Programming Languages and Translators, Fall 2010

Project Proposal

Spoke: a Language for Spoken Dialog Management

William Yang Wang, Xin Chen, Chia-che Tsai, Zhou Yu
(yw2347, xc2180, ct2459, zy2147)@columbia.edu

1. Introduction

Spoken dialog management remains one of the most challenging topics in natural language processing (NLP) and artificial intelligence (AI). There are various different spoken dialog management theories (Cole et al., 1997; Pieraccine and Huerta, 2005) and models (e.g. Figure 1.1), but there's no unified programming language and framework to describe and represent all of them. Traditional programming languages, including Java, C, Perl and Lisp, are not designed to deal with natural language applications, and thus are slow, redundant, and ineffective when dealing with spoken dialog systems.

In this project, we decide to touch the novel area where the "unnatural language processing" problems (Aho, 2009) meet the natural language processing problems. We design the *Spoke* programming language, which is first of its kind for implementing different spoken dialog management strategies. In other words, the user of this language can use *Spoke* set up their own spoken dialog management schema.

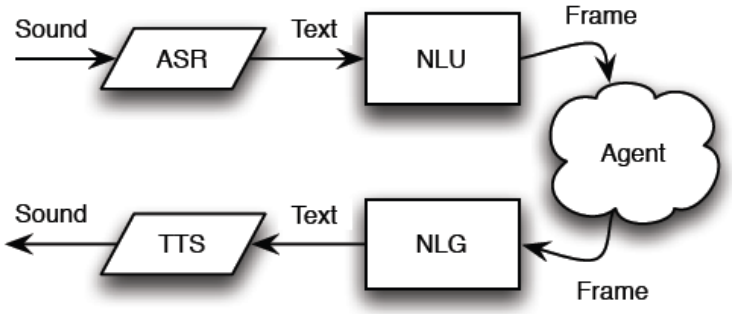

Figure 1.1 An agent based spoken dialog management architecture (Leuski and Traum, 2008)

Figure 1.2 shows the design architecture for the *Spoke* Language. We denote that the person who writes *Spoke* is the developer, and the user of the application is the user. The developer designs the dialog system and writes the *Spoke* source code, and we, as the language designers, implement the parser and interpreter for *Spoke*, using O'Caml language. After the developer has done his job, the back end system will wrap the entire application into the spoken dialog system, which also consists of an Automatic Speech Recognition (ASR) module and an Text-to-speech(TTS) module.

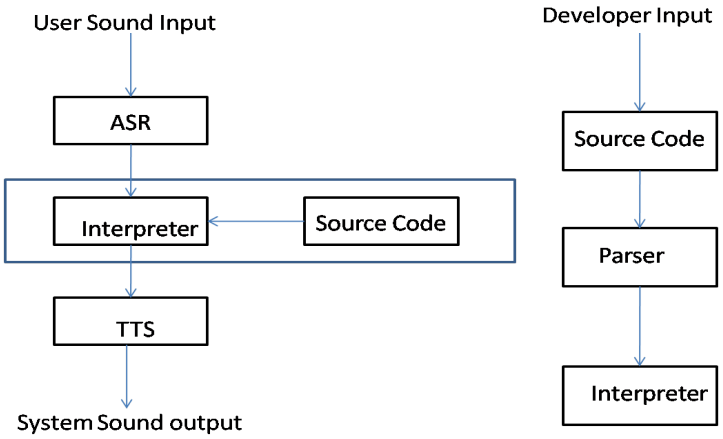

Figure 1.2. Spoke Programing Language Design Architecture

The main advantages of our language:

- Succinct syntax structure
- Special language designs for NLP
- Basic API support
- Joint programming language and natural language parsing

Real Life Significance:

- Automatic Customer Service (Gorin et al., 1997)
- Intelligent Tutoring System (Litman and Silliman, 2004)
- Emergency Call Centers (Vidrascu and Devillers, 2005)
- Virtual Museum Guides (Swartout et al., 2010)

2. Basic Features

2.1 Basic Syntax

The *Spoke* language will be written in sequences, with each line representing one command in the source code. There is no need for using semicolon to terminate the command and the single expression argument containing multiple lines is currently not supported. The source code can be written in a single file, or be typed in by the user through the command line interactive interface.

2.2 Data Types, Operators, Expression

The four basic data types that the language support will be Boolean, Integer, Float and String. A variable will be defined when it is first assigned. For each data type there will be several operators provided for processing. Also the conversion among data types is supported.

2.2.1 Boolean

A variable of Boolean type can only be assigned by two values: TRUE and FALSE. Variable of other kind of data type can be converted to Boolean type by Boolean Expression. Operator supported for Boolean types include **gt** (greater than), **lt** (less than), **eq** (equal), **ne** (not equal), **ge** (not less than), **le** (not greater than) and logic gates like **and, or** and **not.**

2.2.2 Integer and Float

A variable of Integer and Float can be assigned by a numeric value. For simplicity, direct assignment among different numeric types is not supported. Conversion between Integer and Float can be done by casting function Int() and Float(). Note that Int() will round the floating-point value. Operators supported for Integer and Float type include + (addition), - (subtraction), * (multiplication), / (division), % (modulus division).

2.2.3 String

There will be no data type representing characters in this language. A variable of String type will be a sequence of character, and each character can be retrieved by specifying the position in the string. A constant of String type can be given by characters wrapped by quotation marks (“). Two quotation marks (“”) represent empty strings. Variable of other type can be converted into String type by the casting function Str(). One operator supported for String type will be + (concatenate). By the way, equality operator (=) is for the comparison of strings, word by word.

2.2.4 Array

Arrays can be defined by assigning squared brackets (e.g., x = []). Also using squared brackets with a numeric value can specify the element in each array (e.g., x[0]) Addition operator (+) on arrays represents appending elements to the array.

2.3 Control Flows

2.3.1 Conditional Structure

The only condition structure supported in this language will be if-else structure. The structure starts with a keyword “if” and finishes with another keyword “fi”. After the “if” keyword, a Boolean variable or a Boolean argument must be given to specify the condition to execute the code segment. Figure 2.3.1.1 shows an example for the if-then-else structure.

```
if x ne 0
      y = y / x
else
      y = 0
fi
```

Figure 2.3.1.1 Sample code for if-else structure

2.3.2 Loop Structure

Three loop structures will be supported in this language, while-loop, for-loop and do-while-loop. while-loop and do-while-loop will be given a condition by which the iteration of the code

segment is determined. For for-loop, we will not provide a C style for-loop like for (i=0; i<10; i++). Instead, we adopt the common style provided in script languages like Bash or Python, in which for-loop iterates on an array. Figure 2.3.2.1 shows a sample for while-loop and do-while-loop, and Figure 2.3.2.2 features another sample for for-loop.

```
while x ne 0
     y = y + x
     x = x - 1
done
```

```
do
     y = y + x
     x = x - 1
while x ne 0
```

Figure 2.3.2.1 Sample code for while-loop and do-while-loop structure

```
for x in [0, 1, 2]
     y = y + x

done
```

Figure 2.3.2.2
Sample code for for-loop structure

2.4 Input and Output

Our language provides input and output from standard devices and file systems. A “Print”command will print all the variables that follow the keyword (multiple variable has to be separated by comma). A “Rad” command and “readline” command will read a token or a whole line from the standard input and stored to the variable that follows. File I/O will be features in the style of python language. A Open() function will declare a file object and “print” or “read” command can be performed on it. Figure 2.4.1 shows an example of File I/O.

```
f1 = Open “test.txt” “w”
f2 = Open “input.txt” “r”
Readline from f2, buffer
Print to f1, buffer
```

Figure 2.4.1
Sample code for File I/O

2.5 Functions and APIs

Functions in this language will be defined with a keyword “Func”. Parameters that follow the keyword consist of the name of the function, and a sequence of arguments. The argument will not be type-specific and can only be used as variable locally in the functions. Any other variable used outside the function will be global variable. At the end of the function, a “return” keyword will return all the parameters that follow. These return values are also not type-specific. Figure 2.5.1 shows an example of function.

```
Func concat str1 str2
     str = str1 + str2
     return str
End
str = concat “Hello” “world”
```

Figure 2.5.1
sample code for function

For the convenience of developers, we will implement a basic set of APIs in our library. These APIs include string processing, mathematical calculation, array manipulation and system commands. Table 2.4.2 lists all the APIs that we plan to implement.

1.	Len	Size of array or strings	11.	Abs	Aboslute value
2.	Left	Sub-string at the left of strings	12.	Floor	Round up
3.	Right	Sub-string at the right of strings	13.	Ceil	Round down
4.	Mid	Sub-string at the mid of strings	14.	Pow	Raise to power
5.	Find	Position of keyword in string	15.	Sqrt	Square root
6.	Sep	Seperate strings into tokens	16.	Log	Logarithm
7.	Rep	Replace keyword in strings	17.	Exp	Exponential
8.	Index	Index of element in arrays	18.	System	Run command in shell
9.	Sub	Substitution of arrays	19.	Exit	Exit the program
10.	Map	Map elements to another type			

Table 2.5.2 Build-in Functions and APIs

3. Advanced Features

3.1 Spoken Dialog Management

We present a language by which developers can easily design their spoken dialog systems. We introduce a serials of new data types, representing spoken dialogs in English, and provide adequate operators and APIs. Operations will be preformed on a concrete, well-formed syntax structure, which is produced by some NLP parser either provided by the standard libraries or designed by developers. The design of NLP parser must be flexible, and we will not spend time on implementing more than one NLP parser, since this work is not the primary goal of this project. Developers can always implement their own NLP parser using the basic feature (mentioned in the previous chapter) we provide in this language. On the other hand, the NLP parser provided by the library should also be part of APIs instead of a feature of the language.

We adopt the syntax structure of the Penn Treebank designed by M.P. Marcus, B. Santorini, M.A. Marcinkiewicz at UPenn (Marcus et al., 1993). The Penn Treebank is a tag-based natural language repository, primarily but not necessarily designed for English, which can interpret a sentence into grammatical structures. They use a set of predefined tags to categorize each word, phrase or punctuation in a sentence, thus developers can easily perform whatever postprocessing on it. We assume that developers already know the format of tagged dialog and provide operation and expression interfaces.

In addition to tagging the sentence, the Penn Treebank uses a tree structure to describe the semantics of more complicated grammar. They use brackets to mark and separate each branch in the tree. Figure 3.1.1 shows an example of tagging and bracketing an English sentence.

Original sentence:

```
Battle-tested industrial managers here always buck up nervous newcomers.
```

Tagged sentence:

```
Battle-tested/NNP industrial/JJ managers/NNS here/RB always/RB buck/VB
up/IN nervous/JJ newcomers/NNS ./.
```

Bracketed sentence:

```
(S (NP Battle-tested/NNP industrial/JJ managers/NNS here/RB)
   always/RB
   (VP buck/VB up/IN
      (NP nervous/JJ newcomers/NNS) ) .)
```

Figure 3.1.1 Examples for tagging and bracketing sentences

Table 3.1.2 features the tags defined in our language. Tags will be all alphabetic, written in capital. During the development of the language, the definition of tags will be of 3 stages:

I. Adopt predefined tags of only basic representation like NN(Noun), VB(Verb), etc.
II. Adopt predefined tags of all grammatical representation in Penn TreeBank POS tagset.
III. Adopt user-defined tags

1.	CC	Coordinating conjunction	19.	PRPS	Possessive pronoun
2.	CD	Cardinal number	20.	RB	Adverb
3.	DT	Determiner	21.	RBR	Adverb, comparative
4.	EX	Existential *there*	22.	RBS	Adverb, superlative
5.	FW	Foreign word	23.	RP	Particle
6.	IN	Preposition/subord. conjunction	24.	SYM	Symbol
7.	JJ	Adjective	25.	TO	*to*
8.	JJR	Adjective, comparative	26.	UH	Interjection
9.	JJS	Adjective, superlative	27.	VB	Verb, base form
10.	LS	List item marker	28.	VBD	Verb, past tense
11.	MD	Modal	29.	VBG	Verb, gerund or present participle
12.	NN	Noun, singular or mass	30.	VBN	Verb, past participle
13.	NNS	Noun, plural	31.	VBP	Verb, non-3rd person singular
14.	NNP	Proper noun, singular	32.	VBZ	Verb, 3rd personA singular
15.	NNPS	Proper noun, plural	33.	WDT	Wh-determiner
16.	PDT	Predeterminer	34.	WP	Wh-pronoun
17.	POS	Possessive ending	35.	WPS	Possessive wh-pronoun
18.	PRP	Personal pronoun	36.	WRB	Wh-adverb

Table 3.1.2 The Penn TreeBank Part-of-speech (POS) Tag Set

The data type we created for representing a semantic sentence is called Utterance, which means a sentence in the spoken dialog. An utterance must be well-tagged strings, generated by the default POS tagger, parser or customized parser. To distinguish utterances with normal strings, an utterance constant will be wrapped in apostrophe(`) instead of quotation mark (“) used for wrapping strings. The tags in the utterance will be wrapped in a pair of less-than mark and greater-than mark. (To avoid confusion with the less-than and greater-than operators, we use **lt**, **gt** instead as Boolean operators.) Figure 3.1.3 shows two sample codes for utterance manipulation.

```
# The first sample code, without I/O and conversion
Utterance = `<NN>I<VB>am<NN>Jerry`
```

```
if Utterance eq `<NN>I<VB>am<NN>*`
           Name = Utterance[2]
           print `<UH>Hello<NN>World` + Name
fi


# The second sample code, with I/O and conversion
Use Parser default                    # define a default parser

readline Input                        # read a line from stdin
Utterance = Utter(Input)    # conversion

if Utterance eq `<NN>I<VB>am<NN>*`
      Name = Utterance[2]
      Reply = str(`<UH>Hello<NN>World` + Name)
      print Reply + "\n"
fi
```

Figure 3.1.3 Sample codes for utterance manipulation.
The first sample code defines a utterance variable with hard-coded value. If the utterance contains a introduction of the speaker's name, the name is retrieved and a Hello World message is printed.
The second sample code works in a similar way, except the input is read from the keyboard, converted into utterance using default parser.

The structure of bracketed utterance is actually a tree structure, we provide the same way as fetching elements from arrays to fetch branches from the semantic trees. In the first example of Figure 3.3, the name of the speaker is retrieved from the utterance by specifying the third branch (The ordering starts from 0.) Note the branches being retrieved is still of the utterance data type, so the name of speaker here is in the form of another utterance `<NN>Jerry` instead of a string "Jerry". A branch can be appended to the tree structure by using the addition operator (+).

As the Penn Treebank, we also do bracketing on utterance. Parenthesis (**(** and **)**) can be used in the place of words in the utterance. Each sub-utterance wrapped in the parenthesis, tagged as well, represents a phrase in the sentence. From the structural point of view, bracketing utterance is actually creating branches in tree structure with additional descendants. We provides the same way as fetching elements from arrays to fetch descendants from utterances..

Conversion of utterance can be done by simply using casting functions. Utterances can only be casted into strings, by removing all the tags from the sentences. Conversion from strings to utterances is more complicated. Using keyword "Parser", a NLP parser is declared, which is in fact a tagging program. This parser can be used to parse the semantic structure of a string and build up a utterance structure. We will provided a default parser which can be inefficient and only target on sentences in English. However, developer can always design their parser applying their whatever fancy parsing technique.

Matching of utterance is done through Boolean operators. **eq** (equal) and **ne** (not equal) can match utterance variable with another utterance variable or constant. An asterisk (*) in the target means ambiguous matching. This feature helps developers design the logic of spoken dialog system. However we provide a even more powerful way to compare utterances. Using an operation called "Dictionary match", a dictionary (e.g. WordNet) will be given by developers, and they can match utterances by a more flexible

way. For example, the phrases “I am”, “My name is” and “Call me” has the same semantic meaning in English, and all indicate that a name follows. Developers can declare a dictionary file, with these three phrase linked together and use a few line of code to describe the whole logic.

Utterance is one of a most powerful features of this language. It combines natural language and programming language, and can be used in many domain of computer science. For example, the semantic structure of utterances can give advantage for development of machine translation software. Moreover, this feature can be used in domain which has the requirement of pattern matching or semantic parsing. For example, to represent genetic order in biologic information.

4. Team Formation and Schedule

4.1 Team Members:

- **Xin Chen (Chief Language Developer)**
  xc2180@columbia.edu
  1. Responsible for sample programs, basic types, structures, syntax, parsing and mathematical functional design and implementation.
  2. Testing of above basic components in language.

- **Chia-che Tsai (System Administrator and Chief Technical Officer)**
  ct2459@columbia.edu
  1. Set up, monitor and maintain the SVN
  2. Bug and defect tracking
  3. Improve over basic language designs with advanced natural language processing components and features.

- **William Yang Wang (Project Manager and Chief Architect)**
  yw2347@columbia.edu
  1. Plan the blueprint of this project
  2. Set up weekly meetings and interact with team member on project progress
  3. Responsible for project presentation, demo and delivery
  4. Responsible for overall quality control and final integration

- **Yu Zhou (Chief Machine Learning Specialist)**
  zy2147@columbia.edu
  1. Collaborate with Xin Chen on basic language development
  2. Design and deploy advanced machine learning components in the language
  3. Testing of advanced and special components in language

Expertise	Team Members			
	Wang	Chen	Tsai	Zhou
Computational Linguistics and Human Language Technologies	**X**			
Network Systems		**X**		
Operating Systems and Reliable Computing			**X**	
Machine Learning				**X**

4.2 Timeline and Goals

# Phase	Date	Goal	Person-in-charge
1	09/28/2010	Proposal	PM
2	10/10/2010	Language Reference Manual	CLD
3	11/30/2010	Language Implementation	CTO
4	12/04/2010	Testing and Debugging	CTO
5	12/06/2010	Demo and Presentation	PM
6	12/15/2010	Final Report	PM

References

Aho A., Invited Keynote Talk: Unnatural Language Processing, SSST-3, NAACL HLT, 2009

R. Cole, J. Mariani, H. Uszkoreit, A. Zaenen, & V. Zue (eds.), Discourse and Dialogue. Grosz, Barbara; Scott, Donia; Kamp, Hans; Cohen, Phil; Giachin, Egidio. Chapter 6 of Survey of the State of the Art in Human Language Technology, Cambridge University Press, 1997.

Gorin A., G. Riccardi, and J. Wright, "How may I help you?," Speech Communication, vol. 23, pp. 113–127, 1997.

Leuski A. and David Traum A Statistical Approach for Text Processing in Virtual Humans in proceedings of the 26th Army Science Conference December 2008.

Litman D., and S. Silliman. ITSPOKE: An intelligent tutoring spoken dialogue system. In Companion Proc. of the Human Language Technology Conf. of the North American Chap. of the Assoc. for Computational Linguistics (HLT/NAACL), 2004.

Mitchell P. Marcus , Mary Ann Marcinkiewicz , Beatrice Santorini, Building a large annotated corpus of English: the penn treebank, Computational Linguistics, v.19 n.2, June 1993

Pieraccini R. and Huerta, Juan, "Where do we go from here? research and commercial spoken dialog systems", In SIGdial-2005, 1-10.

Swartout W., David Traum, Ron Artstein, Dan Noren, Paul Debevec, Kerry Bronnenkant, Josh Williams, Anton Leuski, Shrikanth Narayanan, Diane Piepol, Chad Lane, Jacquelyn Morie, Priti Aggarwal, Matt Liewer, Jen-Yuan Chiang, Jillian Gerten, Selina Chu, and Kyle White. Ada and Grace: Toward realistic and engaging virtual museum guides. In Proceedings of IVA 2010, Philadelphia, September 2010

Vidrascu, L., Devillers, L., Detection of real-life emotions in call centers. In: Proc. Eurospeech, pp. 1841–1844. 2005