Verbal Language

Kosate Limpongsa 5731012721

Introduction

Verbal language is designed for visually impaired. main benefit of this language is easily to readable and also supported for blind people to write a programming language.

Features included in my language

- Function declaration with multiple variable
- Dynamic type variable
- Some arithmetic calculation (plus, minus, multiple, divide)
- Function call & Return value

Grammar

```
S -> CODE_BLOCK

CODE_BLOCK -> FUNC CODE_BLOCK | LAMBDA

FUNC -> function FUNC_NAME PARAMETER FUNC_BLOCK
PARAMETER -> has input VARIABLE PARAMETER' | LAMBDA
PARAMETER' -> and VARIABLE | LAMBDA

FUNC_BLOCK -> begin FUNC_STATEMENT end
FUNC_STATEMENT -> DO_BLOCK
DO_BLOCK -> do STATEMENT DO_BLOCK | LAMBDA

STATEMENT -> DECLARE | EXPRESSION | ASSIGNMENT | return
EXPRESSION | show EXPRESSION
DECLARE -> declare VARIABLE DECLARE_DEFAULT
DECLARE_DEFAULT -> equal VALUE | LAMBDA
ASSIGNMENT -> assign VARIABLE equal EXPRESSION
EXPRESSION -> OPERAND EXPRESSION' | RUN_FUNC
EXPRESSION' -> OPERATOR OPERAND | LAMBDA

RUN_FUNC -> run FUNC_NAME PARAMS
PARAMS -> VALUE PARAMS | LAMBDA

OPERAND -> VALUE
OPERATOR -> minus | plus | multiple by | divided by

VALUE -> VARIABLE | NUMBER
```

Language Example

```
function square has input side
begin
  do return side multiple by side
end

function area has input width and height
begin
  do return width multiple by height
end

function main
begin
  do declare x equal 20
  do declare y equal 10
  do declare z

  do assign z equal x plus y
  do show z

  do show x minus y
  do show x divided by y

  do show run square x
  do show run area x y
end
```

There are 2 functions declaration names square, area, and main.

Square function get a side argument and return the result of power of two.
Area function receive two variables, width and size, the function will
multiply these two value, then return the result.

In main function, there are some variable initialize and some function call.
For example “do show run square x” run the square function by passing x variable to function.

1. First Set

Non-Terminal Symbol	First Set
CODE_BLOCK	λ, function
FUNC	function
PARAMETER	has, λ
PARAMETER’	and, λ
FUNC_BLOCK	begin
DO_BLOCK	do, λ
STATEMENT	return, show, declare, assign, VARIABLE, NUMBER, run
DECLARE	declare
DECLARE_DEFAULT	equal, λ
ASSIGNMENT	assign
EXPRESSION’	λ, minus, plus, multiple, divided
RUN_FUNC	run
PARAMS	λ, VARIABLE, NUMBER
OPERATOR	minus, plus, multiple, divided
VALUE	VARIABLE, NUMBER
FUNC_STATEMENT	do, λ
OPERAND	VARIABLE, NUMBER
S	λ, function
EXPRESSION	VARIABLE, NUMBER, run

2. Follow Set

Non-Terminal Symbol	Follow Set
S	$
CODE_BLOCK	$
FUNC	function, $
PARAMETER	begin
PARAMETER’	begin
FUNC_BLOCK	function, $
FUNC_STATEMENT	end

Non-Terminal Symbol	Follow Set
DO_BLOCK	end
STATEMENT	do, end
DECLARE	do, end
DECLARE_DEFAULT	do, end
ASSIGNMENT	do, end
EXPRESSION	do, end
EXPRESSION’	do, end
RUN_FUNC	do, end
PARAMS	do, end
OPERAND	minus, plus, multiple, divided, do, end
OPERATOR	VARIABLE, NUMBER
VALUE	VARIABLE, NUMBER, minus, plus, multiple, divided, do, end

3. Parsing Table

3.1 Grammar Order

#	Expression
1	S → CODE_BLOCK
2	CODE_BLOCK → FUNC CODE_BLOCK
3	CODE_BLOCK → λ
4	FUNC → function FUNC_NAME PARAMETER FUNC_BLOCK
5	PARAMETER → has input VARIABLE PARAMETER'
6	PARAMETER → λ
7	PARAMETER' → and VARIABLE
8	PARAMETER' → λ
9	FUNC_BLOCK → begin FUNC_STATEMENT end
10	FUNC_STATEMENT → DO_BLOCK
11	DO_BLOCK → do STATEMENT DO_BLOCK
12	DO_BLOCK → λ
13	STATEMENT → DECLARE
14	STATEMENT → EXPRESSION

#	Expression
15	STATEMENT → ASSIGNMENT
16	STATEMENT → return EXPRESSION
17	STATEMENT → show EXPRESSION
18	DECLARE → declare VARIABLE DECLARE_DEFAULT
19	DECLARE_DEFAULT → equal VALUE
20	DECLARE_DEFAULT → λ
21	ASSIGNMENT → assign VARIABLE equal EXPRESSION
22	EXPRESSION → OPERAND EXPRESSION'
23	EXPRESSION → RUN_FUNC
24	EXPRESSION' → OPERATOR OPERAND
25	EXPRESSION' → λ
26	RUN_FUNC → run FUNC_NAME PARAMS
27	PARAMS → VALUE PARAMS
28	PARAMS → λ
29	OPERAND → VALUE
30	OPERATOR → minus
31	OPERATOR → plus
32	OPERATOR → multiple by
33	OPERATOR → divided by
34	VALUE → VARIABLE
35	VALUE → NUMBER

3.2 Parsing Table

	function	FUNCTION_NAME	has	input	VARIABLE	and	begin	end	do	return
S	1	0	0	0	0	0	0	0	0	0
CODE_BLOCK	2	0	0	0	0	0	0	0	0	0
FUNC	3	0	0	0	0	0	0	0	0	0
PARAMETER	0	0	5	0	0	0	6	0	0	0
PARAMETER’	0	0	0	0	0	7	8	0	0	0
FUNC_BLOCK	0	0	0	0	0	0	9	0	0	0

	function	FUNCTION_NAME	has	input	VARIABLE	and	begin	end	do	return
FUNC_STATEMENT	0	0	0	0	0	0	0	0	10	0
DO_BLOCK	0	0	0	0	0	0	0	12	11	0
STATEMENT	0	0	0	0	14	0	0	0	0	16
DECLARE	0	0	0	0	0	0	0	0	0	0
DECLARE_DEFAULT	0	0	0	0	0	0	0	20	20	0
ASSIGNMENT	0	0	0	0	0	0	0	0	0	0
EXPRESSION	0	0	0	0	22	0	0	0	0	0
EXPRESSION’	0	0	0	0	0	0	0	25	25	0
RUN_FUNC	0	0	0	0	0	0	0	0	0	0
PARAMS	0	0	0	0	27	0	0	28	28	0
OPERAND	0	0	0	0	29	0	0	0	0	0
OPERATOR	0	0	0	0	0	0	0	0	0	0
VALUE	0	0	0	0	34	0	0	0	0	0

Parsing Table (cont.)

	show	declare	equal	assign	run	minus	plus	multiple
S	0	0	0	0	0	0	0	0
CODE_BLOCK	0	0	0	0	0	0	0	0
FUNC	0	0	0	0	0	0	0	0
PARAMETER	0	0	0	0	0	0	0	0
PARAMETER’	0	0	0	0	0	0	0	0
FUNC_BLOCK	0	0	0	0	0	0	0	0
FUNC_STATEMENT	0	0	0	0	0	0	0	0
DO_BLOCK	0	0	0	0	0	0	0	0
STATEMENT	17	13	0	15	14	0	0	0
DECLARE	0	18	0	0	0	0	0	0
DECLARE_DEFAULT	0	0	19	0	0	0	0	0
ASSIGNMENT	0	0	0	21	0	0	0	0

	show	declare	equal	assign	run	minus	plus	multiple
EXPRESSION	0	0	0	0	23	0	0	0
EXPRESSION’	0	0	0	0	0	24	24	24
RUN_FUNC	0	0	0	0	26	0	0	0
PARAMS	0	0	0	0	0	0	0	0
OPERAND	0	0	0	0	0	0	0	0
OPERATOR	0	0	0	0	0	30	31	32
VALUE	0	0	0	0	0	0	0	0

Parsing Table (cont.)

	by	divided	NUMBER	$
S	0	0	0	0
CODE_BLOCK	0	0	0	3
FUNC	0	0	0	0
PARAMETER	0	0	0	0
PARAMETER’	0	0	0	0
FUNC_BLOCK	0	0	0	0
FUNC_STATEMENT	0	0	0	0
DO_BLOCK	0	0	0	0
STATEMENT	0	0	0	0
DECLARE	0	0	14	0
DECLARE_DEFAULT	0	0	0	0
ASSIGNMENT	0	0	0	0
EXPRESSION	0	0	22	0
EXPRESSION’	0	24	0	0
RUN_FUNC	0	0	0	0
PARAMS	0	0	27	0
OPERAND	0	0	29	0
OPERATOR	0	33	0	0
VALUE	0	0	35	0

Parser Program

I write my code and upload to https://github.com/neungkl/verbal-language-parser file are locate in /program/parser.js

The program I write with Node.JS language. If you need more readable code here are a pseudo code below.

```
RuleList <- Set of rule from table 3.1
ParsingTable <- Table value set from table 3.2

TerminalSymbol <- All set of terminal symbol from grammar
NonTerminalSymbol <- All set of non terminal symbol

stack <- (“$” “S”)
# the string sequence for checking syntax assign here
inputString <- (function square has input …)

while (stack is not empty) {
   top <- stack.pop();

   if(top is in TerminalSymbol) {
        if(top == inputString.front()) {
             inputString.enqueue();
        } else {
             return Error;
        }
   } else if(top is in NonTerminalSymbol) {
        ruleOrder <- ParsingTable[top][inputString.front()];
        stack.push(RuleList[ruleOrder]);
   } else {
        return Error;
   }
}

return True;
```

The Implements detail about type classification (Number is clarify as TerminalSymbol), type checking, information extraction, and other detail about how to parsing value from ParsingTable are provided at the URL above.

But the main concept detail of this algorithm is the same as LL(1) algorithm. Push stack with symbol, checking the type of symbol, compare the type and string, then push the grammar rule to the stack following parsing table number.