SCI Parser

Programmer's Reference

by Pablo Ghenis

July 21, 1988 9:56:56 am

Contents

Introduction

Purpose

The parser is the part of SCI that accepts sentences from the user and allows game coders to specify how to respond to it. For example "get the diamond" is an acceptable sentence for a user to type, and the coder can recognize it with a "spec" such as 'get/diamond', which is followed by SCI code to be executed IF this sentences is entered.

There are three functional blocks in the parser: the sentence parser, the spec parser and the matcher. The sentence parser takes the words typed by the user and generates a "tree" that describes the sentence's structure using traditional grammar rules. The spec parser accepts "specs" and also generates tree structures. A special syntax is provided for specs since they are more versatile than simple sentences; for example one can SPECify alternative or optional words to be recognized. Finally, the matcher is the module that takes both trees and decides whether or not there is a match.

The following block diagram describes the parser:

```
*********           ************
* User  ************* Sentence *
* input *      * *  * Parser   *
*********      * *  ************
               * *     *
************   * *     *    ************
* External ***** *     ****** Sentence *
* Grammar  *     *          * Tree     *** ***********
************     *          ************ * *         *
                 *                       *** Matcher *
**************   *          ************ * *         *
* External   *****          * Spec     *** ***********
* Dictionary ****       ****** Tree     *
**************  *       *    ************
                *       *
                *       *
*********       *   ************
* Coder ************* Spec     *
* Specs *           * Parser   *
*********           ************
```

In the case of a common sentence, only a handful of comparisons may be required before a match is found. If the user were to type in something totally inappropriate, the resulting tree might be compared to fifty or even a hundred specs before falling through to a default answer like "what are you trying to say?".

Parseable Sentences:

SENTENCE	POSSIBLE MATCHING SPECS
"look"	'look[/!*]' or 'look'
"get the food"	'get/food' or 'get'
"hit the small tree"	'hit/tree<small'
	'hit/tree[<small]'
	'hit[/tree[<small]]
	'hit'
"hit the small green tree with the ax"	'hit/tree/ax'
	'hit/tree<(green<small)/ax<with'
"burn it" = "burn tree" after last sentence	'burn/tree'
"when do fairies sleep?"	'(sleep<do<fairies)<when'
"what time is it?"	'is<what<time'

User Parse Trees

When a user types in a sentence, the sequence of words is used to create a parse tree that represents the syntactic structure of the sentence. This is done according to traditional grammar rules.

The parser uses two external resources: the dictionary and the grammar. The dictionary defines which words will be recognized and the grammar defines the rules used to analyze the sentence. Since both of these resources are external, they can be changed and recompiled. The dictionary is recompiled by typing VC (vocabulary compiler) and the grammar requires GC (grammar compiler). For information on how to modify these files, see your local friendly and infinitely patient SCI system programmer. :-)

All sentences typed during a game are imperative, that is, they constitute commands. An imperative sentence starts with a verb and is followed by optional direct and indirect objects.

Example: "take the gold from the dwarf"

```
    root=vp
    |
    +-------------+------------------+
    |             |                  |
    root=verb     dobj=np            iobj=ap
    |             |                  |
    "take"        |                  |
                  +-------+          +---------+
                  |       |          |         |
                  ignore  root       mod       root
                  art     noun       prep      np
                  "the"   "gold"     "from"    |
                                               +-------+
                                               ignore  root
                                               art     noun
                                               "the"   "dwarf"
```

The above tree can also be written using parenthesized notation as follows:

```
(root vp
   (root verb . 'take')
   (dobj np
      (ignore art . 'the')
      (root noun . 'gold')
   )
   (iobj ap
      (mod prep . 'from')
      (root np
         (ignore art . 'the')
         (root noun . 'dwarf')
      )
    )
)
```

take some time to look at this example and understand the new notation. For the sake of convenience it will be used throughout this document.

The grammar rules used to produce the above tree are:

```
s    = vp
vp   = (root verb) (dobj np) (iobj ap)
np   = (ignore art) (root noun)
ap   = (mod prep) (root np)
verb = from dictionary
art  = from dictionary
noun = from dictionary
prep = from dictionary
```

Said-specs

The previous section briefly describes the upper branch of the parsing system. The lower branch allows the SCI coder to specify what sentences he wants to recognize using "Said" statements. The above sentence ("take the gold from the dwarf") can be recognized by:

```
(Said 'take/gold/dwarf<from')
```

which would return TRUE if the user typed the phrase in. Let's dissect this spec:

take:

the first of three sections separated by slashes, it is the verb in the verb phrase.

/gold:

the slash signals the beginning of the direct object, which is a noun phrase with root "gold"

/dwarf:

the slash signals the beginning of the indirect object, which is an Associated Phrase. The root is "dwarf"

<from:

the "<" means "modified by". In this case the preposition "from" modifies the root of the AP

Said syntax

Said-specs are written in a sub-language with its own syntax and operators to produce expressions not unlike those we use for arithmetic.

The main concept implemented by the "Said-er" is:

VERB/DIRECT OBJECT/INDIRECT OBJECT

Note that the slash is a part of the description of a dobj or iobj, not a detached separator.

Each part can be made OPTIONAL by enclosing it in brackets. This is also true for any subpart. Thus 'look [/rock]' means a phrase with verb 'look' and an optional direct object that should be 'rock' if present. Note that the slash is INSIDE the brackets.

MODIFIERS:

Signalled using "<", a given root can have multiple modifiers as in "sit down on bed" which can be specified by 'sit<down<on/bed' since the two prepositions are modifiers to the verb.

In general, adverbs and prepositions modify verbs, adjectives modify nouns or other adjectives and nouns modify other nouns.

ALTERNATIVES:

Let's say we want to respond the same way to either
"start the car" and
"turn on the car"
this can be done with OR alternatives:
'start,(turn<on)/car'

The comma means "OR". Notice the parenthesis surrounding turn<on; they are required because "," has higher PRECEDENCE than "<". This is just like common arithmetic expressions, as in
2*3+4 versus 2*(3+4) where the parenthesis make the difference between getting 10 or 14 because "*" has higher precedence than "+"

Said Spec Trees:

When a spec is processed, it is turned into a tree that can be compared with the user parse tree. Since the spec IS a structural specification, the mapping is straightforward. It is important to keep these mappings in mind though, because our goal is to imagine the structure of possible user input trees and create specs that match them closely enough for the similarity to be recognized by the matcher.

At the top level there are three possible "slots" to fill: verb, dobj and iobj. At lower levels the only slots to be filled are ROOT and MOD(ifier). Alternatives and optionals generate OR and OPT nodes with the appropriate children nodes under them.

Example: 'start,(turn<on)[/car]' generates the following tree:

```
(root s
   (root verb
      (or
         (root verb . 'start)
         (root
               (root verb . 'turn')
               (mod  prep . 'on)
          )
       )
    )
   (opt
      (dobj np
          (root noun . 'car')
       )
    )
 )
```

Tree Matching:

The tree matching algorithm is quite simple: for each part of the spec tree, find a corresponding branch in the user parse tree. For example if one is looking for a modifier match, one may search both among the modifiers at the current tree depth and also among the modifiers of the roots, the modifiers of the root's roots and so on...

Special handling is required for OR and OPT nodes. An OR-node match will only be declared a failure if all the options fail, and it will succeed as soon as one of the options does, not bother to check the rest. An OPT-node match will succeed if a true match is found but also if no matching slot exists in the user tree. However if there is a matching slot with a different word in it the OPT-node match will fail.

Example: "start the radio" will fail to unify with 'start,(turn<on)[/car]' as follows:

User tree:

```
(root vp
   (root verb . 'start')
   (dobj np
      (ignore art . 'the')
      (root noun . 'radio')
    )
 )
```

Procedure:

```
match spec root:
   match spec OR-node
      try first OR-option (root verb .'start') -> OK

match spec optional dobj:
   look for a dobj in user tree -> found
   if found, compare            -> FAILED
```

Thus the tree comparison fails in this example.

EXAMPLES:

```
"get rock"
                              'get/rock'
                              'get[/rock]'
                              '[get]/rock'
                              'get' or '/rock'

"hit tree with ax"
                              'hit/tree/ax<with'
                              'hit/tree/ax[<with]'

"go get prison guard jacket"
                              'get<go/jacket<(guard<prison)'

"get food" followed by
"eat it" is the same as
"get food" and "eat food"

"what time is it?"
                              'is<what<time'

"which witch made stew?"
                              'made<which<witch/stew'

"do fairies sleep?"
                              'sleep<fairies<do'

"when do fairies sleep?"
                              '(sleep<fairies<do)<when'

"get across creek with boat"
                              'get<across/creek/boat'

"hamburger"
                              '/hamburger'

"coffee with sugar"
                              '/coffee/sugar<with'
```