, sad_from(share_properties,[LexA,LexB]),  % in Q
                  sad_from(make_phrase, [LexA, LexB]))     % in A
  ]
```

What do you call a strange market? A bizarre bazaar.

Negcomp

```
Lexical preconditions:
  (
   spoonerize([WordA, WordD],[WordB, WordC]),
               % [[‘cute’,‘mitten’][‘mute’,‘kitten]]
   written_form([LexA], [WordA]),
   written_form([LexB], [WordB]),
   written_form([LexC], [WordC]),
   written_form([LexD], [WordD]),
   adj(LexA),           %  LexA = cute
   adj(LexB),           %  LexB = mute
   noun(LexC),          %  LexC = kitten
   noun(LexD)           %  LexD = mitten
  )
Output specification:
  [sad_reln(same,
            sad_from(share_properties, [LexA, LexD]),  % in Q
            sad_from(make_class_spec, [LexA, LexD])) , % in A
   sad_reln(same,
            sad_from(share_properties, [LexB, LexC]),  % in Q
            sad_from(make_class_spec, [LexB, LexC]) )  % in A
  ]
```

What’s the difference between a pretty glove and a silent cat? One’s a cute mitten, the other’s a mute kitten.

Phonsub

```
Lexical preconditions:
  (phon_form(Word, PhonForm),
          % Word = ‘locomotive’, PhonForm= /locomotive/
   subphons(PhonForm, SubPhonForm, Remainder),
                   % SubPhonForm=/lo/,Remainder=/comotive/
   phon_form(SubWord, SubPhonForm),   % SubWord = ‘low’
   Word \== SubWord,                  % ‘locomotive’ =/= ‘low’
   match(Word, PhonForm, Remainder, RemSpell),
       % ‘locomotive’, /locomotive/, /comotive/,‘comotive’
   subspell(NewWord, SubWord, RemSpell),  % NewWord = ‘lowcomotive’
   written_form([SubLex], [SubWord]),     % SubLex = low
   written_form([Lex], [Word]),       % Lex = locomotive
   \+ synonym(Lex, SubLex),           % locomotive & low not synonyms
   noun(Lex),                         % locomotive a noun
   Word \== NewWord)                 % ‘locomotive’ =/= ‘lowcomotive’
```

```
Output specification:
   [sad_reln(same,
        sad_from(share_properties, [SubLex, Lex]),  % in Q
        [NewWord]) % in A
   ]
```

What do you call a depressed train? A low-comotive.

Hopchew

```
Lexical preconditions:
  (spoonerizes([WordA, WordD],[WordB,WordC]),
   written_form([LexA], [WordA]),
   written_form([LexB], [WordB]),
   written_form([LexC], [WordC]),
   written_form([LexD], [WordD]),
   verb(LexA),
   verb(LexB),
   verb(LexC),
   verb(LexD))

Output specification:
   [sad_reln(same,
        sad_from(shared_subject,[LexA, LexD]),   % in Q
        sad_from(conjoin_acts, [LexA, LexD])),   % in A
    sad_reln(same,
        sad_from(shared_subject,[LexB, LexC]),   % in Q
        sad_from(conjoin_acts, [LexB, LexC]))    % in A
   ]
```

What’s the difference between a hungry kangaroo and a lumberjack?
One hops and chews, the other chops and hews.

Brushrake

```
Lexical preconditions:
        (
         spoonerizes([WordA, WordD],[WordB,WordC]),
        % WordA = ‘brush’, WordD = ‘rake’, WordB = ‘rush’, WordC = ‘brake’
         written_form([LexA], [WordA]),
         written_form([LexB], [WordB]),
         written_form([LexC], [WordC]),
         written_form([LexD], [WordD]),
         verb(LexA),
         verb(LexB),
         verb(LexC),
         verb(LexD)
        ),
Output specification:
   [sad_reln(same,
              sad_from(shared_object,[LexA, LexD]), % in Q
              sad_from(conjoin_inacts, [LexA, LexD])),  % in A
    sad_reln(same,
              sad_from(shared_object,[LexB, LexC]), % in Q
              sad_from(conjoin_inacts, [LexB, LexC])) % in A
   ],
```

What’s the difference between leaves and a car? One you brush and rake, the other you rush and brake.

Poscomp

```
 Lexical preconditions:
 (adj(LexA),                         % sweet1 is an Adj
  adj(LexB),                         % sweet2 is an Adj
  alternate_meaning(LexA, LexB),     % LexA= sweet1, LexB = sweet2
  noun(LexC),                        % chick1 is a Noun
  noun(LexD),                        % chick2 is a Noun
  alternate_meaning(LexC, LexD),     % LexC= chick1, LexD = chick2
  written_form([LexA], [WordA]),     % WordA = ‘sweet’
  written_form([LexD], [WordD]),     % WordD = ‘chick’
  WordA \== WordD                    % to prevent, e.g. ‘absinth absinth’
  )
Output specification:
   [sad_reln(same,
        sad_from(share_properties, [LexA, LexC]),  % in Q
        sad_from(make_phrase, [LexA, LexC])),      % in A
    sad_reln(same,
        sad_from(share_properties, [LexB, LexD]),  % in Q
        sad_from(make_phrase, [LexA, LexC]))      % in A
   ]
```

How’s a nice girl like a sugary bird? They’re both sweet chicks.

Rhyming_lotus

```
Lexical preconditions:
  (noun_phrase(NPLex),                        % NPLex = pub crawl
   component_lexemes(NPLex, LexA, LexB),      % LexA = pub, LexB = crawl
   written_form([LexA], [WordA]),             %  WordA = ‘pub’
   rhyme([WordA], [RhymWord]),                % RhymWord = ‘tub’
   written_form([RhymLex],[RhymWord])         %  RhymLex = tub
  )

Output specification:
  [sad_reln(same,
        sad_from(share_properties, [RhymLex, NPLex]), % in Q
        sad_from(make_phrase, [RhymLex, LexB]))]      % in A
```

What do you call a bath tour? a tub crawl.

Elan1

```
Lexical Preconditions:
 (noun_phrase(NPLex),                    %  NPLex = grizzly bear
  component_lexemes(NPLex, LexA, LexB), % LexA = grizzly, LexB = bear
  written_form([LexB], [WordB]),        % WordB = ‘bear’
  homophone([WordB], [HomWord]),        % HomWord = ‘bare’
  written_form([HomLex], [HomWord]),    % HomLex = bare
  noun(HomLex)
 )

Output specification:
   [sad_reln(same,
        sad_from(share_properties, [HomLex, LexB]), % in Q
        sad_from(make_phrase, [LexA, HomLex]))     % Answer
   ]
```

What do you call a naked bruin? A grizzly bare.

Elan2

```
Lexical Preconditions:
  (noun_phrase(NPLex),                  %  NPLex = grizzly bear
  component_lexemes(NPLex, LexA, LexB), % LexA = grizzly, LexB = bear
  written_form([LexB], [WordB]),        % WordB = ‘bear’
  homophone([WordB], [HomWord]),        % HomWord = ‘bare’
  written_form([HomLex], [HomWord]),    % HomLex = bare
  noun(HomLex)
  )

Output specification:
   [sad_reln(same,
        sad_from(share_properties, [NPLex, HomLex]), % in Q
        sad_from(make_phrase, [LexA, HomLex]))       % in A
   ]
```

What do you call a nude that has claws? A grizzly bare.

Coatshed

```
Lexical preconditions:
  (
   verb(LexA),                     % shed1 is a Verb
   noun(LexB),                     % shed2 is a Noun
   alternate_meaning(LexA, LexB),  % LexA = shed1, LexB = shed2
   verb(LexC),                     % coat1 is a Verb
   noun(LexD),                     % coat2 is a Noun
   alternate_meaning(LexC, LexD),   % LexC = coat1, LexD = coat2
   \+ alternate_meaning(LexB, LexD) % don’t want "coat a coat"
  )

Output specification:
   [sad_reln(same,
        sad_from(possible_subject,[LexA, LexD]),  % in Q
        sad_from(subject_object, [LexA, LexD])),  % in A
    sad_reln(same,
        sad_from(possible_subject,[LexC, LexB]),  % in Q
        sad_from(subject_object, [LexC, LexB]))   % in A
   ]
```

What’s the difference between a hairy dog and a painter? One sheds his coat and one coats his shed.

Jumper

```
Lexical Preconditions:
  (noun_phrase(NPLex),                   %  NPLex = grizzly bear
  component_lexemes(NPLex, LexA, LexB) , % LexA = grizzly, LexB = bear
  written_form([LexB], [WordB]),         % WordB = ‘bear’
  homophone([WordB], [HomWord]),         % HomWord = ‘bare’
  written_form([HomLex], [HomWord]),     % HomLex = bare
  noun(HomLex)
  )

Output specification:
   [sad_reln(same,
        sad_from(share_properties, [LexA, HomLex]), % in Q
        sad_from(make_phrase, [LexA, HomLex]))]     % in A
```

What do you call a grumpy nude? A grizzly bare.

Woolly

```
Lexical preconditions:
  (noun_phrase(NPLex),                   % NPLex = serial killer
   component_lexemes(NPLex, LexA, LexB), % LexA = serial, LexB = killer
   written_form([LexA], WordA),          % WordA = ‘serial’
   homophone(WordA, HomWord),            % HomWord = ‘cereal’
   written_form([HomLex], HomWord)       % HomLex = cereal
  )

Output specification:
   [sad_reln(same,
        sad_from(share_properties, [HomLex, LexB]),   % in Q
        sad_from(make_phrase, [HomLex, LexB]))   ]    % in A
```

What do you get when you cross a murderer with a breakfast food? A cereal killer.

Double pun

```
Lexical preconditions:
 (noun_phrase(NPLex),
  component_lexemes(NPLex, LexA, LexB),
  written_form([LexA], [WordA]),
  written_form([LexB], [WordB]),
  homophone([WordA], [HomWordA]),
  homophone([WordB], [HomWordB]),
  written_form([HomLexA], [HomWordA]),
  written_form([HomLexB], [HomWordB]),
  noun(HomLexB)
 )

Output specification:
 [sad_reln(same,
         sad_from(share_properties, [HomLexA, HomLexB]),   % in Q
         sad_from(make_phrase, [HomLexA, HomLexB]))]       % in A
```

What do you call a gory nude? A grisly bare.

VN

```
Lexical preconditions:
( noun(LexA),  verb(LexB),
  written_form([LexA], WordA), % WordA = ‘sea’
  written_form([LexB], WordB), % WordB = ‘sail’
  homophone(WordA, WordC),     % ‘sea’, ‘see’
  homophone(WordB, WordD),     % ‘sail’, ‘sale’
  written_form([LexC], WordC), % WordC = ‘see’
  written_form([LexD], WordD), % WordD = ‘sale’
  verb(LexC),  noun(LexD)
)

Output specification:
  [sad_reln(property,
      sad_from(this_lexeme, [LexA]),
      sad_from(possible_action,[LexB, LexA])),
   sad_reln(property,
      sad_from(this_lexeme, [LexD]),
      sad_from(not_possible_action,[LexC, LexD]))]
```

What’s the difference between a sea and a sale? You can sail a sea, but you can’t see a sale.

References

Kim Binsted. *Machine humour: An implemented model of puns*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1996.

Kim Binsted, Helen Pain, and Graeme Ritchie. Children's evaluation of computer-generated punning riddles. *Pragmatics and Cognition*, 5(2):309–358, 1997.

Kim Binsted and Graeme Ritchie. An implemented model of punning riddles. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, USA, 1994.

Kim Binsted and Graeme Ritchie. Computational rules for generating punning riddles. *HUMOR*, 10(1):25–76, 1997.

Alan Bundy. What kind of field is AI? In Partridge and Wilks (1990), pages 215–222.

John Campbell. Three novelties of AI: theories, programs and rational reconstructions. In Partridge and Wilks (1990), pages 237–246.

William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer Verlag, Berlin, 1981.

Paul Griffiths. Lexical support for joke generation. Master's thesis, Division of Informatics, University of Edinburgh, Edinburgh, Scotland, 2000.

Derek Partridge and Yorick Wilks, editors. *The foundations of artificial intelligence*. Cambridge University Press, Cambridge, 1990.

F. Pereira and D. H. D. Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

G. D. Ritchie and F. K. Hanna. AM : A case study in AI methodology. *Artificial Intelligence*, 23:249–268, 1984.