TC3048 Compilers Design
Final Documentation
May 8, 2019

Kotoba

Isabela Escalante Campbell
A01193251

Carolina Galván Villarreal
A01192953

Index

Description and Technical Documentation

Project Description

Purpose

To enhance text analysis worldwide and to be the go-to tool for online literacy statistics by visualizing the results that are obtained.

Objective

Our objective is to create a language that simplifies the process and improves the scope of text analysis. It will be able to show visually the results of the analysis the program makes. The uses of this program are focused on basic mathematical statistics, text mining and user-made functions for analysis. It will be an easy to use language so that people working in the literacy area can implement it in their projects.

Scope

Kotoba is a compiler designed mainly for text mining purposes. Our compiler accepts basic arithmetic operations, the use of logical and relational operators, declaration of variables and arrays, four data types, conditions, cycles, declaration and use of functions and recursion. Kotoba also includes some predefined functions including; mean, median, mode, tokenize, sort, amongst others. On the other side, Kotoba does not accept multidimensional variables, files, classes and objects, amongst others.

The web interface can be used to write the code input, display the output, view the global variables used the graphing tool. The input and output are handled using Flask Rest Service.

Requirements and Use Cases

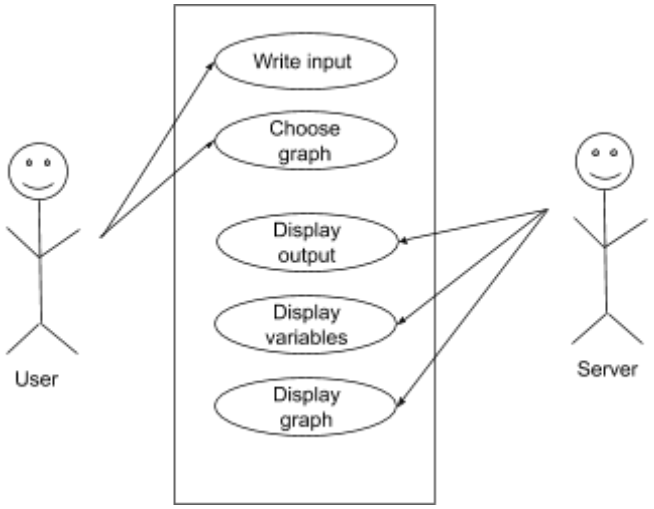
<u>Functional Requirements</u>

- The website must allow the user to upload a text file with the code.
- The website must display the correct output.
- The website must display the variables obtained after execution.
- The website must show the options of graphs.
- The website must display the result graph chosen by the user.
- The compiler must show all compilation errors.
- The virtual machine must show all execution errors.

<u>Non-functional Requirements</u>

- The website must display output and variable results in no more than 3 seconds.
- The website must be able to run in Safari, Chrome and Firefox.

Use Case Diagram



Main Test Cases Description

The following test cases for Kotoba were created. The code and outcome of these can be viewed in the Test Cases section of this document.

- **TC1.** Print statement
- **TC2.** Variable declaration and assignment
- **TC3.** Read statement
- **TC4.** Arithmetic operations
- **TC5**. Expressions and Conditions
- **TC6.** Cycles
- **TC7.** Function declaration and call
- **TC8.** Array manipulation
- **TC9.** Recursion
- **TC10.** Special functions use
- **TC11.** Compilation Error
- **TC12.** Runtime Error

Process Description

General Process for Development

During the nine weeks of development of the project a shared Github was used to upload weekly progresses. Both team members had their own branch to upload their modifications/additions.

The github can be found in the following link:
https://github.com/isabelaescalante/kotoba

Weekly Progress

Week 1 March 4-10 Lexical and Syntax Analyzer	This week the lexical and syntax analyzer were finished. These files accept our tokens and examples. All diagrams included in our proposal have their corresponding grammars in our yacc file. All the specified tokens are included in our lex file as well. Our test-cases were coded to test every grammar. We used auxiliary grammars in order to remove ambiguity. In case there is an error in a token or a grammar the program specifies it.
Week 2 March 11-17 Function Directorio and Variable Table	In this week's progress the function directory, which includes the management of our variable table, was included. We also created our own class for the structures Stack and Queue. We tested our directory by creating functions and variables and ensuring that all data was stored correctly. Until this week a lexical and syntax analysis about a program can be made, and function and variable attributes can be stored inside a directory.
Week 3 March 18-22 Semantic Cube and Quadruple Class	This week we added a structure for our semantic cube with the accepted operations for each of our data types. We also added a Quad class to structure our quadruples. The intermediate code for the start function, declaration of functions and variables, read, write and assignment were also added. Until this week a lexical and syntax analysis about a program can be made, and function and variable attributes can be stored inside a directory. Quadruples for assignment operations, read and write are also stored and displayed. Changes: we added a grammar in order to allow arrays as parameters in functions.

Week 4 March 25-29 Intermediate Code Generation for Decisiones and Cycles	This week we finished the intermediate code generation for all expressions containing logical, relational and arithmetic operations. After execution, our yacc file displays all quadruples created. Until this week a lexical and syntax analysis about a program can be made, and function and variable attributes can be stored inside a directory. Quadruples for all expressions, operations, assignment, read and write are stored. The quadruple generation for conditions was started, however the goto values need to be revised. Changes: the grammar containing the NOT operator was modified. The boolean token regular expression was also modified.
Week 5 April 1-5 Intermediate Code Generation for Functions	During this week we corrected the intermediate code generation for decisiones and the one for cycles was completed. The quads for functions created by the user were also created. We added a Memory class in order to manage our variable addresses and their values. Conflicts in a grammar were also fixed. Until this week a lexical and syntax analysis can be made, functions and variables are stored, and quadruples for expressions, assignment, condiciones, cycles and functions are created. We also started the code for our special functions and the assignment of values to memory addresses. Changes: We added relational operations to words and sentences (equal, notequal).
Week 6 April 8-12 Memory Map, Virtual Machine (arithmetic expressions, statements)	This week we created a memory map to establish the address range for each of our data types in global, local and constant memory. We also implemented arithmetic, logic and relational operations as well as

	conditions to our virtual machine. Our special functions code was also developed during this week. Until this week a lexical and syntax analysis can be made and functions and variables are stored. All quadruples (expressions, assignment, read, write, conditions, cycles, functions) are created with the corresponding addresses. In addition, our virtual machine can execute arithmetic, logic and relational operations and store variable values in our virtual memory.
Week 7 April 22-26 Intermediate Code Generation for Arrays	This week we implemented the intermediate code generation for arrays in order to store correctly in memory all necessary data (with index verification). Functionality regarding functions was also added to our virtual machine. Until this week a lexical and syntax analysis can be made and functions and variables are stored. All quadruples are created with the corresponding addresses. In addition, our virtual machine can execute all operations except arrays and functions, which are incomplete.
Week 8 April 29 - May 3 Intermediate Code Generation and Virtual Machine	During this week we finished the implementation of the array and function code in the virtual machine. We had to modify the functionality and location of the return statement. We also implemented the special functions in the virtual machine. Until this week a lexical and syntax analysis can be made and functions and variables are stored. All quadruples are created with the corresponding addresses. In addition, our virtual machine can execute all operations (statements, functions, special functions, etc) except functions with recursion.

Personal Learnings

Carolina Galván	“During these nine weeks, the biggest lesson for me was learning how to manage the planning, documentation and development of a long and complex project. Since the first week, me and my teammate had to plan in detail what our compiler would include, however, as we progressed in the development, we had to make changes to those plans. This made me realize the importance of making detailed test cases while planning in order to avoid rework. In addition, I learned how to use languages and tools that I had never used before. I believe this project helped me shape and improve my professional skills, and will serve as a management guideline for future projects.”
Isabela Escalante	“This project taught me many things related to time and task management of a team project. Since the beginning of the semester, we started planning how we were going to work on each part of the project. Every week brought a new challenge we had to face, but we also learned each week ways to improve and not get behind on the deliveries. This months of work taught me how to manage my time and work in a way that will increase the probability of having a good project. I learned how to respond to mistakes in our project so that it could be a learning experience and instead of slowing our work, it motivated us more.”

Language Description

Language Name: Kotoba

Main Language Characteristics

Kotoba is a language that focuses on basic mathematical statistics, text mining and user-made functions. It includes assignment statements, an if-else condition, while and do while cycles, read and write functions. Kotoba also supports the basic arithmetic operations (+, -, *, /), logical operators (and, or), and some relational operators (<, >, ==, !=). The four data types in Kotoba are number, word, sentence, and bool. In addition, an array of any of these data types can be created.

To facilitate text analysis several modules are included in our language. These are: length (for words), frequency (for words inside arrays), search (for words inside arrays), mean, median, mode, WordCount (for sentences), tokenize (for sentences), and remove (for sentences).

Errors List

Compilation:

- Function ID already exists
- Variable ID already exists
- Parameter of type *parType* cannot be added
- ID *varId* does not exist
- Input ID *varId* does not exist
- Unable to assign value of type *varType* to ID of type *varType*
- Unable to add/subtract term of type *varType* with term of type *varType*
- Unable to multiply/divide term of type *varType* with term of type *varType*
- Unable to compare expression of type *varType* with expression of type *varType*
- Type *varType* and type *varType* can't be combined with a logical operator
- Operator 'not' can only be applied to operands of type bool
- Cannot evaluate 'if' condition with expression of type *expressionType*
- Do while expression can't be evaluated with value of type *whileType*
- While expression can't be evaluated with value of type *whileType*
- Function ID doesn't exist in directory
- Parameter incorrect for function *pendingFunction*
- Wrong type of variable for index

Runtime:

- Variable has no value to perform arithmetic operation
- Unable to perform division by 0
- Error in arithmetic operation
- No value to assign to
- Variable size is smaller than result
- Variable has no value to perform relational operation
- Error in relational operation
- Variable has no value to perform not operation
- Variable has no value to perform logical operation
- Error in logical operation
- Variable has no value to print
- Wrong input for type
- Variable has no value to be evaluated with
- Index value incorrect for variable of size *varSize*
- Incorrect type of variable for function
- Function *funcName* does not exist in language

Compiler Description

Tools

- OS: macOS Mojave
- Language: Python 3 and PLY for the Lex & Yacc

Lexical Analysis

#	Token	Regular Expression
RESERVED WORDS:		
1	KOTOBA	“kotoba”
2	BEGIN	“begin”
3	END	“end”
4	READ	“kread”
5	WRITE	“kprint”
6	DEC	“declare”
7	BOOL	“bool”
8	NUMBER	“number”
9	WORD	“word”
10	SENTENCE	“sentence”
11	IF	“if”
12	ELSE	“else”
13	DO	“do”
14	WHILE	“while”
15	FUNC	“function”
16	RETURN	“return”
17	VOID	“void”
18	CALL	“call”
19	LENGTH	“length”
20	FREQUENCY	“frequency”

21	SEARCH	“search”
22	EXISTS	“exists”
23	MEAN	“mean”
24	MEDIAN	“median”
25	MODE	“mode”
26	WORDCOUNT	“wordCount”
27	TOKENIZE	“tokenize”
28	REMOVE	“remove”
29	SORTWORDS	“sortWords”
30	SORTNUMBERS	“sortNumbers”
31	SET	“set”
MORE TOKENS:		
32	AND	\&
33	OR	|
34	ID	[a-zA-Z][a-zA-Z0-9]*
35	BOOLCTE	true|false
36	NUMBERCTE	[\+|\-]?[0-9]+(\.[0-9]+)
37	WORDCTE	\”[a-zA-Z0-9]+\”
38	SENTENCECTE	\"(.*?)\"
39	RELOP	(< | > | == | !=)
40	PLUS	\+
41	MINUS	\-
42	MULT	*
43	DIV	\/
44	NOT	\!
45	ENDSTMT	\;
46	COMA	\,
47	DOT	\.

48	OPENCURL	\{
49	CLOSECURL	\}
50	OPENPAREN	\(
51	CLOSEPAREN	\)
52	OPENBRAC	\[
53	CLOSEBRAC	\]
54	EQUAL	\=

Syntax Analysis

START → KOTOBA ID ENDSTMT **DECLARE STARTAUX** BEGIN **BLOCK** END
| KOTOBA ID ENDSTMT **STARTAUX** BEGIN **BLOCK** END

STARTAUX → **FUNCTION STARTAUX**
| empty

BLOCK → OPENCURL **BLOCKAUX** CLOSECURL

BLOCKAUX → **ACTION BLOCKAUX**
| empty

ACTION → **INPUT**
| **OUTPUT**
| **STATEMENT**

INPUT → KREAD OPENPAREN ID CLOSEPAREN ENDSTMT

OUTPUT → KPRINT OPENPAREN **OUTPUTAUX** CLOSEPAREN ENDSTMT

OUTPUTAUX → **EXPRESSION**
| **EXPRESSION** COMA **OUTPUTAUX**

DECLARE → DEC **DECAUX**

DECAUX → **TYPE** ID **DECLAREAUX**
| **TYPE** ID OPENBRAC **CTE** CLOSEBRAC **DECLAREAUX**

DECLAREAUX → ENDSTMT
| COMA **DECAUX**

ASSIGN →SET ID EQUAL **ASSIGNAUX**
| SET ID OPENBRAC **INDEX** CLOSEBRAC EQUAL **ASSIGNAUX**

ASSIGNAUX → **EXP** ENDSTMT
| OPENCURL **ASSIAUX** CLOSECURL ENDSTMT

ASSIAUX → **EXP**
| **CALLFUNCTION**
| **EXP** COMA **ASSIAUX**

CTE → ID
| ID OPENBRAC **INDEX** CLOSEBRAC
| BOOLCTE
| NUMBERCTE
| WORDCTE
| SENTENCECTE

INDEX → NUMBERCTE
| ID

TYPE → BOOL
| NUMBER
| WORD
| SENTENCE

STATEMENT → **ASSIGN**
| **EXPRESSION**
| **CONDITION**
| **CYCLE**
| **CALLFUNCTION**
| **RETURNAUX**

EXPRESSION → **LOGEXPRESSION**
| NOT **LOGEXPRESSION**

LOGEXPRESSION → **RELOPEXPRESSION**
| **RELOPEXPRESSION** AND **LOGEXPRESSION**
| **RELOPEXPRESSION** OR **LOGEXPRESSION**

RELOPEXPRESSION → **EXP**
| **EXP** RELOP **EXP**

EXP → **TERM**
| **TERM** PLUS **EXP**
| **TERM** MINUS **EXP**

TERM → **FACTOR**
| **FACTOR** MULT **TERM**
| **FACTOR** DIV **TERM**

FACTOR → OPENPAREN **EXPRESSION** CLOSEPAREN
| **CTE**

CONDITION → IF OPENPAREN **EXPRESSION** CLOSEPAREN **BLOCK**
| IF OPENPAREN **EXPRESSION** CLOSEPAREN **BLOCK** ELSE **BLOCK**

CYCLE → WHILE OPENPAREN **EXPRESSION** CLOSEPAREN **BLOCK**
| DO **BLOCK** WHILE OPENPAREN **EXPRESSION** CLOSEPAREN ENDSTMT

FUNCTION → FUNC **FUNCAUX** ID OPENPAREN **PARAMETER** CLOSEPAREN OPENCURL **DECLARE BLOCKAUX** CLOSECURL
| FUNC **FUNCAUX** ID OPENPAREN **PARAMETER** CLOSEPAREN OPENCURL **BLOCKAUX** CLOSECURL

FUNCAUX → **TYPE**
| VOID

PARAMETER → **TYPE** ID **PARAMETERAUX**
| **TYPE** ID OPENBRAC **CTE** CLOSEBRAC **PARAMETERAUX**
| empty

PARAMETERAUX → COMA **PARAMETER**
| empty

RETURNAUX → RETURN **EXPRESSION** ENDSTMT

CALLFUNCTION → CALL ID DOT **SPECIAL** OPENPAREN **SPAUX** CLOSEPAREN ENDSTMT
| CALL ID OPENPAREN **SPAUX** CLOSEPAPREN ENDSTMT

SPAUX → CTE
| CTE COMA **SPAUX**
| empty

SPECIAL → LENGTH
| FREQUENCY
| SEARCH
| EXISTS
| MEAN
| MEDIAN
| MODE
| WORDCOUNT
| TOKENIZE
| REMOVE
| SORTWORDS
| SORTNUMBERS

Intermediate Code Generation and Semantic Analysis

Operations Code

Operation Code	Operation
operator_add	Addition
operator_minus	Subtraction
operator_mult	Multiplication
operator_div	Division
operator_assign	Assign
operator_equal	Equal
operator_notequal	Not equal
operator_greater	Greater than
operator_less	Less than
operator_and	And
operator_or	Or
operator_not	Not

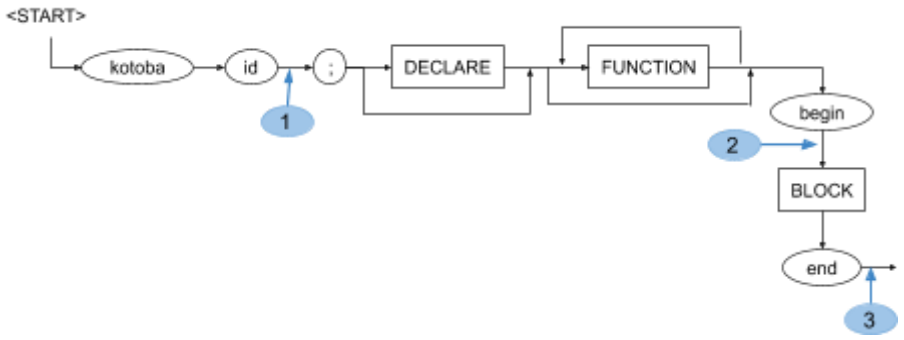
operator_goto	Goto
operator_gotoT	Goto false
operator_gotoF	Goto true
operator_read	Read (input)
operator_print	Wsrite (output)
operator_era	ERA (Activation Record)
operator_param	Parameter
operator_gosub	Go substitute
operator_return	Return
operator_special	Operation for calling special functions
operator_verify	Verification for arrays
operator_address	Adds index to base address

Virtual Memory Addresses

Address Range	Values
1000-1249	Global numbers
4250-1499	Global words
1500-1749	Global sentences
1750-1999	Global bools
2000-2499	Local/Temporal numbers
2500-2999	Local/Temporal words
3000-3499	Local/Temporal sentences
3500-3999	Local/Temporal bools
4000-4249	Constant numbers
4250-4499	Constant words
4500-4749	Constant sentences
4750-4999	Constant bools

Syntax Diagrams

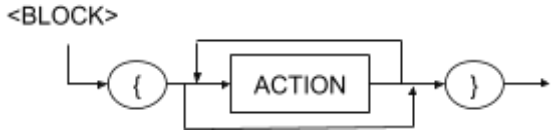
Start



Semantic Actions:

1. Creates initial goto quadruple and adds it to the pending jumps stacks. Adds function Main to Function Directory.
2. Sets the top pending jump result to current quadruple count (indicates where main function begins).
3. Creates end quadruple.

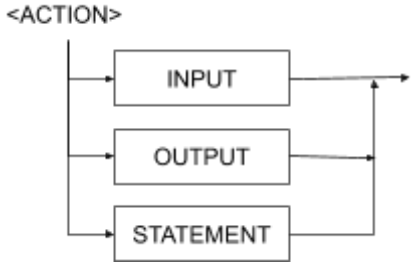
Block



Semantic Actions:

No actions.

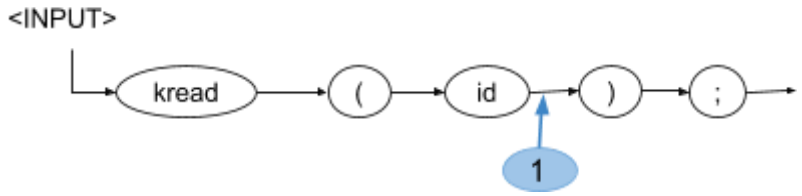
Action



Semantic Actions:

No actions.

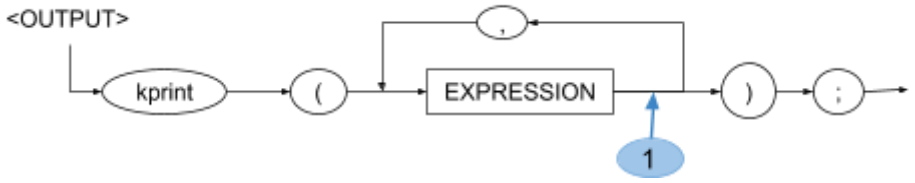
Input



Semantic Actions:

1. Verifies if input id is a valid id, creates a quadruple for the action.

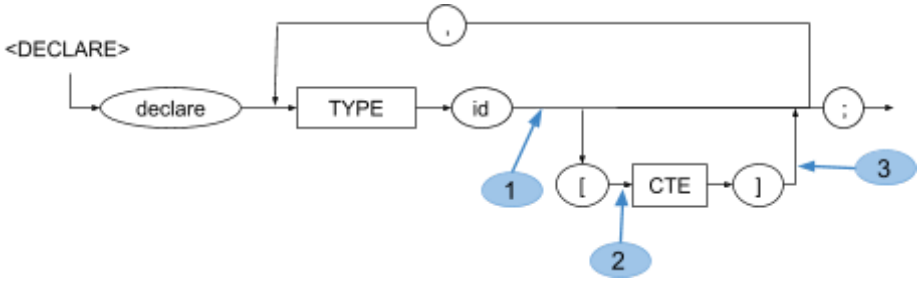
Output



Semantic Actions:

1. Gets operand from pending operands stack and created a quadruple for the action.

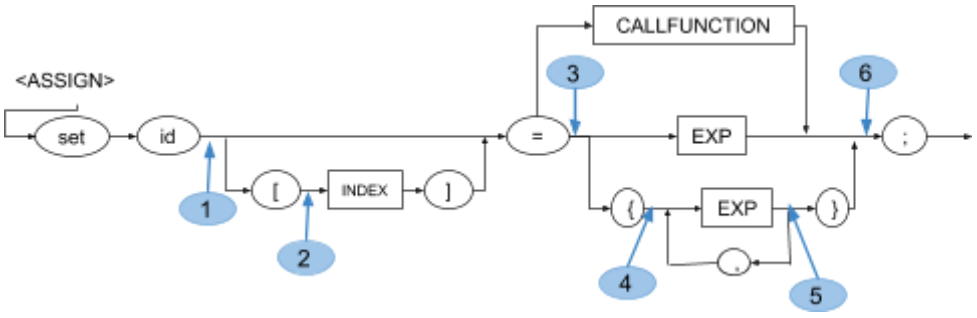
Declare



Semantic Actions:

1. Verifies if the variable does not exist. If it doesn't, the variable is added to corresponding variable table.
2. Activates a flag to state that the next token is a variable size not a constant value.
3. Verifies if the variable does not exist. If it doesn't, the variable array is added to corresponding variable table.

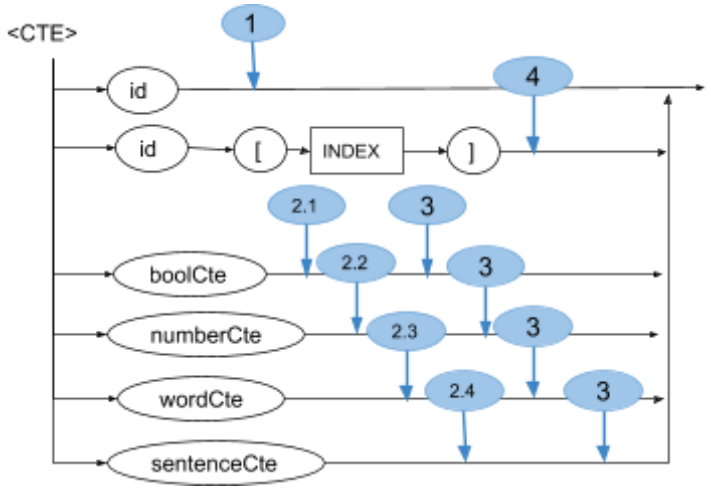
Assign



Semantic Actions:

1. Verifies if id exists. If it does, it adds it to pending operands stack
2. Activates a flag to state that the next token is a variable size not a constant value.
3. Adds operator_assign to pending operator stack.
4. Verifies array size and starts a counter
5. Ends reading of values for array, if there is another value an error is displayed.
6. Gets the operands and operator from the corresponding stacks, verifies result, and adds quadruple for the action.

Constant

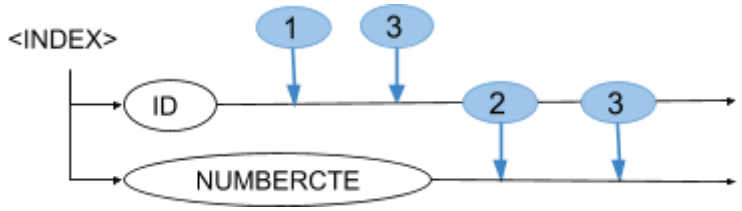


Semantic Actions:

1. Verifies if id exists. If it does, it adds it to pending operands stack
2. Adds type to operand types stack
    2.1 Adds type bool to stack
    2.2 Adds type number to stack
    2.3 Adds type word to stack
    2.4 Adds type sentence to stack
3. Adds constant to pending operands stack
4. Adds index to pending operands stack and checks if its type is correct

---

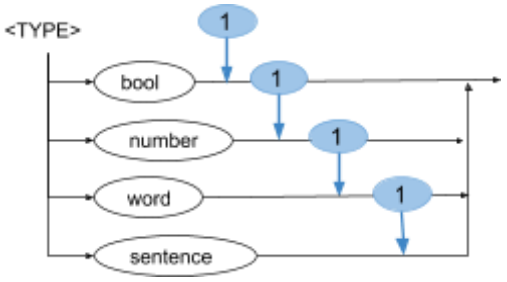
Index



Semantic Actions:

1. Verifies if id exists. If it does, it adds it to pending operands stack
2. Adds type and constant value to corresponding stacks
3. Adds index accessed to global scope's index var

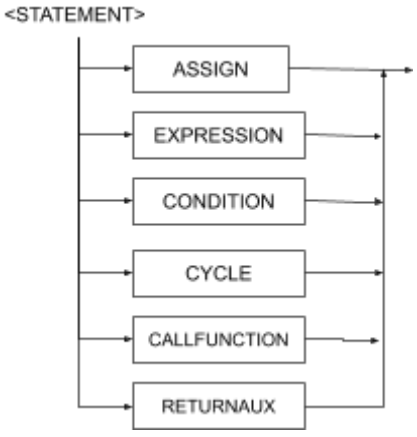
Type



Semantic Actions:

4. Sets global scope variable varType to corresponding type.

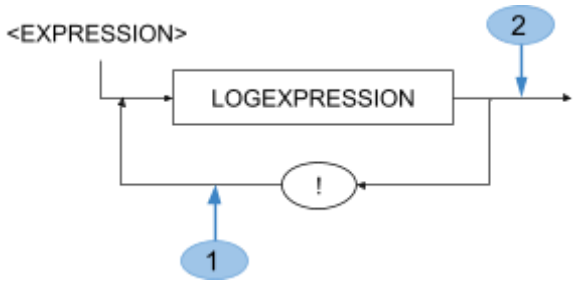
Statement



Semantic Actions:

No actions.

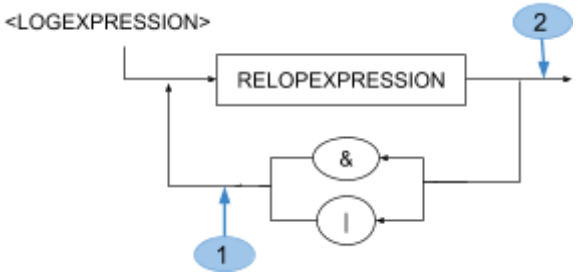
Expression



Semantic Actions:

1. Adds operator_not to pending operator stack
2. Gets operands and operator form corresponding stacks, verifies result, and creates quadruple for action.

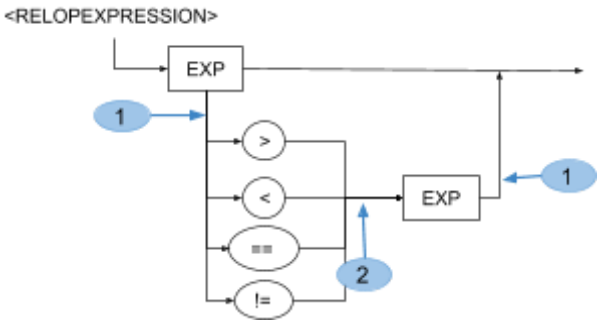
Logexpression



Semantic Actions:

1. Adds operator to pending operator stack.
2. Gets operands and operator form corresponding stacks, verifies result, and creates quadruple for action.

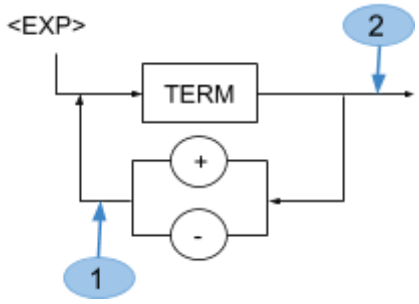
Relopexpression



Semantic Actions:

1. Gets operands and operator form corresponding stacks, verifies result, and creates quadruple for action.
2. Adds operator to pending operator stack.

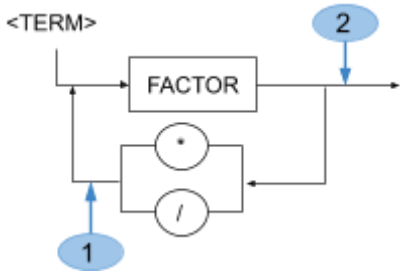
Exp



Semantic Actions:

1. Adds operator to pending operator stack.
2. Gets operands and operator form corresponding stacks, verifies result, and creates quadruple for action.

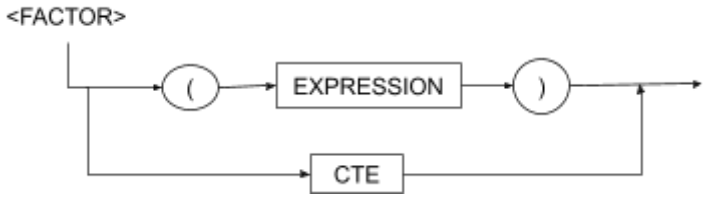
Term



Semantic Actions:

1. Adds operator to pending operator stack.
2. Gets operands and operator form corresponding stacks, verifies result, and creates quadruple for action.

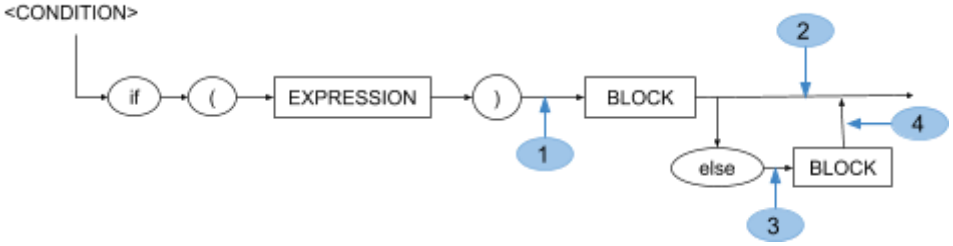
Factor



Semantic Actions:

No actions.

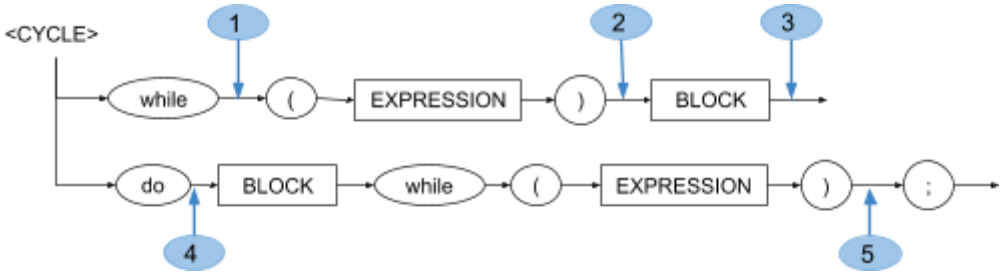
Condition



Semantic Actions:

1. Verifies that expression is of type bool, creates quadruple with operator_gotoF, and adds quadruple count -1 to pending jumps stack.
2. Sets current quadruple result to top of pending jumps stack.
3. Creates a quadruple with operator_goto, sets current quadruple result to top of pending jumps, adds quadruple count -1 to pending jumps stack.
4. Sets current quadruple result to top of pending jumps stack.

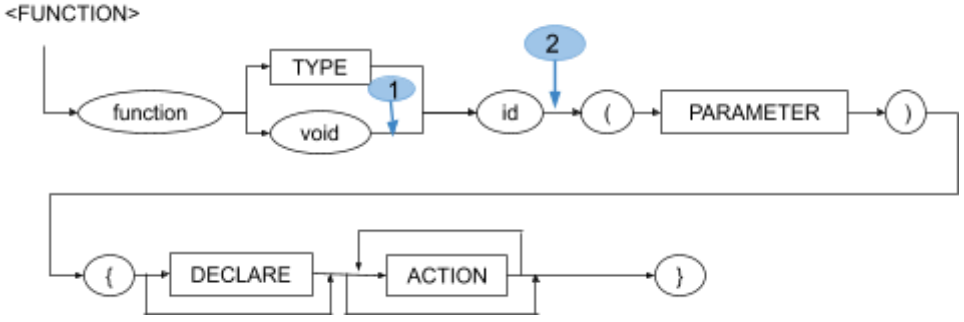
Cycle



Semantic Actions:

1. Adds current quadruple count to pending jumps stack.
2. Verifies that expression is of type bool, creates quadruple with operator_gotoF, and adds quadruple count -1 to pending jumps stack.
3. Gets last two pending jumps, and fills out the result for the quadruple that returns to the while expression and the one where the cycle ends.
4. Adds current quadruple count to pending jumps stack.
5. Verifies that expression is of type bool, creates quadruple with operator_gotoT and result equal to the top of the pending jumps stack.

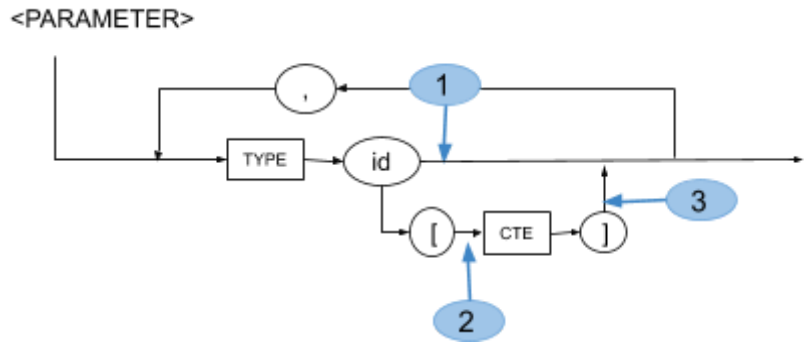
Function



Semantic Actions:

1. Sets global scope's variable "type" to void.
2. Verifies if function id already exists, if it doesn't it adds the id to the directory.

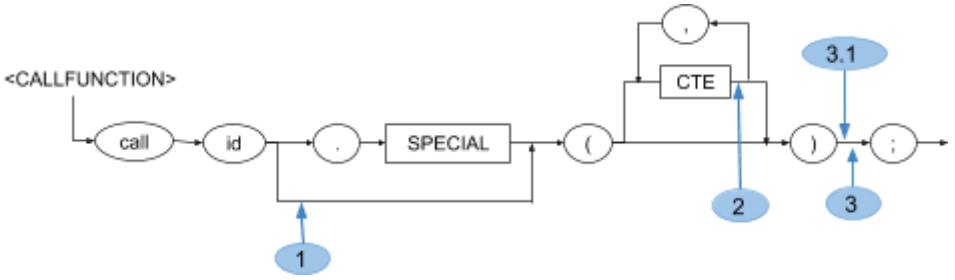
Parameter



Semantic Actions:

1. Adds id to function's directory.
2. Activates a flag to state that the next token is a variable size not a constant value.
3. Adds array to function's directory.

Callfunction

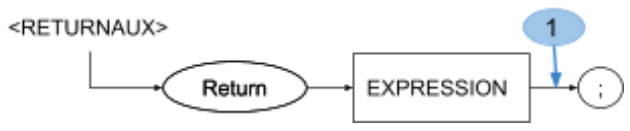


Semantic Actions:

1. Creates activation record for function.
2. Generates param quadruple.
3. Generates go sub quadruple for function.

3.1 Adds quadruple to initiate special function's action.

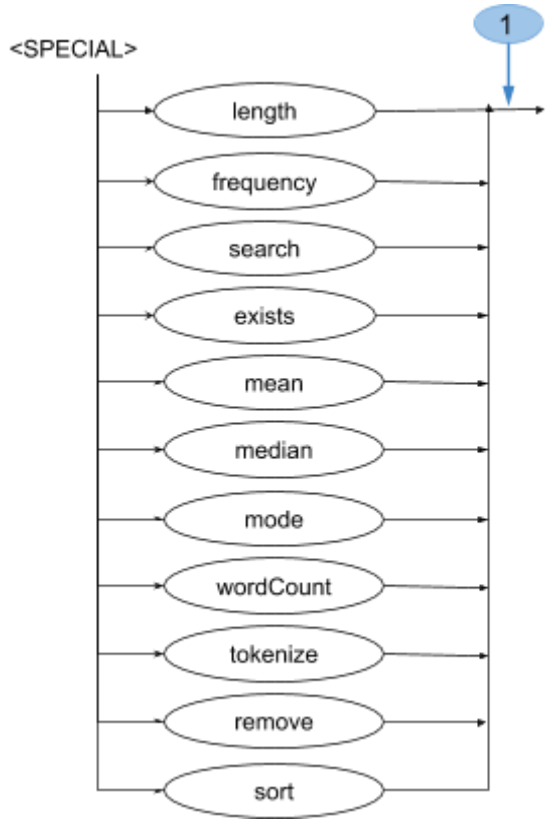
Returnaux



Semantic Actions:

1. Gets return value from functions, and creates corresponding quadruple

Special



Semantic Actions:

1. Sets global scope's variable Special with special function's name and activates flag isSpecial.

Semantic Characteristics (semantic Cube)

Operations in Kotoba can only be applied to data of the same type. The following structure was created to state the semantic characteristics of every operation.

Semantic Cube		
Operand Type (Left and right operands)	**Operator**	**Result**
Number	=	Number
Number	+ | - | * | /	Number
Number	> | < | == | !=	Bool

Word	=	Word
Word	+	Sentence
Word	== | !=	Bool
Sentence	=	Sentence
Sentence	== | !=	Bool
Bool	!	Bool
Bool	=	Bool
Bool	&	Bool
Bool	|	Bool

Memory Management Description

Function Directory Class			
Functions	**Global Memory**	**Local Memory**	**Constant Memory**
Dictionary	String	Dictionary	List

Function Directory Reference Table				
Key	**Value**			
Function Name	**Return Type**	**Variables**	**Parameters**	**Quad Position**
String	String	Dictionary	List	Int

Variable Reference Table			
Key	**Value**		
Variable Name	**Variable Type**	**Variable Size**	**Variable Address**
String	String	Dictionary	List

Quadruple Class			
Operator	**Left Operand**	**Right Operand**	**Result**
String	Int	Int	Int

Virtual Machine Description

Memory Management in Execution

The following Memory Class is the template for all our different memory types.

Memory Class	
Variable	**Type**
name	String
initial_address	Int
final_address	Int
number_memory	Dictionary
word_memory	Dictionary
sentence_memory	Dictionary
bool_memory	Dictionary
slots	int
num_slots	int
word_slots	int
sentence_slots	int
bool_slots	int

The Activation Record Class is used to store the current local memory. When a new function is called a new activation record with an empty era_memory is created.

Activation Record Class	
Variable	**Type**
era_memory	Memory

Virtual Addresses (Compilation) vs. Real Addresses (Execution)

The virtual address established in the quadruples during compilation represents the same address accessed inside the Virtual Machine during execution.

Test Cases

TC1. Print statement

Test Description: Tests a print statement from the main function.

Code

```
kotoba program1;
begin
{
    kprint(“Hello World!”);
}
end
```

Intermediate Code Generation Results

```
Compilation succeeded
My quads are:
1   operator_goto   -1     -1     2
2   operator_print  4500   -1     -1
3   end             -1     -1     -1
```

Execution Results

```
"Hello World!"

----------------------------
Execution Successful
```

TC2. Variable declaration and assignment

Test Description: Assigns a value to a declared variable and prints it.

Code

```
kotoba program1;
declare number x, bool b, word w, sentence s;
begin
{
    set x = 15.0;
    set b = false;
    set w = "Hello";
    set s = "Welcome to Kotoba.";
```

```
    kprint(x);
    kprint(b);
    kprint(w);
    kprint(s);
}
end
```

Intermediate Code Generation Results

```
Compilation succeeded
My quads are:
1   operator_goto   -1     -1     2
2   operator_assign 4000   -1     1000
3   operator_assign 4750   -1     1750
4   operator_assign 4250   -1     1250
5   operator_assign 4500   -1     1500
6   operator_print  1000   -1     -1
7   operator_print  1750   -1     -1
8   operator_print  1250   -1     -1
9   operator_print  1500   -1     -1
10   end              -1    -1     -1
```

Execution Results

```
15.0
false
"Hello"
"Welcome to Kotoba."

----------------------------
Execution Successful
```

TC3. Read statement

Test Description: Asks for input and prints variable.

Code

```
kotoba program1;
declare word name;
begin
{
    kread(name);
    kprint("Hello");
    kprint(name);
}
end
```

Intermediate Code Generation Results

```
Compilation succeeded
My quads are:
1   operator_goto   -1     -1     2
2   operator_read   word   -1     1250
3   operator_print  4250   -1     -1
4   operator_print  1250   -1     -1
5   end             -1     -1     -1
```

Execution Results

```
Enter input: Carolina
"Hello"
Carolina

------------------------------
Execution Successful
```

TC4. Arithmetic operations

Test Description: Performs all basic arithmetic operations and displays results.

Code

```
kotoba program1;
declare number x, number y;
begin
{
    set x = 10.0;
    set y = x / 2.0;
    set x = x + 10.0;
    set y = y * 3.0;
    kprint(x);
    kprint(y);
}
end
```

Intermediate Code Generation Results

```
Compilation succeeded
My quads are:
1   operator_goto   -1     -1     2
2   operator_assign 4000   -1     1000
3   operator_div    1000   4001   2000
4   operator_assign 2000   -1     1001
5   operator_add    1000   4000   2001
6   operator_assign 2001   -1     1000
7   operator_mult   1001   4002   2002
```

```
8   operator_assign 2002   -1     1001
9   operator_print  1000   -1     -1
10   operator_print 1001   -1     -1
11   end             -1    -1     -1
```

Execution Results

```
20.0
15.0

------------------------------
Execution Successful
```

TC5. Expressions and Conditions

Test Description: Assigns value to a variable and performs an if and if/else condition.

Code

```
kotoba program1;
declare number x, number z, sentence s;
begin
{
    set x = 4.0;
    if(x > 2.0) {
       kprint(x);
    }

    if(x < 2.0){
        kprint(x);
    }else{
        kprint("X is greater than 2.0");
    }
}
end
```

Intermediate Code Generation Results

```
Compilation succeeded
My quads are:
1   operator_goto   -1     -1     2
2   operator_assign 4000   -1     1000
3   operator_greater       1000   4001   3500
4   operator_gotoF  3500   -1     6
5   operator_print  1000   -1     -1
6   operator_less   1000   4001   3501
7   operator_gotoF  3501   -1     10
```

```
8   operator_print  1000   -1     -1
9   operator_goto   -1     -1     11
10   operator_print 4500   -1     -1
11   end            -1     -1     -1
```

Execution Results

```
4.0
"X is greater than 2.0"

----------------------------
Execution Successful
```

TC6. Cycles

Test Description: Prints fibonacci sequence for number 7 using a while cycle.

Code

```
kotoba program1;
declare number n, number a, number b, number aux, number i;
begin
{
     set n = 7.0;
     set a = 0.0;
     set b = 1.0;
     set i = 2.0;
     kprint(a);
     kprint(b);

     while(i < (n + 1.0)) {
        set aux = a + b;
        set a = b;
        set b = aux;
        set i = i + 1.0;
        kprint(aux);
     }
}
end
```

Intermediate Code Generation Results

```
Compilation succeeded
My quads are:
1   operator_goto        -1      -1     2
2   operator_assign     4000     -1     1000
```

```
3   operator_assign     4001      -1     1001
4   operator_assign     4002      -1     1002
5   operator_assign     4003      -1     1004
6   operator_print      1001      -1     -1
7   operator_print      1002      -1     -1
8   operator_add        1000      4002   2000
9   operator_less       1004      2000   3500
10   operator_gotoF     3500      -1     19
11   operator_add       1001      1002   2001
12   operator_assign    2001      -1     1003
13   operator_assign    1002      -1     1001
14   operator_assign    1003      -1     1002
15   operator_add       1004      4002   2002
16   operator_assign    2002      -1     1004
17   operator_print     1003      -1     -1
18   operator_goto       -1       -1     8
19   end                 -1       -1     -1
```

Execution Results

```
0.0
1.0
1.0
2.0
3.0
5.0
8.0
13.0

------------------------------
Execution Successful
```

TC7. Function declaration and call

Test Description: Declares function “sum” and “one”. From main it calls function one with value 3, in function one it calls function sum and returns the result times 2.

Code

```
kotoba program1;
declare number x, number y;
function number sum(number x){
    return (x + 1.0);
}
function number one(number n) {
    declare number a;
    set a = call sum(n);
    return (a*2.0);
}
```

```
begin
{
    set x = 3.0;
    set y = call one(x);
    kprint(y);
}
end
```

Intermediate Code Generation Results

```
My quads are:
1   operator_goto     -1      -1      10
2   operator_add      2000    4000    2001
3   operator_return   -1      -1      2001
4   operator_era      -1      -1      sum
5   operator_param    2002    -1      param1
6   operator_gosub    -1      -1      sum
7   operator_assign   (sum)   -1      2003
8   operator_mult     2003    4001    2000
9   operator_return   -1      -1      2000
10   operator_assign  4002    -1      1000
11   operator_era     -1      -1      uno
12   operator_param   1000    -1      param1
13   operator_gosub   -1      -1      uno
14   operator_assign  (one)   -1      1001
15   operator_print   1001    -1      -1
16   end               -1     -1      -1
```

Execution Results

```
8.0

------------------------------
Execution Successful
```

TC8. Array manipulation

Test Description: Creates a number and word array, assigns values and prints an index's value.

Code

```
kotoba program1;
declare word w[4.0], number arr[5.0];
begin
{
     set w = {"apple", "banana", "orange", "grape"};
     set arr = {2.0, 34.5, 67.56, 4.1, 0.12};
     kprint(w[1.0]);
     kprint(arr[3.0]);
```

```
}
end
```

Intermediate Code Generation Results

```
Compilation succeeded
My quads are:
1   operator_goto          -1      -1     2
2   operator_assign      4250      -1     1250
3   operator_assign      4251      -1     1251
4   operator_assign      4252      -1     1252
5   operator_assign      4253      -1     1253
6   operator_assign      4000      -1     1000
7   operator_assign      4001      -1     1001
8   operator_assign      4002      -1     1002
9   operator_assign      4003      -1     1003
10   operator_assign     4004      -1     1004
11   operator_verify       0       4      1.0
12   operator_address     1.0     1250    1251
13   operator_print       1251      -1     -1
14   operator_verify       0       5      3.0
15   operator_address     3.0      1000   1003
16   operator_print       1003      -1     -1
17   end                   -1       -1     -1
```

Execution Results

```
"banana"
4.1

------------------------------
Execution Successful
```

TC9. Recursion

Test Description: Solves fibonacci sequence for number 10.

Code

```
kotoba program1;
declare number x, number f;
function number fib(number n) {
    declare number a, number aAux, number b, number bAux;
    if(n < 2.0){
       return n;
    }else{
       set aAux = n - 1.0;
       set bAux = n - 2.0;
       set a = call fib(aAux);
       set b = call fib(bAux);
```

```
       return (a + b);
     }
}
begin
{
     set x = 10.0;
     set f = call fib(x);
     kprint(f);
}
end
```

Intermediate Code Generation Results

```
Compilation succeeded
My quads are:
1   operator_goto        -1      -1     20
2   operator_less      2000      4000   3500
3   operator_gotoF     3500      -1     6
4   operator_return     -1       -1     2000
5   operator_goto       -1       -1     20
6   operator_minus     2000      4001   2005
7   operator_assign    2005      -1     2002
8   operator_minus     2000      4000   2006
9   operator_assign    2006      -1     2004
10   operator_era       -1       -1     fib
11   operator_param    2002      -1     param1
12   operator_gosub     -1       -1     fib
13   operator_assign  (fib)      -1     2001
14   operator_era       -1       -1     fib
15   operator_param    2004      -1     param1
16   operator_gosub     -1       -1     fib
17   operator_assign   (fib)     -1     2003
18   operator_add       2001     2003   2000
19   operator_return    -1       -1     2000
20   operator_assign   4002      -1     1000
21   operator_era       -1       -1     fib
22   operator_param    1000      -1     param1
23   operator_gosub     -1       -1     fib
24   operator_assign   (fib)     -1     1001
25   operator_print    1001      -1     -1
26   end                -1       -1     -1
```

Execution Results

```
55.0

------------------------------
Execution Successful
```

TC10. Special functions use

Test Description: Declares a sentence and stores its words in an array using our special function tokenize.

Code

```
kotoba program1;
declare word w[4.0], number n, sentence s;
begin
{
    set s = "The dog is running.";
    set w = call s.tokenize();
    set n = 0.0;
    set s = w[n] + w[1.0];

    kprint(w[n]);
    kprint(s);
}
end
```

Intermediate Code Generation Results

```
Compilation succeeded
My quads are:
1   operator_goto        -1       -1     2
2   operator_assign     4500      -1     1500
3   operator_special    1500      w      tokenize
4   operator_assign   (tokenize)  -1     1250
5   operator_assign      4000     -1     1000
6   operator_verify       0       4      1000
7   operator_address     1000     1250   4001
8   operator_verify       0       4      1.0
9   operator_address     1.0      1250   1251
10   operator_add        4001     1251   3000
11   operator_assign     3000     -1     1500
12   operator_verify       0      4      1000
13   operator_address    1000     1250   4003
14   operator_print      4003     -1     -1
15   operator_print      1500     -1     -1
16   end                  -1      -1     -1
```

Execution Results

```
"The"
"The dog"

------------------------------
Execution Successful
```

TC11. Compilation Error

Test Description: Test for undeclared id

Code

```
kotoba program1;
declare number x;
begin
{
    kprint(y);
}
end
```

Intermediate Code Generation Results

```
ID y does not exist
```

Execution Results

```
None
```

TC12. Runtime Error

Test Description: Tests a division by 0.

Code

```
kotoba program1;
declare number x;
begin
{
    set x = 4.0/0.0;
}
end
```

Intermediate Code Generation Results

```
Compilation succeeded
My quads are:
1   operator_goto     -1     -1     2
2   operator_div      4000   4001   2000
3   operator_assign   2000   -1     1000
4   end               -1     -1     -1
```

Execution Results

```
Unable to perform division by 0
```

Documentation Comments

Important Modules

The following list of Modules represent the most important modules of our compiler.

Name: addVariable
Location: directory.py --- class Directory
Description: This modules verifies that a variable can be added, and it gets the next free address for that variable's type.
Parameters: function name, variable name, variable type, variable size
Output: returns true if the variable was added successfully and false otherwise.
Modules that use this module: in yaccKotoba.py → func_declare_var, func_declar_array, and func_declare_parameter
Code:

```
# Adds variable to corresponding function's var table
   def addVariable(self, functionName, varName, varType, varSize):
       if varName in self.functions[functionName][1]:
           return False
       else:
           if functionName == "Main":
               # Adds variable to global memory
               varAddress = self.global_memory.get_nextAddress(varType)
               if varSize > 1:
                   # Assigns memory address for every value in the array
                   for i in range(varSize-1):
                       self.global_memory.reserve_ListAddress(varType)
           else:
               # Adds variable to local memory
               varAddress = self.local_memory.get_nextAddress(varType)
               if varSize > 1:
                   for i in range(varSize-1):
                       self.local_memory.reserve_ListAddress(varType)

           varData = [varType, varSize, varAddress]
           # Add variable to function's variable dictionary
           # key: FunctionName, 1: position 1 of functions data, varName: key for variables dictionary
           self.functions[functionName][1][varName] = varData

           return True
```

Name: get_ValueForAddress
Location: memory.py -- Class Memory
Description: This module returns the value stored for the requested address
Parameters: address
Modules that use this module: In directory.py → getVarValue
Code:

```
# Returns variable;s value
def get_ValueForAddress(self, address):
       # Checks address range to determine which memory to access
       if address >= self.initial_address and address < self.word_slot: # Number memory
           if address in self.number_memory:
               return self.number_memory[address]
```

```
        if address >= self.word_slot and address < self.sentence_slot: # Word memory
            if address in self.word_memory:
                return self.word_memory[address]

        if address >= self.sentence_slot and address < self.bool_slot:  # Sentence memory
            if address in self.sentence_memory:
                return self.sentence_memory[address]

        if address >= self.bool_slot and address <= self.final_address:  # Bool memory
           if address in self.bool_memory:
                return self.bool_memory[address]

        return -1
```

Name: p_func_declare_function
Location: YaccKotoba.py --
Description: This module adds a function id to our function directory.
Parameters: p (corresponds to current token in grammar)
Modules that use this module: In YaccKotoba.py → p_function
Code:

```
# Add a new function to the function directory
def p_func_declare_function(p) :
    'func_declare_function : '
     # Verify is function already exists
    if globalScope.functionDirectory.addFunction(p[-1], globalScope.varType, globalScope.quadCount) :
        globalScope.functionName = p[-1]
    else :
        sys.exit("Error: Function ID " + p[-1] + " already exists")
```

Name: era_operation
Location: virtualMachine.py
Description: This module is in charge of creating an activation record everytime a new function is called.
Parameters: current_quad
Modules that use this module: In virtualMachine.py → execute_program
Code:

```
# Create activation record for function called
def era_operation(current_quad):
   current_function_name.push(current_quad.getResult())
   current_era = ActivationRecord(globalScope.functionDirectory.local_memory) # Store current activation record
   local_memory_handler.push(current_era)  # Add that era to local memory stack

   new_memory = Memory("Local/Temporal", 2000, 3999)  # Create new memory
   new_era = ActivationRecord(new_memory)  # Create new era for function called
   local_memory_handler.push(new_era)  # add that era to local memory stack
```

Name: goto_operation
Location: virtualMachine.py
Description: This module handles the goto jumps for goto, goto false and goto true.
Parameters: current_quad
Modules that use this module: In virtualMachine.py → execute_program
Code:

```
# Goto operations
def goto_operation(operator, current_quad):
   goto_jump = current_quad.getResult() - 1 #gets jump from quad result
   if operator == "operator_goto":
       return goto_jump # if operator is goto, it returns the jump
   elif operator == "operator_gotoF":
       # Gets expression evaluated address and value
       exp_address = current_quad.getLeftOperator()
       exp_value = globalScope.functionDirectory.getVarValue(exp_address)
       if exp_value == None: # Returns error if value is none
           sys.exit("Variable has no value to be evaluated with")
       if exp_value == "false": # If the expression is false it returns the jump
           return goto_jump
       else:
           return -1 # If the expression is true it returns -1 and instruction pointer continues to next quad
   elif operator == "operator_gotoT":
       # Gets expression evaluated address and value
       exp_address = current_quad.getLeftOperator()
       exp_value = globalScope.functionDirectory.getVarValue(exp_address)
       if exp_value == None: # Returns error if value is none
           sys.exit("Variable has no value to be evaluated with")
       if exp_value == "true": # If the expression is true it returns the jump
           return goto_jump
       else:
           return -1 # If the expression is false it returns -1 and instruction pointer continues to next quad
```

User Manual

Quick Reference Manual

Input Statement

To set a variable's value from the terminal use the following statement:

kread(varName);

Output Statement

To print a variable use the following statement:

kprint(varName);

Variable Types

Number: a negative or positive float number
Word: a single word
Sentence: a set of two or more words
Bool: true or false

Declaration and assignment of variables

To declare a variable use the following statement:

declare varType varName;

To set a variable's value use the following statement:

Set varName = varValue;

Declaration and assignment of arrays

To declare an array variable use the following statement:

declare varType varName[varSize];

To set a value to a specific index of an array use the following statement:

Set varName[index] = indexValue;

*note: the index value must be a number constant or a predefined number variable

To set all values to an array of size n use the following statement:

Set varName = {value1, value2, …, valueN}

Accepted operations

Arithmetic operations can only be performed between data types of the same type.

Conditions

To start a simple condition use the following statement:

```
if (expression){
    actions…
}
```

To start a double action condition use the following statement:

```
if (expression){
    actions…
}else{
    actions…
}
```

Cycles

To start a while cycle use the following statement:

```
While (expression){
    actions...
}
```

To start a do while cycle use the following statement:

```
Do {
    actions...
}while (expression);
```

Function declarations calls

To declare a function with a returnType use the following statement:

```
Function returnType functionName(parameterType parameter*){
    declaration of variables...
```

actions…

return returnValue;
}

*To send more than one parameter, use the same syntax and separate parameters with a comma.

To declare a void function with use the following statement:

Function void functionName(parameterType parameter*){
declaration of variables...

actions…

return “void”;
}

To call a function use the following statement:

call functionName(parameter);

Special Functions

To call a special function use the following statement:

call varName.specialFunctionName();
or call varName.specialFunctionName(parameter);

Video-Demo

Link to access video: https://youtu.be/1b4r1LljYG8