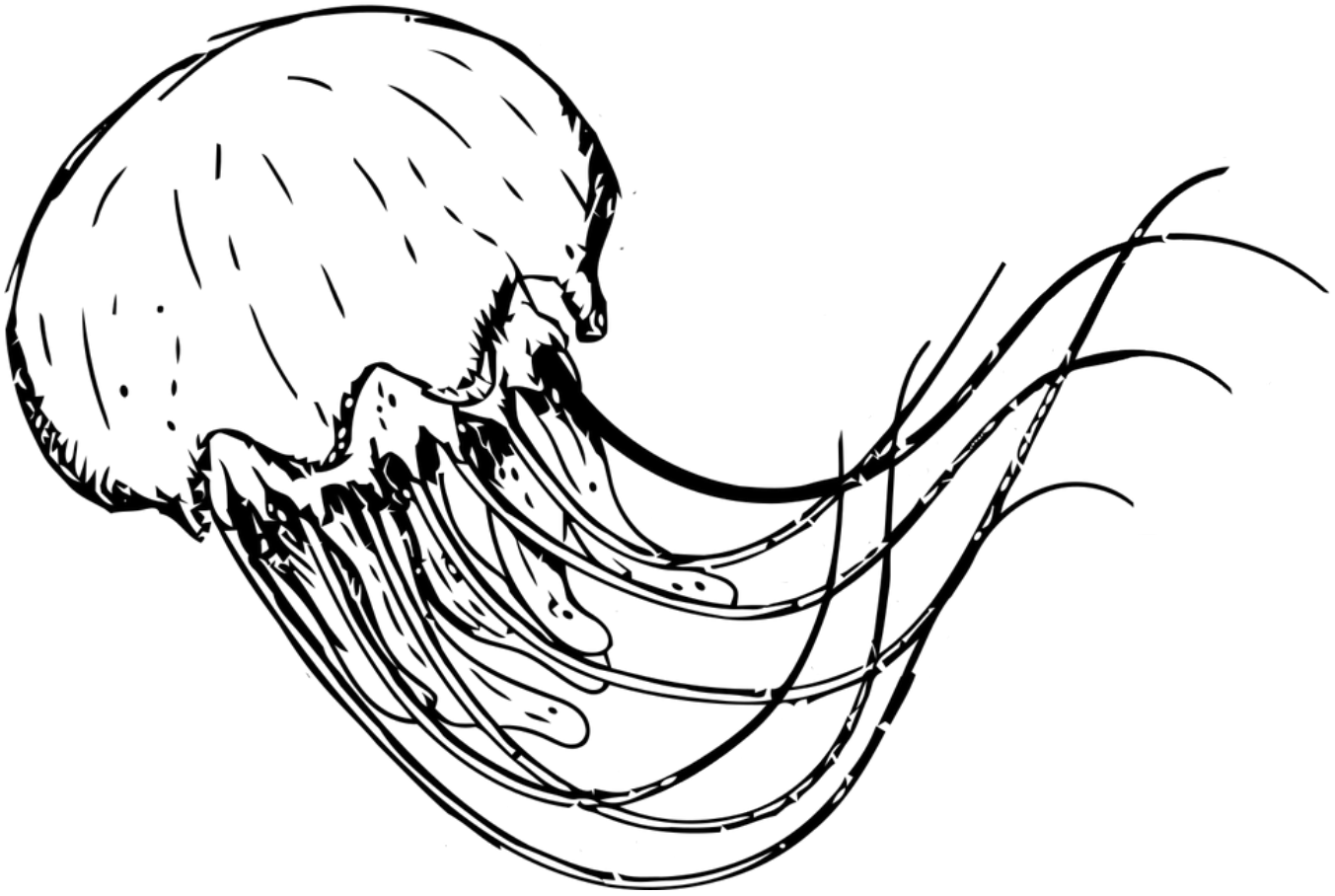
A guide to getting started with JPP



JPP

Object Oriented Language

Oscar Rodríguez
Patricio Güereque

ITESM

Contents

List of Figures

1 Project Description

1.1 Project scope and goals

The project consists of designing and implementing a programming language in our language of choice. The programming language will compile into an intermediate representation and our own virtual machine will interpret the intermediate representation.

The aim of the programming language will be to support callable routines (functions), module imports, object oriented paradigm, unlimited nested scopes (global, local and temporary), one and two dimensional arrays, condition flows, and full expression support with **arithmetic**, **boolean** and **bit-wise** operators.

1.2 Requirements analysis and test cases

The concrete requirements/features JPP aims to contemplate in its scope are:

- Complex arithmetic expressions
- I/O support
- Conditional flows
- Iteration mechanisms (while-loop, for-loop)
- Bit-wise operations
- Subroutines with arguments
- Module imports
- Native functions
- Execution optimization
- Simple Classes (Objects)

The following test cases aim to correctly evaluate and display the full range of support JPP offers:

- Printing "Hello World!" into the console
- Basic if/else statement
- Printing the results of different arithmetic operations
- Printing the results of different bit-wise operations
- Filling out a matrix with a for loop and printing then printing out the values of said matrix
- Basic Fibonacci sequence to test recursive functions
- Fibonacci sequence with dynamic programming to test the capabilities of arrays
- Create a linked list of nodes and iterate over it

1.3 Development Workflow

The project was built relying on state of the art technologies such as Git and GitHub. The repository for the official JPP programming language can be found in: `https://github.com/dmosc/jpp`.

1.3.1 Commits

Small changes

Small changes, like hot-fixes, grammar patches, and minor miscellaneous changes that are not relevant to program functionality were directly committed into the main branch.

Feature changes

New features and/or major re-factorization changes are worked in separated branches that must pass tests, and be peer-reviewed by another JPP team member before merging.

1.3.2 Change log

Date	Description
Sun Apr 10 20:38:11 2022	Add first iteration of grammar.
Thu Apr 14 10:28:52 2022	Support function_call with no semicolon inside expressions.
Mon Apr 18 13:12:32 2022	Implement code comment regex
Mon Apr 18 20:10:02 2022	Add Quadruples structure.
Tue Apr 19 20:10:54 2022	Add quadruples processing algorithm.
Wed Apr 20 07:57:50 2022	Add support for quad jumps in conditional statements.
Thu Apr 21 19:14:06 2022	Add GOTO, GOTO_F, GOTO_T opcodes
Sat Apr 23 15:29:19 2022	Add support for variable storage with N scopes.
Sun Apr 24 13:07:39 2022	Add function registering and scope management.
Sun Apr 24 18:32:07 2022	Add function call and return support with parameter initialization.
Sun Apr 24 19:06:08 2022	Add INIT opcode to mark beginning of execution tape.
Sun May 1 21:34:10 2022	(WIP) Start writing optimizer
Sun May 1 21:34:11 2022	Change function calls from GOTO to CALL
Sun May 1 21:34:11 2022	Constant folding optimizer
Thu May 12 19:33:10 2022	Add MemoryManager with memory segments.
Thu May 12 20:51:08 2022	Memory manager v2 using bit manipulation
Sat May 14 09:56:33 2022	Add JumpsManager to handle jumps.
Mon May 16 20:22:20 2022	Add malloc array/matrix support.
Fri May 20 10:37:35 2022	Add imports to grammar syntax
Fri May 20 12:05:02 2022	Basic import system
Mon May 23 00:19:43 2022	Add EXIT Opcode
Thu May 26 13:25:56 2022	More robust cli, still needs work but allows for more debugging
Thu May 26 16:24:43 2022	Virtual Machine! (Interpreter)
Fri May 27 00:01:15 2022	Native function support
Fri May 27 16:40:15 2022	Implement more native functions
Fri May 27 17:22:52 2022	Working spinning donut (torus)

1.3.3 Individual learning

Oscar D. Rodríguez Moscosa

Building a programming language from scratch is nothing less than a daunting task. Having the opportunity to dissect and understand the ins and outs of the entire process of doing so was extremely enriching. I definitely finish this project a better engineer in contrast to how I started.

Learning about the different step layers on what is required; from processing text, and understanding it, to designing memory management strategies requires maintaining an array of knowledge stacked from multiple years of courses, competitions, and experiences.

The work Patricio and I did for the project is an homage to engineering

excellence and teamwork. Having undergone all these years of study and gaining knowledge **had** to culminate in a project with sufficient breadth and depth to max out our skills and we felt the responsibility to challenge ourselves as such.

Patricio Güereque

Compilers have always fascinated me, how could a file end up being interpreted by the computer. Before taking compilers course, I had a very limited knowledge of compilers. Even though we didn't cover everything in the course, it immensely helped understand the internals of some programming language, limits and challenges that come within them.

Memory management is a very important subject when programming. A lot of modern programming languages have garbage collectors and the compilers are smarter than the best programmers out there. C on the other hand, has a very delicate memory. If you are not careful you can end up with buffer overflows, effectively overriding other parts of the memory and be subject to a Buffer Overflow exploitation. C also has memory pointers, which is essentially variables that point to other addresses. Comparing this to Java, Java internally manages all objects as pointers while in C when you pass an object as a value, it makes a copy of the object's memory. This was fundamental knowledge when creating our own memory manager.

Something that I learned from this course was the syntax and lexical parsing process, as well as syntatic and semantic analysis. I knew the existence of an intermediate representation, by poking around Java Bytecode with ASM and dissemblers, but it's fascinating learning how there are so many ways of representing code. Java uses a Stack Machine and we used a 3-address-code structure, which we learned from the course. Both have their pros and cons, but both are very interesting to work with.

2 Language Description

2.1 Language

The name for the language is **J++** (or **JPP**) after C++ and JavaScript as it feels like a mixture between both programming languages in terms of syntax, behavior and tooling experience.

2.2 Description

J++ is an object oriented programming language aimed at offering commonly found mechanism most major modern languages offer such as: **arithmetic expressions**, **native iteration syntax**, **subroutines**, **conditional flows**, **multi-dimensional variables**, **module creation and imports**, etc.

Additionally, during intermediate code generation, J++ implements code optimization strategies to compress object code output and subsequently improve execution times.

2.3 Expected unexpected behavior

- Arithmetic operations do not support single operand negation (i.e. -5)
  - **Workaround**: Use 0-x (ex. 0-5 will equal to -5)
- Access to uninitialized object attributes does not yield errors
  - **Workaround**: Have an attribute that indicates if the object is initialized or not

3 Compiler Description

3.1 Technologies and libraries

The J++ compiler is written in native JavaScript and thought for common Linux environments (although it can technically run in any operating system capable of running Node.js) and is packaged as an NPM project to import the following external libraries:

- jison
- datastructures-js
- commander
- chalk
- prompt-sync

3.2 Lexical analysis

3.2.1 Tokens

Table 3.1: Tokens and Regular Expressions

Regex	Token
".*?"	CONST_STRING
`/]{2}(.|\n|\r)+?/]2`	COMMENTS
"«"	BITWISE_LEFT_SHIFT
"»"	BITWISE_RIGHT_SHIFT
"=="	EQUALS
"!="	NOT_EQUALS
"<="	LTE
">="	GTE
"<"	LT

Continued on next page

Table 3.1 – continued from previous page

Regex	Token
">"	GT
"||"	BOOLEAN_OR
"&&"	BOOLEAN_AND
"!"	BOOLEAN_NOT
"|"	BITWISE_OR
"&"	BITWISE_XOR
"^"	BITWISE_AND
" "	BITWISE_NOT
"+"	PLUS
"-"	MINUS
"*"	MULTIPLICATION
"/"	DIVISION
"%"	MODULO
"="	ASSIGN
"("	OPEN_PARENTHESIS
")"	CLOSE_PARENTHESIS
"{"	OPEN_CURLY_BRACKET
"}"	CLOSE_CURLY_BRACKET
"["	OPEN_SQUARE_BRACKET
"]"	CLOSE_SQUARE_BRACKET
","	COMMA
";"	SEMICOLON
":"	COLON
"."	DOT
"if"	IF
"elif"	ELIF
"else"	ELSE
"return"	RETURN
"for"	FOR
"while"	WHILE
"class"	CLASS
"extends"	EXTENDS
"construct"	CONSTRUCT
"destruct"	DESTRUCT
"void"	VOID
"program"	PROGRAM
"func"	FUNC
"var"	VAR
"native"	NATIVE
"import"	IMPORT
("int"|"bool")	INT
"float"	FLOAT
"string"	STRING
[\s\t\n\r]+	<new line/space>
0-9]+\.0-9]+	CONST_FLOAT
[0-9]+	CONST_INT

Continued on next page

Table 3.1 – continued from previous page

Regex	Token
[A-Za-z_][A-Za-z0-9_]*	ID

3.3 Syntax analysis

3.3.1 Grammar rules

⟨*arithmetic_op_l1*⟩ ::= PLUS
| MINUS;

⟨*arithmetic_op_l2*⟩ ::= MULTIPLICATION
| DIVISION
| MODULO;

⟨*relational_op_l1*⟩ ::= EQUALS
| NOT_EQUALS;

⟨*relational_op_l2*⟩ ::= LT
| LTE
| GT
| GTE;

⟨*bitwise_op_l1*⟩ ::= BITWISE_OR;

⟨*bitwise_op_l2*⟩ ::= BITWISE_XOR;

⟨*bitwise_op_l3*⟩ ::= BITWISE_AND;

⟨*bitwise_op_l4*⟩ ::= BITWISE_LEFT_SHIFT
| BITWISE_RIGHT_SHIFT;

⟨*bitwise_op_l5*⟩ ::= BITWISE_NOT;

⟨*boolean_op_l1*⟩ ::= BOOLEAN_OR;

⟨*boolean_op_l2*⟩ ::= BOOLEAN_AND;

⟨*boolean_op_l3*⟩ ::= BOOLEAN_NOT;

⟨*assignment_op_l1*⟩ ::= ASSIGN;

⟨*type_s*⟩ ::= INT
| FLOAT
| STRING
| BOOL ;

⟨*type_c*⟩ ::= ID;

⟨*const_type*⟩ ::= CONST_INT
| CONST_FLOAT
| CONST_STRING
| CONST_BOOLEAN ;

⟨*program*⟩ ::= ⟨*program_imports*⟩ ⟨*program_1*⟩ ⟨*program_init*⟩
@push_scope ⟨*block*⟩ @pop_scope
| ⟨*native_functions*⟩;

⟨*program_imports*⟩ ::= /* ⟨*empty*⟩ */
| IMPORT OPEN_PARENTHESIS ⟨*imports*⟩
CLOSE_PARENTHESIS;

⟨*imports*⟩ ::= CONST_STRING
| CONST_STRING COMMA ⟨*imports*⟩ ;

⟨*program_init*⟩ ::= PROGRAM ID ;

⟨*program_1*⟩ ::= /* ⟨*empty*⟩ */
| ⟨*function*⟩ ⟨*program_1*⟩
| ⟨*vars*⟩ ⟨*program_1*⟩
| ⟨*class*⟩ ⟨*program_1*⟩;

⟨*block*⟩ ::= OPEN_CURLY_BRACKET ⟨*block_1*⟩
CLOSE_CURLY_BRACKET;

⟨*block_1*⟩ ::= /* ⟨*empty*⟩ */
| ⟨*statement*⟩ ⟨*block_1*⟩;

⟨*params*⟩ ::= OPEN_PARENTHESIS ⟨*params_1*⟩
CLOSE_PARENTHESIS;

⟨*params_1*⟩ ::= /* ⟨*empty*⟩ */
| ⟨*type_s*⟩ ID ⟨*params_2*⟩ ;

⟨*params_2*⟩ ::= /* ⟨*empty*⟩ */
| COMMA ⟨*type_s*⟩ ID ⟨*params_2*⟩ ;

⟨*function*⟩ ::= FUNC ⟨*function_1*⟩ @push_scope ⟨*params*⟩ ⟨*block*⟩
@close_function @pop_scope;

⟨*function_1*⟩ ::= ⟨*function_2*⟩ ID;

⟨*function_2*⟩ ::= ⟨*type_s*⟩
| VOID;

⟨*native_functions*⟩ ::= ⟨*native_function*⟩
| ⟨*native_function*⟩ ⟨*native_functions*⟩;

⟨*native_function*⟩ ::= NATIVE ⟨*native_function_1*⟩
@push_scope ⟨*params*⟩ @close_function
@pop_scope SEMICOLON;

⟨*native_function_1*⟩ ::= ⟨*native_function_2*⟩ ID ;

⟨*native_function_2*⟩ ::= ⟨*type_s*⟩
| VOID;

⟨*variable_declare*⟩ ::= ID
| ID OPEN_SQUARE_BRACKET CONST_INT
CLOSE_SQUARE_BRACKET
| ID OPEN_SQUARE_BRACKET CONST_INT
CLOSE_SQUARE_BRACKET
OPEN_SQUARE_BRACKET CONST_INT
CLOSE_SQUARE_BRACKET ;

⟨*variable*⟩ ::= ID
| ID OPEN_SQUARE_BRACKET ⟨*expression*⟩
CLOSE_SQUARE_BRACKET

| ID OPEN_SQUARE_BRACKET ⟨*expression*⟩
CLOSE_SQUARE_BRACKET
OPEN_SQUARE_BRACKET ⟨*expression*⟩
CLOSE_SQUARE_BRACKET ;

⟨*vars*⟩ ::= VAR ⟨*type_s*⟩ ⟨*variable_declare*⟩ ⟨*vars_1*⟩
SEMICOLON
| VAR ⟨*type_c*⟩ ID ⟨*vars_2*⟩ SEMICOLON;

⟨*vars_1*⟩ ::= /* ⟨*empty*⟩ */
| COMMA ⟨*variable_declare*⟩ ⟨*vars_1*⟩;

⟨*vars_2*⟩ ::= /* ⟨*empty*⟩ */
| COMMA ID ⟨*vars_2*⟩;

⟨*class*⟩ ::= CLASS ID ⟨*class_1*⟩ ⟨*class_block*⟩;

⟨*class_1*⟩ ::= /* ⟨*empty*⟩ */
| EXTENDS ID;

⟨*class_block*⟩ ::= OPEN_CURLY_BRACKET ⟨*class_block_1*⟩
CLOSE_CURLY_BRACKET;

⟨*class_block_1*⟩ ::= /* ⟨*empty*⟩ */
| ⟨*vars*⟩ ⟨*class_block_1*⟩
| ⟨*function*⟩ ⟨*class_block_1*⟩
| ⟨*construct*⟩ ⟨*class_block_1*⟩
| ⟨*destruct*⟩ ⟨*class_block_1*⟩;

⟨*construct*⟩ ::= CONSTRUCT ⟨*params*⟩ ⟨*block*⟩;

⟨*destruct*⟩ ::= DESTRUCT OPEN_PARENTHESIS
CLOSE_PARENTHESIS ⟨*block*⟩;

⟨*assign*⟩ ::= ⟨*variable*⟩ ⟨*assignment_op_l1*⟩ ⟨*expression*⟩ ;

⟨*condition*⟩ ::= IF OPEN_PARENTHESIS ⟨*expression*⟩
CLOSE_PARENTHESIS @push_delimiter @push_jump
@goto_f @push_scope ⟨*block*⟩ @pop_scope @push_jump

@goto @link_jump_down_n1 ⟨*condition_1*⟩ @pop_all_-
jumps;

⟨*condition_1*⟩ ::= /* ⟨*empty*⟩ */
| ELIF OPEN_PARENTHESIS ⟨*expression*⟩
CLOSE_PARENTHESIS @push_jump @goto_f
@push_scope ⟨*block*⟩ @pop_scope @push_jump
@goto @link_jump_down_n1 ⟨*condition_1*⟩
| ELSE @push_scope ⟨*block*⟩ @pop_scope;

⟨*for_loop*⟩ ::= FOR OPEN_PARENTHESIS @push_scope
⟨*for_loop_1*⟩ @push_jump ⟨*for_loop_2*⟩
@push_jump @goto_f
@push_jump @goto @push_jump ⟨*for_loop_3*⟩
CLOSE_PARENTHESIS @goto @link_jump_up_n3
@link_jump_down_n1 ⟨*block*⟩ @goto
@link_jump_up @link_jump_down @pop_scope;

⟨*for_loop_1*⟩ ::= SEMICOLON
| ⟨*assign*⟩ SEMICOLON;

⟨*for_loop_2*⟩ ::= SEMICOLON
| ⟨*expression*⟩ SEMICOLON;

⟨*for_loop_3*⟩ ::= /* EMPTY */
| ⟨*assign*⟩;

⟨*while_loop*⟩ ::= WHILE @push_jump OPEN_PARENTHESIS
⟨*expression*⟩ CLOSE_PARENTHESIS @push_jump
@goto_f @push_scope ⟨*block*⟩ @pop_scope @goto
@link_jump_down @link_jump_up;

⟨*function_call*⟩ ::= ID ⟨*function_call_1*⟩ OPEN_PARENTHESIS
CLOSE_PARENTHESIS
| ID ⟨*function_call_1*⟩ OPEN_PARENTHESIS
⟨*expression*⟩ ⟨*function_call_2*⟩ CLOSE_PARENTHESIS
;

⟨*function_call_1*⟩ ::= /* ⟨*empty*⟩ */
| DOT ID;

⟨*function_call_2*⟩ ::= /* ⟨*empty*⟩ */
| COMMA ⟨*expression*⟩ ⟨*function_call_2*⟩;

⟨*statement*⟩ ::= ⟨*vars*⟩
| ⟨*assign*⟩ SEMICOLON
| ⟨*condition*⟩
| ⟨*while_loop*⟩
| ⟨*for_loop*⟩
| ⟨*function_call*⟩ SEMICOLON
| RETURN ⟨*expression*⟩ SEMICOLON
| RETURN SEMICOLON ;

⟨*expression*⟩ ::= ⟨*expression_l1*⟩
| ⟨*expression*⟩ ⟨*boolean_op_l1*⟩ ⟨*expression_l1*⟩ ;

⟨*expression_l1*⟩ ::= ⟨*expression_l2*⟩
| ⟨*expression_l1*⟩ ⟨*boolean_op_l2*⟩ ⟨*expression_l2*⟩ ;

⟨*expression_l2*⟩ ::= ⟨*expression_l3*⟩
| ⟨*expression_l2*⟩ ⟨*bitwise_op_l1*⟩ ⟨*expression_l3*⟩ ;

⟨*expression_l3*⟩ ::= ⟨*expression_l4*⟩
| ⟨*expression_l3*⟩ ⟨*bitwise_op_l2*⟩ ⟨*expression_l4*⟩ ;

⟨*expression_l4*⟩ ::= ⟨*expression_l5*⟩
| ⟨*expression_l4*⟩ ⟨*bitwise_op_l3*⟩ ⟨*expression_l5*⟩ ;

⟨*expression_l5*⟩ ::= ⟨*expression_l6*⟩
| ⟨*expression_l5*⟩ ⟨*relational_op_l1*⟩ ⟨*expression_l6*⟩ ;

⟨*expression_l6*⟩ ::= ⟨*expression_l7*⟩
| ⟨*expression_l6*⟩ ⟨*relational_op_l2*⟩ ⟨*expression_l7*⟩ ;

⟨*expression_l7*⟩ ::= ⟨*expression_l8*⟩
| ⟨*expression_l7*⟩ ⟨*bitwise_op_l4*⟩ ⟨*expression_l8*⟩ ;

⟨*expression_l8*⟩ ::= ⟨*expression_l9*⟩
| ⟨*expression_l8*⟩ ⟨*arithmetic_op_l1*⟩ ⟨*expression_l9*⟩ ;

⟨*expression_l9*⟩ ::= ⟨*expression_l10*⟩
| ⟨*expression_l9*⟩ ⟨*arithmetic_op_l2*⟩ ⟨*expression_l10*⟩ ;

⟨*expression_l10*⟩ ::= ⟨*expression_l11*⟩
| ⟨*boolean_op_l3*⟩ ⟨*expression_l11*⟩
| ⟨*bitwise_op_l5*⟩ ⟨*expression_l11*⟩ ;

⟨*expression_l11*⟩ ::= OPEN_PARENTHESIS ⟨*expression*⟩
CLOSE_PARENTHESIS
| ⟨*function_call*⟩
| ⟨*const_type*⟩
| ⟨*variable*⟩;

3.4 Semantic analysis and intermediate code

3.4.1 Syntax diagrams and neuralgic points

Each of the following definitions introduce all of J++'s syntax diagrams along with its corresponding neuralgic point actions.

Expressions

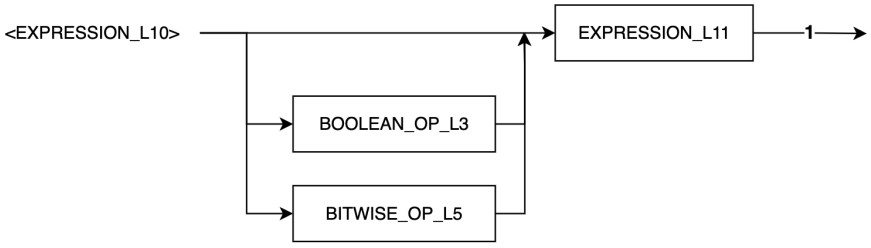


Figure 3.1: Expression L10 Syntax Diagram

Push BOOLEAN_OP_L3 or BITWISE_OP_L5 operator to the operand stack

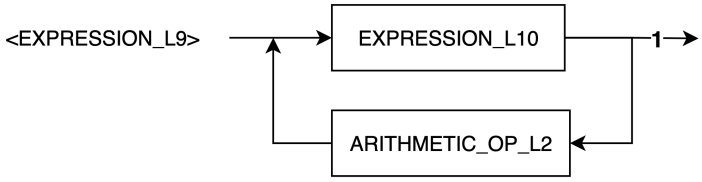


Figure 3.2: Expression L9 Syntax Diagram

Push ARITHMETIC_OP_L2 operator to the operand stack

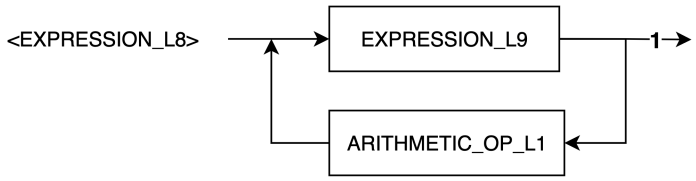


Figure 3.3: Expression L8 Syntax Diagram

Push ARITHMETIC_OP_L1 operator to the operand stack

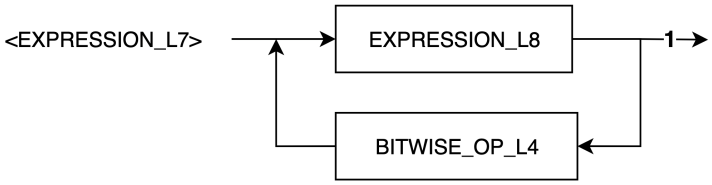


Figure 3.4: Expression L7 Syntax Diagram

Push BITWISE_OP_L4 operator to the operand stack

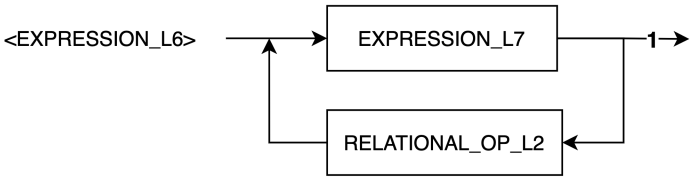


Figure 3.5: Expression L6 Syntax Diagram

Push RELATIONAL_OP_L2 operator to the operand stack

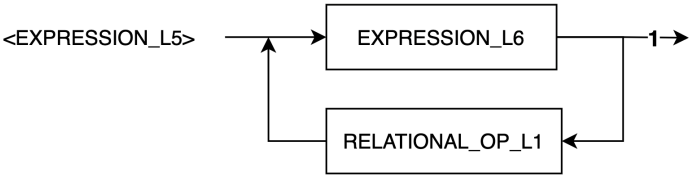


Figure 3.6: Expression L5 Syntax Diagram

Push RELATIONAL_OP_L1 operator to the operand stack

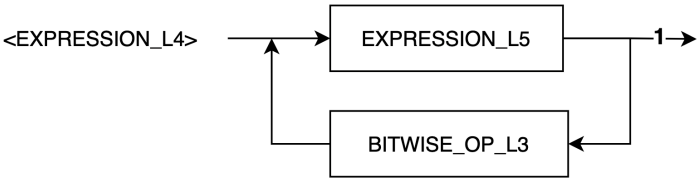


Figure 3.7: Expression L4 Syntax Diagram

Push BITWISE_OP_L3 operator to the operand stack

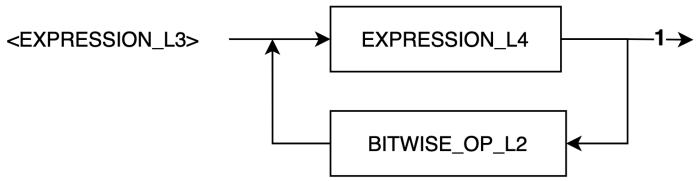


Figure 3.8: Expression L3 Syntax Diagram

Push BITWISE_OP_L2 operator to the operand stack

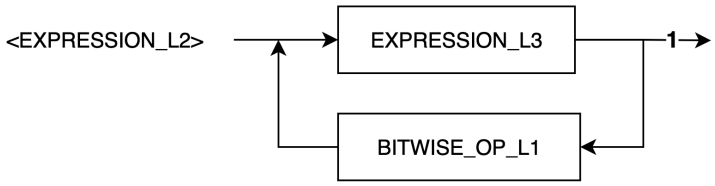


Figure 3.9: Expression L2 Syntax Diagram

Push BITWISE_OP_L1 operator to the operand stack

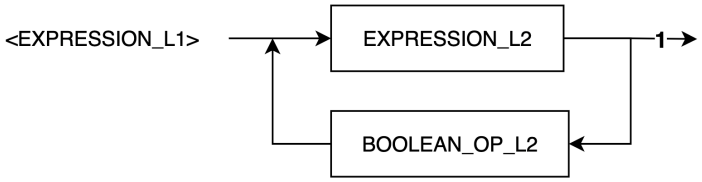


Figure 3.10: Expression L1 Syntax Diagram

Push BOOLEAN_OP_L2 operator to the operand stack

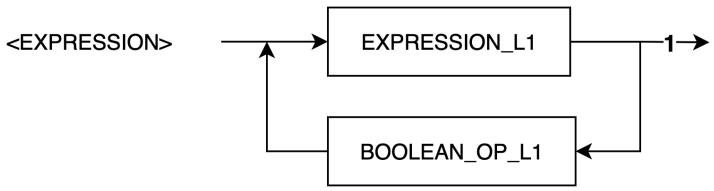


Figure 3.11: Expression Base Syntax Diagram

Push BOOLEAN_OP_L1 operator to the operand stack

Types

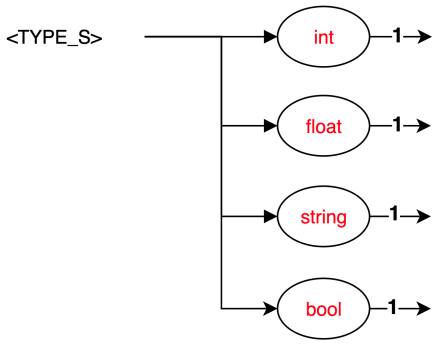


Figure 3.12: Static Types Syntax Diagram

Set global current type

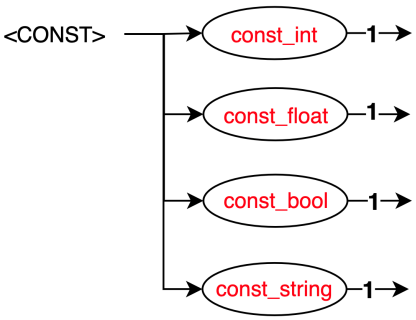


Figure 3.13: Constant types Syntax Diagram

Push constant operant to operand stack

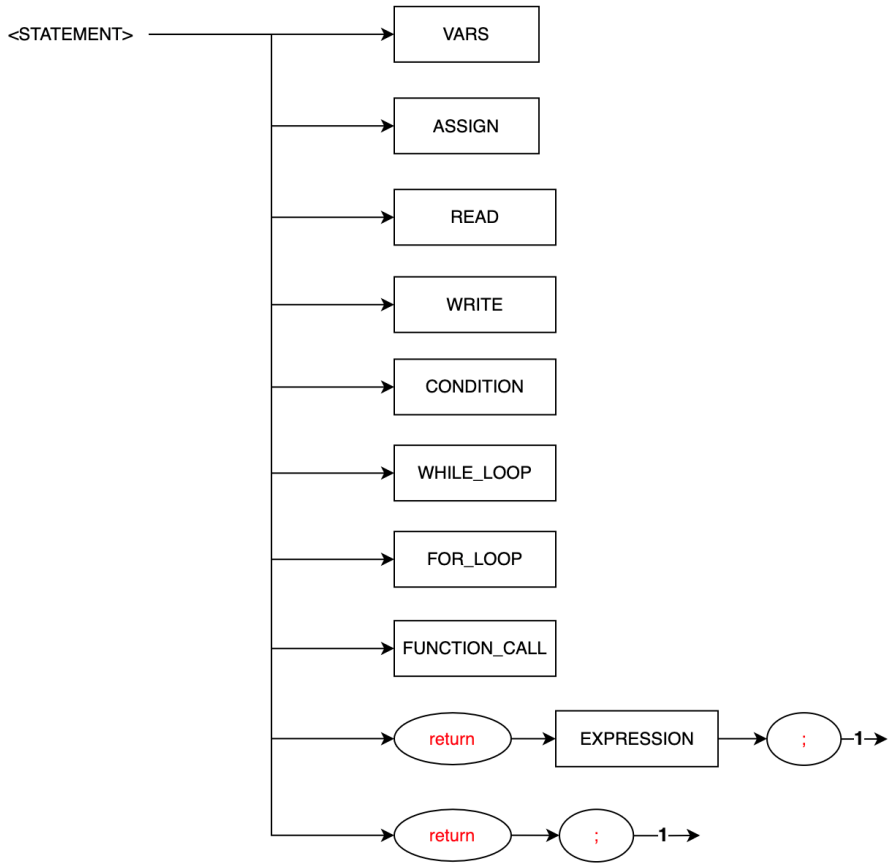


Figure 3.14: Statement Syntax Diagram

Insert return statement and verify current scope return type — if any — matches the resulting expression type

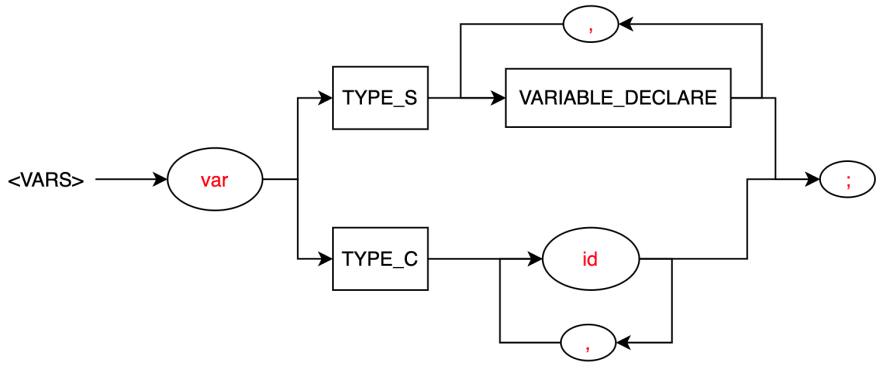


Figure 3.15: Vars Syntax Diagram

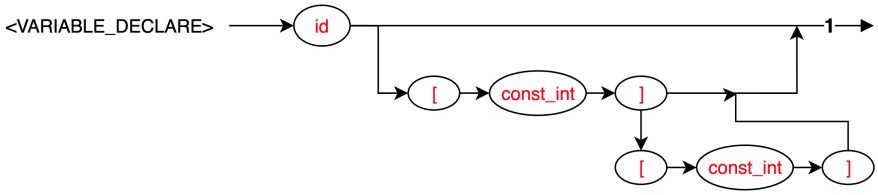


Figure 3.16: Variable Declaration Syntax Diagram

1. Register variable to current scope
   (a) Verify variable name has not been used in the current scope
   (b) Determine memory type to assign based on global current type and scope type
   (c) If variable is multi-dimensional, assign the necessary amount of addresses

Figure 3.17: Assign Syntax Diagram

1. Pop two operands from stack
   (a) Push assign quadruple with ASTORE (address store) if ADDRESS_-REFERENCE flag is on; else push assign quadruple with STORE

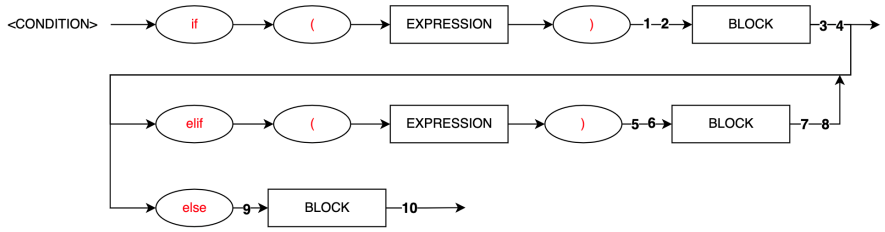


Figure 3.18: Condition Syntax Diagram

1. Push fake bottom (-1) on jumps stack
   (a) Push in jumps stack current quadruples.size()
   (b) Push GOTO_F into quadruples list
2. Push scope into scopes tree
3. Pop current scope
4. Push in jumps stack current quadruples.size()
   (a) Push GOTO into quadruples list

(b) Link second-last jump quadruple with current quadruples.size()

5. Push in jumps stack current quadruples.size()
   (a) Push GOTO_F into quadruples list
6. Push scope into scopes tree
7. Pop current scope
8. Push in jumps stack current quadruples.size()
   (a) Push GOTO into quadruples list
   (b) Link second-last jump quadruple with current quadruples.size()
9. Push scope into scopes tree
10. Pop current scope

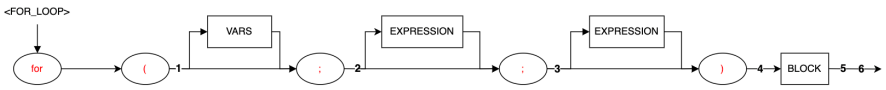


Figure 3.19: For Loop Syntax Diagram

1. Push scope into scope tree
2. Push in jumps stack current quadruples.size()
3. Push in jumps stack current quadruples.size()
   (a) Push GOTO_F into quadruples list
   (b) Push in jumps stack current quadruples.size()
   (c) Push GOTO into quadruples list
   (d) Push in jumps stack current quadruples.size()
4. Push GOTO into quadruples list; pop third-last jump from the jumps stack and set it to the just inserted GOTO
   (a) Link jump.top() quadruple with current quadruples.size() and jump.pop()
5. Push GOTO into quadruples list; pop last jump from the jumps stack and set it to the just inserted GOTO
   (a) Link jump.top() quadruple with current quadruples.size() and jump.pop()
6. Pop current scope

Figure 3.20: For Loop Syntax Diagram

1. Push in jumps stack current quadruples.size()
2. Push in jumps stack current quadruples.size()
   (a) Push GOTO_F into quadruples list
3. Push scope into scope tree
4. Pop current scope
5. Push GOTO into quadruples list
   (a) Link jump.top() quadruple with current quadruples.size() and jump.pop()
6. Link previous GOTO quadruple with jump.top() and jump.pop()

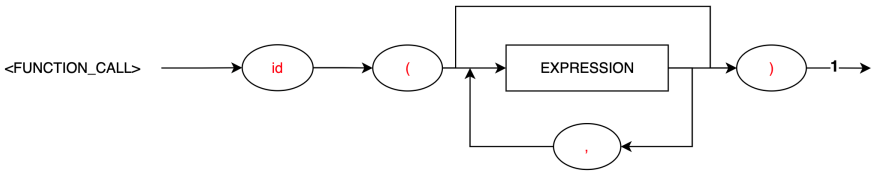


Figure 3.21: Function Call Syntax Diagram

1. Find function data payload in the scope tree
   (a) Push AIR quadruple
   (b) Iterate over function argument addresses and push PARAM quadruples with addresses to resolve from the operand stack
   (c) Push CALL quadruple along with the function start index
   (d) If function type is not VOID, push STORE quadruple to assign result from return statement to the function variable

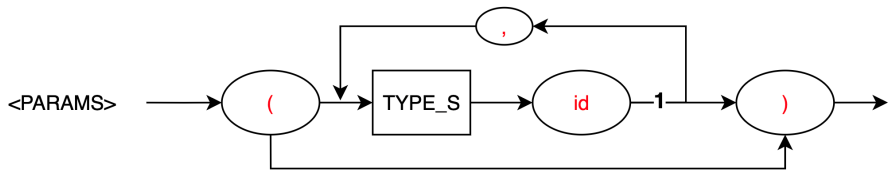


Figure 3.22: Params Syntax Diagram

1. Register variable alias in current scope (the scope expected to be one of a function)
   (a) Register variable details (address, alias, etc.) into the current function declaration

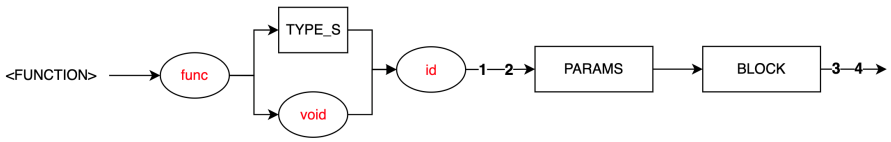


Figure 3.23: Function Syntax Diagram

1. Register function in current scope and set it as current function declaration
2. Push scope into scope tree
3. Remove function from current function declaration
   (a) If function type is VOID, insert default RETURN quadruple with no value
   (b) Reset local memory addresses
4. Pop current scope

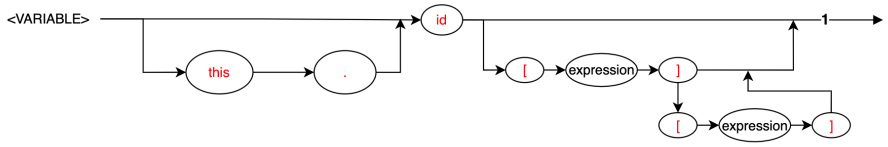


Figure 3.24: Variable Syntax Diagram

1. Verify and fetch variable id from the nearest scope possible
2. If variable is multi-dimensional, insert multiplication and addition quadruples based on expressions from the operand stack to compute resulting address in runtime
3. Push variable address to operand stack
4. Push address reference placeholder to store result

3.4.2 Types and their relations

J++'s types and operators relations are defined by its Type-Type-Operator cube where left and right operands are interchangeable. Anything **not** in the cube, yields a type mismatch error. The following table displays all of the possible type combinations and their type cast result:

Table 3.2: TTO Table

Left Type	Right Type	Operator	Result Type

Continued on next page

Table 3.2 – continued from previous page

Left Type	Right Type	Operator	Result Type
INT	INT	PLUS	INT
INT	INT	MINUS	INT
INT	INT	MULTIPLICATION	INT
INT	INT	DIVISION	INT
INT	INT	MODULO	INT
INT	INT	BITWISE_OR	INT
INT	INT	BITWISE_XOR	INT
INT	INT	BITWISE_AND	INT
INT	INT	BITWISE_LEFT_SHIFT	INT
INT	INT	BITWISE_RIGHT_SHIFT	INT
INT	INT	BOOLEAN_OR	INT
INT	INT	BOOLEAN_AND	INT
INT	INT	EQUALS	INT
INT	INT	NOT_EQUALS	INT
INT	INT	GT	INT
INT	INT	GTE	INT
INT	INT	LT	INT
INT	INT	LTE	INT
INT	INT	ASSIGN	INT
	INT	BITWISE_NOT	INT
	INT	BOOLEAN_NOT	INT
INT	FLOAT	PLUS	FLOAT
INT	FLOAT	MINUS	FLOAT
INT	FLOAT	MULTIPLICATION	FLOAT
INT	FLOAT	DIVISION	FLOAT
INT	FLOAT	MODULO	FLOAT
INT	FLOAT	BOOLEAN_OR	INT
INT	FLOAT	BOOLEAN_AND	INT
INT	FLOAT	EQUALS	INT
INT	FLOAT	NOT_EQUALS	INT
INT	FLOAT	GT	INT
INT	FLOAT	GTE	INT
INT	FLOAT	LT	INT
INT	FLOAT	LTE	INT
STRING	STRING	PLUS	STRING
STRING	STRING	EQUALS	STRING
STRING	STRING	NOT_EQUALS	STRING
STRING	STRING	ASSIGN	STRING
STRING	INT	PLUS	STRING
STRING	FLOAT	PLUS	STRING

3.5 Memory management

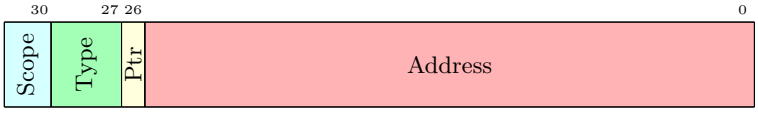
3.5.1 Memory Addresses

Internally *JavaScript* uses the double-precision floating-point format (also known as IEEE 754 64-bit) for storing numbers, it lacks the different number types such as integer, double, byte, short. Additionally, while the numbers are 64-bit numbers, all bit shifts transform the number into a 32-bit number before applying the shift. This is described in the ECMA language specification section 6.1.6.1.9 to 6.1.6.1.11.

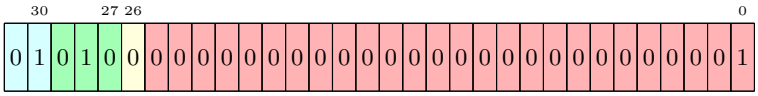
Since *J++*'s virtual machine will run on top of *JavaScript*, *J++* will use 32-bit numbers to represent memory addresses. Each address has 4 basic components:

1. **Memory Scope** - (Bits 31 and 32) The scope of the memory address
   - 00 - Global
   - 01 - Local
   - 10 - Temp
   - 11 - *(Unused)*
2. **Memory Type** - (Bits 28 to 30) The type of data that the address is going to store
   - 000 - INT
   - 001 - FLOAT
   - 010 - STRING
3. **Pointer Flag** - (Bit 27) A flag that will indicate if the value in the address is a pointer to another address
   - 0 - Not a Pointer
   - 1 - It is a pointer
4. **Address** - (Bit 1-26) The numerical address

This leaves 2^{26} usable addresses, which is $67,108,864$ total usable addresses. The following image visually shows how the address is partitioned



An example address can be $1,342,177,280$ ($Base_{10}$), which in binary would be equal to 01010000000000000000000000000001



The scope is **01**, indicating it's a local variable, memory type is **010**, which means it's a String and the assigned address is **1**.

3.5.2 Scope tree

J++ scoping mechanics follow a traditional scoping tree structure represented in a flattened array linked through indices. All elements declared by the user (variables, multi-dimensional variables, and functions) are all considered variables with an assigned type and varying properties and capabilities. Internally, J++ sees a function as a **callable** variable with its return type as its type.

As a rule of thumb, every time a pair of brackets is introduced into a program, a new scope is established. When a variable is referenced in an expression, or passed as an argument, a bottom-up scope lookup algorithm traverses the tree to find the nearest match.

For example:

```
import("libs/io.jpp")

func int add(int a, int b) {
    return a + b;
}

program ScopeTreeExample {
    var int d, e;
    d = 1;
    e = 2;
    write(add(d, e));
}
```

The previous code snippet would yield the following scoping tree; each scope containing a list of all visible variables and a reference to its **parent** scope up the tree.

{add:int, parent:-1}	**{b:int, a:int, parent:0}**	**{d:int, e:int, parent:0}**

4 Virtual Machine Description

4.1 Run-time memory management

J++ compile-time memory management is sufficiently succinct and abstract that the exact same data structure and memory segmentation strategy suffices during run-time. All intermediate code assigned addresses are directly utilized to set values while the virtual machine is traversing the generated quadruples.

Alongside the memory manager, the virtual machine relies on a "call stack" to keep track of subroutine jumps that change the instruction pointer. In a straightforward manner, the virtual machine's job is to focus on safely traversing the instruction pointer over the generated intermediate code (a.k.a quadruples) and switch between memory instances when a subroutine occurs.

4.1.1 Transition between memory instances

A very important and concrete feature any virtual machine with subroutine support requires is a transition strategy between memory instances to "freeze" the current state of the memory and replace it with a new memory instance usable by the subroutine.

```
era() {
    this.previousMemory = Object.assign(
      ...this.eraTypes.map((type) => {
        return { [type]: this.segments[this.scopeLookup[type]] };
      })
    );
    this.memoryStack.push(this.previousMemory);
    this.eraTypes.forEach((type) => {
      const scopeIndex = this.scopeLookup[type];
      const scopeBits = scopeIndex << 3;
      this.segments[scopeIndex] = this.dataTypes.map((type,
    typeIndex) => {
        return new Memory((scopeBits | typeIndex) << 27, type);
      });
    });
```

```
}
```

The **era()** method makes a copy of the current memory state (all segments with assigned addresses) and pushes it to a memory stack prior to replacing all memory segments with new instances. Allowing J++ to recover its previous memory state when the subroutine finishes.

5 Language Examples

5.1 Basic Examples

5.1.1 Hello world

The tribute to the history of programming languages.

```
import("libs/io.jpp")

program HelloWorld {
    write("Hello World!");
}
```

Console output

```
Hello World!
```

Intermediate code

(index)	0	1	2	3
0	'GOTO'	null	null	1
1	'INIT'	null	null	null
2	'LOAD'	'Hello World!'	null	'TEMP.STRING.0'
3	'NPARAM'	'TEMP.STRING.0'	null	'LOCAL.STRING.0'
4	'NCALL'	'write'	null	null
5	'EXIT'	null	null	null

5.1.2 Type casting

```
import("libs/io.jpp", "libs/string.jpp")

program TypeCasting {
    var int a;
    var float b;
    a = str_to_int(read("Value for a: "));
    b = str_to_float(read("Value for b: "));
    write(a);
    write(b);
}
```

Console output

```
Value for a: // 10.5
Value for b: // 10
10
10
```

5.1.3 For-loop

```
import("libs/io.jpp", "libs/string.jpp")

program ForLoop {
    var int i, j;
    i = str_to_int(read("Iterate from 0 to: "));
    for (j = 0; j < i; j = j + 1) {
        write(j);
    }
}
```

Console output

```
Iterate from 0 to: // 5
0
1
2
3
4
```

5.1.4 Binary search

```
import("libs/io.jpp", "libs/string.jpp")

program BinarySearch {
    var int i, k, left, right, middle, arr[10];
    var bool found;

    i = 0;
    while (i < 10) {
        arr[i] = str_to_int(read("Value: "));
        i = i + 1;
    }

    k = str_to_int(read("Search for value: "));
    left = 0;
    right = 10;
    found = false;
    while (left < right && !found) {
        middle = (left + right) / 2;
        write(middle);
        if (arr[middle] == k) {
            write(k + " at index " + middle);
            found = true;
        } elif (arr[middle] < k) {
            left = middle + 1;
        } else {
            right = middle - 1;
        }
    }
    if (!found) {
        write(k + " not in array");
    }
}
```

Console output

```
Value: // 1
Value: // 3
Value: // 5
Value: // 7
Value: // 9
Value: // 10
Value: // 14
Value: // 17
Value: // 20
Value: // 24
```

```
Search for value: // 24
5
8
9
24 at index 9
```

5.2 Advanced Examples

5.2.1 Fibonacci

```
import("libs/io.jpp")

func int fib(int n) {
    if (n <= 1) {
        return n;
    }

    return fib(n - 1) + fib(n - 2);
}

program Fibonacci {
    var int i;

    for (i = 1; i < 25; i = i + 1) {
        write("fib " + i + ": " + fib(i));
    }
}
```

```
Running .\tests\files\test11.jpp...
fib 1: 1
fib 2: 1
fib 3: 2
fib 4: 3
fib 5: 5
fib 6: 8
fib 7: 13
fib 8: 21
fib 9: 34
fib 10: 55
fib 11: 89
fib 12: 144
fib 13: 233
fib 14: 377
fib 15: 610
fib 16: 987
fib 17: 1597
fib 18: 2584
fib 19: 4181
fib 20: 6765
fib 21: 10946
fib 22: 17711
fib 23: 28657
fib 24: 46368
Program finished.
```

```
Done in 0.87s.
```

5.2.2 Fibonacci Optimized

```
import("libs/io.jpp")

var int cache[50];

func int fib(int n) {
    if (n <= 1) {
        return n;
    }

    if (cache[n] > 0) {
        return cache[n];
    }

    var int res;
    res = fib(n - 1) + fib(n - 2);
    cache[n] = res;
    return res;
}

program FibonacciOptimized {
    var int i;

    for (i = 0; i < 50; i = i + 1) {
        cache[i] = 0;
    }

    for (i = 1; i < 50; i = i + 1) {
        write("fib " + i + ": " + fib(i));
    }
}
```

```
Running .\tests\files\test12.jpp...
fib 1: 1
fib 2: 1
fib 3: 2
fib 4: 3
fib 5: 5
fib 6: 8
fib 7: 13
fib 8: 21
fib 9: 34
fib 10: 55
fib 11: 89
fib 12: 144
fib 13: 233
fib 14: 377
```

```
fib 15: 610
fib 16: 987
fib 17: 1597
fib 18: 2584
fib 19: 4181
fib 20: 6765
fib 21: 10946
fib 22: 17711
fib 23: 28657
fib 24: 46368
fib 25: 75025
fib 26: 121393
fib 27: 196418
fib 28: 317811
fib 29: 514229
fib 30: 832040
fib 31: 1346269
fib 32: 2178309
fib 33: 3524578
fib 34: 5702887
fib 35: 9227465
fib 36: 14930352
fib 37: 24157817
fib 38: 39088169
fib 39: 63245986
fib 40: 102334155
fib 41: 165580141
fib 42: 267914296
fib 43: 433494437
fib 44: 701408733
fib 45: 1134903170
fib 46: 1836311903
fib 47: 2971215073
fib 48: 4807526976
fib 49: 7778742049
Program finished.
Done in 0.43s.
```

5.2.3 Spinning Donut

```
import(
    "libs/io.jpp",
    "libs/string.jpp",
    "libs/math.jpp"
)

var string chars[12];

func void init_chars() {
    chars[0] = ".";
    chars[1] = ",";
    chars[2] = "-";
    chars[3] = "~";
    chars[4] = ":";
    chars[5] = ";";
    chars[6] = "=";
    chars[7] = "!";
    chars[8] = "*";
    chars[9] = "#";
    chars[10] = "$";
    chars[11] = "@";
}

program Torus {
    init_chars();
    var int k;
    var float A;
    A = 0;
    var float B;
    B = 0;
    var float i;
    var float j;
    var float z[1760];
    var string b[1760];

    clear_console();

    while (true) {
        for (i = 0; i < 1760; i = i + 1) {
            b[i] = " ";
            z[i] = 0;
        }

        for (j = 0; 6.28 > j; j = j + 0.07) {
            for (i = 0; 6.28 > i; i = i + 0.02) {
                var float c, d, e, f, g, h, D, l, m, n, t;
                c = sin(i);
                d = cos(j);
```

```
                e = sin(A);
                f = sin(j);
                g = cos(A);
                h = d + 2;
                D = 1 / (c * h * e + f * g + 5);
                l = cos(i);
                m = cos(B);
                n = sin(B);
                t = c * h * g - f * e;

                var int x, y, o, N;
                x = f2i(40 + 30 * D * (l * h * m - t * n));
                y = f2i(12 + 15 * D * (l * h * n + t * m));
                o = f2i(x + 80 * y);
                N = f2i(8 * ((f * e - c * d * g) * m - c * d * e
    - f * g - l * d * n));

                if (22 > y && y > 0 && x > 0 && 80 > x && D > z[o
    ]) {
                    z[o] = D;
                    if (N > 0) {
                        b[o] = chars[N];
                    } else {
                        b[o] = chars[0];
                    }
                }
            }
        }

        cursor_home();

        for (k = 0; 1761 > k; k = k + 1) {
            if (k % 80) {
                putchar(b[k]);
            } else {
                write();
            }
        }

        A = A + 0.04;
        B = B + 0.02;
    }
}
```