CS4118 PROGRAMMING LANGUAGES AND TRANSLATORS

Spoke

Language Reference Manual*

William Yang Wang, Chia-che Tsai, Zhou Yu, Xin Chen

2010/11/03

(yw2347, ct2459, zy2147, xc2180)@columbia.edu
Columbia University, New York, NY

*This Language Reference Manual is written based on D. M. Ritchie 's C Reference Manual.

Table of Content

1. Introduction 3
2. Lexical conventions 3
   - 2.1 Comments 3
   - 2.2 Identifiers (Names) 3
   - 2.3 Keywords 4
   - 2.4 Constants 4
     - 2.4.1 Integer constants 4
     - 2.4.2 Floating constants 4
   - 2.5 Strings 4
     - 2.5.1 Native Strings 4
     - 2.5.2 Escape Strings 4
     - 2.5.3 Tagged Strings 4
3. Fundamental Types 5
4. Conversions 5
5. Expressions 5
   - 5.1 Primary expressions 5
     - 5.1.1 LPAREN/RPAREN expression (expression) 5
     - 5.1.2 LBRACK/RBRACK expression [expression] 5
     - 5.1.3 COMMA expression ,expression 5
     - 5.1.4 COLON expression :expression 6
   - 5.2 Additive operators 6
     - 5.2.1 PLUS expression +expression 6
     - 5.2.2 MINUS expression –expression 6
   - 5.3 Multiplicative operators 6
     - 5.3.1 TIMES expression *expression 6
     - 5.3.2 DIVIDE expression /expression 7
     - 5.3.3 MODULUS expression %expression 7
   - 5.4 Equality operator 7
     - 5.4.1 EQ expression ==expression 7
     - 5.4.2 NEQ expression !=expression 7
   - 5.5 Relational operators 7
     - 5.5.1 LT expression <expression 7
     - 5.5.2 GT expression >expression 7
     - 5.5.3 LEQ expression <=expression 8
     - 5.5.4 GEQ expression >=expression 8
   - 5.6 AND expression &&expression 8
   - 5.7 OR expression ||expression 8
   - 5.8 Unary operators 8
     - 5.8.1 – expression 8
     - 5.8.2 NOT expression !expression 8
   - 5.9 ASSIGN expression =expression 8
6. Declarations 8
7. Statements 9
   - 7.1 Expression statement 9
   - 7.2 Compound statement 9
   - 7.3 Conditional statement 9

7.4 While statement........9
7.5 For statement........9
7.6 Return statement........9
8. Scope rules........10
8.1 Lexical scope........10
8.2 Scope of externals........10
9 Sample ***Spoke*** Program........10

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, 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.

Spoke programming language is a domain specific language designing for implementing different spoken dialog management strategies. The Spoke user can set up their own spoken dialog management schema with very succinct syntax structure. In addition, Spoke provides basic API support and joint programming language and natural language parsing.

2. Lexical conventions

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.

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The character "#" introduces a comment, which comments out the entire line.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore ‘‘_’’ counts as alphabetic. Upper and lower case letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

2.3 Keywords

IF ELSE
ELIF FI
FOR IN
NEXT WHILE
LOOP BREAK
CONTINUE FUNCTION
NATIVE RETURN
END IMPORT
EXTERN GLOBAL

2.4 Constants

There are several kinds of constants, as follows:

2.4.1 Integer constants

An integer constant is a sequence of digits. All integers are decimal only.

2.4.2 Floating constants

A floating constant consists of an integer part, a decimal point, and a fraction part. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing.

2.5 Strings

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.

2.5.1 Native Strings

Native strings will be wrapped by the single quotation mark “ ‘ ” symbol from the beginning to the end.

2.5.2 Escape Strings

Escape strings will be wrapped by the double quotation mark “ “ ” symbol from the beginning to the end.

2.5.3 Tagged Strings

Escape strings will be wrapped by the “ ` ” symbol from the beginning to the end.

3. Fundamental Types

Spoke supports three fundamental types of objects: strings, integers, and single-precision floating-point numbers.

Strings consist of characters (declared, and hereinafter called, char) chosen from the ASCII set; they occupy the right-most seven bits of an 8-bit byte. It is also possible to interpret chars as signed, 2's complement 8-bit numbers.

Integers (int) are represented in 16-bit 2's complement notation.

Single precision floating point (float) quantities have magnitude in the range approximately 10±38 or 0; their precision is 24 bits or about seven decimal digits.

4. Conversions

The ***spoke*** language itself does not support conversions at this moment. But it is possible to call API functions in Java to do conversions.

5. Expressions

5.1 Primary expressions

5.1.1 LPAREN/RPAREN expression (expression)

Description: A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue. It does affect the order of operation.
Rules: If we have "(expression1) expression2", expression1 will get computed first.
Example: (1 + 2) * 3 => 3*3 = 9

5.1.2 LBRACK/RBRACK expression [expression]

Description: A primary expression followed by an expression in square brackets is a primary expression. It is used to define an array.
Rules: we can have an array of size two with two constants as elements [constant, constant]; we can also append an array into another array [constant]+[constant : constant]. Also we can have expression as element in an array [expression, expression].
Example: array = [ ] + [1 , 2] => [1, 2]
array = [ ] + [1+2, 3] => [3, 3]
array[0] = 1 => [1]
array = array[0 : 2] => [0, 1, 2]

5.1.3 COMMA expression ,expression

Description: A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right.

Rules: It should be avoided in situations where comma is given a special meaning, for example in actual arguments to function calls and lists of initializers.
Example: lists of initializers *: [1 , 2]*
Function calls: function foo
return 1, 2, 3

5.1.4 COLON expression :expression
Description: a : expression is used in array
Example: array[0 : 2] => [0, 1, 2]

5.2 Additive operators
The additive operators +and − group left-to-right.

5.2.1 PLUS expression +expression
Description: The result is the sum of the expressions. If both operands are int, the result is int. If both are float, the result is float. If both are string, the result is string. We can also append an array to another array. No other type combinations are allowed.

Rules: int + int => int; float + float => float; string + string => string;
[ ] + [ ] => [ ]
Example: 3+4 => 7
[1, 2] + [3, 4] => [1, 2, 3, 4]
1.2 + 2.1 => 3.3
1.2 + 1 => error

5.2.2 MINUS expression –expression
Description: The result is the difference of the operands. If both operands are int, the result is int. If both are float, the result is float. We can't use "-" expression in string or array manipulation.
Rules: int - int => int; float - float => float;
Example: 7-3 => 4
1.2-3.2 => -2.0

5.3 Multiplicative operators
The multiplicative operators *, /, and % group left-to-right.

5.3.1 TIMES expression *expression
*Description: The binary * operator indicates multiplication.* The result is the sum of the expressions. If both operands are int, the result is int. If both are float, the result is float. *No other combinations are allowed.*
Rules: int * int => int; float * float => float;

Example: 7*3 => 21
1.2*2.0=> 2.4

5.3.2 DIVIDE expression /expression
Description: The binary /operator indicates division. The same type considerations as for multiplication apply.
Example: 7/3 => 2
    1.2/2.0=> 0.6

5.3.3 MODULUS expression %expression
Description: The binary %operator yields the remainder from the division of the first expression by the second. Both operands must be int, and the result is int. In the current implementation, the remainder has the same sign as the dividend.
Rules: int % int => int
Example: 7%3 => 1

5.4 Equality operator
The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus ‘‘a<b == c<d’’ is 1 whenever a<b and c<d have the same truth-value).

5.4.1 EQ expression ==expression
Description: equal to comparison
Rules: int == int; float == float
Example: a==b

5.4.2 NEQ expression !=expression
Description: not equal to comparison
Rules: int == int; float == float
Example: a!=b

5.5 Relational operators
The relational operators group left-to-right, but this fact is not very useful; ‘‘a<b<c’’ does not mean what it seems to. The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield false if the specified relation is false and true if it is true. Operand conversion is exactly the same as for the + operator.

5.5.1 LT expression <expression
Description: less than comparison
Rules: int < int ; float < float
Example: 1<2 => true; 1.0<2.0 => true

5.5.2 GT expression >expression
Description: greater than comparison
Rules: int > int ; float > float
Example: 2>1 => true; 2.0<1.0 => true

5.5.3 LEQ expression <=expression
Description: less than or equal to

5.5.4 GEQ expression >=expression
Description: greater than or equal to

5.6 AND expression &&expression
Description: The && operator returns true if both its operands are non-zero, false otherwise. && guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is false. The operands need have the same type, and each must have one of the fundamental types.
Example: (a&&b)

5.7 OR expression ||expression
Description: The || operator returns true if either of its operands is non-zero, and false otherwise. || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero. The operands need to have the same type, and each must have one of the fundamental types.
Example: (a||b)

5.8 Unary operators
Expressions with unary operators group right-to-left.

5.8.1 – expression
The result is the negative of the expression, and has the same type. The type of the expression must be int or float.

5.8.2 NOT expression !expression
Description: The result of the logical negation operator ! is true if the value of the expression is false, false if the value of the expression is non-zero. The type of the result is Boolean. This operator is applicable only to int.
Example: (!a)

5.9 ASSIGN expression =expression
Description: The assignment operator groups right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.
Rules: lvalue = expression
Example: a=3; a=2.0+3.0;

6. Declarations

There will be no declaration for variables and fundamental data types.

7. Statements

Except as indicated, statements are executed in sequence.

7.1 Expression statement

Most statements are expression statements, which have the form expression; Usually expression statements are assignments or function calls.

7.2 Compound statement

So that several statements can be used where one is expected, the compound statement is provided:

```
compound-statement:
( statement-list )
statement-list:
    statement
    statement statement-list
```

7.3 Conditional statement

The three forms of the conditional statement are

```
1) IF ( expression ) THEN statement
2) IF ( expression ) THEN statemtent ELSE statement
3) IF ( expression ) THEN statement
   ELIF (expression) THEN statement
   ELSE statement
   FI
```

In all three cases the expression is evaluated and if it is non-zero, the first sub statement is executed.

7.4 While statement

The while statement has the form

```
WHILE ( expression ) statement LOOP
```

The sub statement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

7.5 For statement

The for statement has the form:

```
FOR element in [number of iterations]
statement
NEXT
```

7.6 Return statement

A function returns to its caller by means of the return statement, which has one of the forms

```
return ( expression )
```

In this case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

8. Scope rules

A complete ***Spoke*** program might not be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing ‘‘undefined identifier’’ diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

8.1 Lexical scope

Spoke is neither a block-structured language or a parenthesis-based language. The lexical scope of names declared in external definitions extends from their definition through the end of the file in which they appear. It is an error to redeclare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

8.2 Scope of externals

If a function declares an identifier to be extern, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and extent specified in the definition are compatible with those specified by each function which references the data.

9 Sample ***Spoke*** Program

```
 # 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 9.1 Sample codes for utterance manipulation.
The first sample code defines an utterance variable with hard-coded value. If the utterance contains an 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 key- board, converted into utterance using default parser.

References

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, 1995.
Pieraccini R. and Huerta, Juan, "Where do we go from here? research and commercial spoken dialog systems", In SIGdial-2005, 1-10.