Universidad del Valle

UNIVERSIDAD DEL VALLE
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
FUNDAMENTOS DE LENGUAJE DE PROGRAMACIÓN

PROYECTO FINAL DEL CURSO

JEFFERSON AMADO PEÑA TORRES CÓDIGO 1425590
JHON JAVIER CARDONA M. CÓDIGO 0747558
WILMAR RIASCOS MONTAÑO CÓDIGO 1410104

1. Decisiones tomadas

1.1. Se utilizó el generador automático **sllgen** para generación de los tipos de dato sobre los cuales consideramos que conformaría la parte estructural del lenguaje, en consecuencia, dichos tipos fueron definidos como categorías de la gramática. Otros tipos de datos que asumimos como adicionales al lenguaje y por tanto no estaría bien definirlos como parte de su estructura básica, fueron definidos manualmente mediante **define data-type.**

1.2. En planteamiento inicial del lenguaje optamos por la utilización de la categoría gramatical **expression** para representar toda instrucción que se necesitará evaluar en el cuerpo del método principal. Una decisión importante fue la redefinición de esta idea inicial, separando las instrucciones que tras su ejecución retornaran algún dato, de aquellas que simplemente realizarían modificaciones sin retornar dato alguno, a partir de esta idea incluimos en la gramática la categoría **java-expression**, con dos variantes, una que representa las instrucciones con retorno (**java-single-exp**) y otra para aquellas instrucciones que solo realizan modificaciones (**java-single-stat**).

mas…

2. Descripción de la gramática y las funciones que evalúan cada categoría gramatical.

La gramática del lenguaje está representada por 11 categorías principales con sus respectivas variantes (para los casos en que aplicaba), a continuación describiremos dichas categorías indicando en cada caso el número de variantes utilizado para la misma, como también la función definida para su evaluación:

2.1. program

La categoria program, cuya unica variante es **a-program**, representa el cuerpo general de un programa desarrollado en **R-Java**.

Para la evaluación de esta categoría se definió la función eval-program, cuyo argumento de entrada es un programa en **R-Java**, y su finalidad es evaluar dicho programa, esto lo hace a partir del uso de las funciones **elaborate-class-decls!, eval-main**, cada una de las cuales será descrita en la categoría correspondiente.

2.2. java-expression

Esta categoría es usada para representar las instrucciones en el cuerpo del método principal (**main-statement**) de un programa en R-Java, consta de las 2 variantes mencionadas en el numeral (1.2) de este documento, para esta categoria no se realizo ninguna función evaluadora, ya que cada vez que se requiere evaluar, se verifica la variante a la que corresponde y se usa la o la funcion **eval-statement** o la funcion **eval-expression**, segun sea el caso.

2.3. procedure

Con esta categoría representamos los procedimientos del lenguaje, tiene una única variante **proc-decl,** y su evaluación que recibe como argumentos el cuerpo

2.4. class

2.5. method

2.6. main-statement

La categoría **main-statement** también tiene una sola variante, para la evaluación de esta categoría implementamos la función **eval-main**, que toma como parámetros el cuerpo de un programa en R-java y un ambiente en el cual debe ser ejecutado, dado que el cuerpo una secuencia de **java-expression**, se chequea cada una de estas a fin de saber si aplicamos la funcion **eval-statement** o la funcion **eval-expression.**

2.7. expression

Esta categoría consta de 23 variantes, y su función evaluadora es **eval-expression,** que recibe una **expression** y un ambiente donde esta debe ser evaluada. Los primero 5 casos de esta función se encargan de retornar los datos básicos del lenguaje **number, string, char, t(true) y (false)**, seguidamente evalúan las variantes para **variable** e **identifier** utilizando para ello la función **apply-env**, que la cual a su vez usa una función auxiliar **apply-env-ref**, para extraer los valores de identificadores y variables respectivamente.

Las siguientes tres variantes **create-array, access-array-exp**, se encargan de la creación y acceso al contenido de un array respectivamente.

A continuación aparecen las variantes para o para acceder y adicionar un elemento e insertar una posición específica a un vector **access-vector-exp, addelement-vector-exp, insert-vector-exp**, respectivamente.

2.9. patron

2.10. statement

2.11. patron

2.11. operator

2.12. primitive

3. Definición de las estructuras y tipos de datos

Se definieron

4. Código fuente de la aplicación

4.1. La definición BNF para las expresiones del lenguaje:

```
;; <program>   ::={<procedure>}*m
;;               {<class>}*
;;               <main-statement> "{" <java-expression> "}"
;;               <a-program (procs clases main)>
;;
;; <java-expression>  ::=<expression> ";"
;;               <java-single-exp(exp)>
;;               ::=<statement> ";"
;;               <java-single-stat(stat)>
;;
;; <procedure>        ::= "proc""(" identifier {<variables>}* ")"
;;               "{" {<java-expression>}* "}"
;;               <proc-exp (id vars body)>
;;
;; <class>            ::= "class" identifier "extends" <class>
;;               "{"
;;                  {{<variable>}*(,) ";"}*
;;                  {<method>}*
;;                  "}"
;;                  <class-exp (id attribs meths)>
;;
;; <method>               ::= "method" identifier "(" {<variables>}*(,) ")"
;;                  "{" {<java-expression>}* "}"
;;                  <meth-exp (id vars body)>
;;
;; <main-statement>   ::= "main()" "{" {<java-expression>}* "}"
;;                  <a-main(body)>
;;
;; <expression>          ::= <number>
;;                  <lit-exp(datum)>
;;                  ::= <variable>
;;                  <var-exp (var)>
;;                  ::= <identifier>
;;                  <id-exp (id)>
;;                  ::= <string>
;;                  <string-exp (str)>
;;                  ::= <character>
;;                  <char-exp (chart)>
;;                  ::= "true"
;;                  <true-exp>
;;
;;                  <false-exp>
;;                  ::= " ("<primitive> {<expression>}* ")"
;;                  <app-exp(prim args)>
;; <expression>          ::= array <variable>"=#""{" {<expression>}*(,) "}"
;;                  <create-array (var exps)>
;; <expression>          ::= [ <variable> <expression> ]
;;                  <access-array-exp (var exp)>
;; <expression>          ::= "make-vector""(" <expression> ")"
;;                  <create-vector-exp (exp)>
;;                  ::= "elementAt""(" <variable> <expression> ")"
;;                  <access-vector-exp (var exp)>
;;                  ::= "addElement""("<variable> <expression>")"
;;                  <addelement-vector-exp (var exp)>
;;                  ::= "insertElementAt""("<variable> <expression> <expression>")"
;;                  <insert-vector-exp (var exp1 exp2)>
;; <expression>          ::=  ++ <variable>
;;                  <increment-exp (var)>
;;                  ::= -- >variable>
;;                  <decrement-exp (var)>
;; <expression>         ::= "struct" <expression> "{" {<variable>}* "}"
;;                  <structure-declaration (id-struct vars)>
;;                  ::= "record" <expression> "{" {<expression>}*(,) "}"
;;                  <struct-specifier (id_struct rands)>
;;                  ::= "struct." <expression> "." <expression>
;;                  <struct-selection (id_struct rand)>
;;                  ::= "instanceOf" "(" <expression> <expression>")"
;;                  <struct-instanceof (id_struct id)>
;;                  ::= "new" identifier ({<expression>}*)
;;                  <instance-class (exp)>
;;                  ::= "class."<variable>"."<expression>
;;                  <selection-class (var exp)>
;;
;; <patron>             ::= <expression> " : "
;;                  <patron-exp(exp)>
;;
;; <operator>           ::=  /= <div-assign-oper>
;;                  ::=  += <add-assign-oper>
;;                  ::=  -= <substract-assign-oper>
;;                  ::=  *= <mult-assign-oper>
;;                  ::=  =  <assign-oper>
;;
;; <statement>          ::= set <variable> <operator> <expression>
;;                  <set-exp (var oper rhsexp)>
;;                  ::= "switch""("<expression>")"
;;                  "{" {case <expression> ":" <java-expression> "break;"}*
;;                  "default:" <java-expression>}
;;                  <switch-case-exp (exp exps jexps jexp)>
;;                  ::= "for""("<variable> " = " <expression>";" <expression>";" <expression>")"
;;                  "{"<java-expression>"}"
;;                  <for-exp (id initexp exp2 exp3 body)>
;;                  ::= "while""(" <expression> ")"
;;                  "{" <java-expression> "}"
;;                  <while-exp (exp body)>
;;                  ::= "do""{" {<java-expression>}* "}""while"(" <expression> ")"
;;                  <do-while-exp (body exp)>
;;                  ::= "if""(" <expression> ")""{" {<java-expression>}* "}"
;;                  {"elseif" "("<expression>")""{"{<java-expression>}*"}"}*
;;                  "else""{"{<java-expression>}*"}"
;;                  <if-exp (exp jexps1 exps jexps jexps2)>
;;                  ::= "cases" <expresion> "of" { <patron> "begin" {<java-expression>}* "end"}* "}"
OJO  esto estaba como <expression>???
```

```
;;                  <cases-exp(exp1 ptrs jexps)>
;;                  ::= local {<variable>}* in {<java-expression>}* "," <java-expression> OJO  esto
estaba como <expression>???
;;                  <let-exp (vars body jexp)>
;;
;; <primitive>               ::= newarray
;;                  <array-prim>
;;                  ::= setarray
;;                  <setarray-prim>
;;                  ::= length
;;                  <length-prim>
;; <primitive>               ::= + <add-prim>
;;                  ::= - <substract-prim>
;;                  ::= * <mult-prim>
;;                  ::= / <div-prim>
;;                  ::= % <modulo-prim>
;;                  ::=inver <reverse-prim>
;;                  ::= pow <pow-prim>
;;                  ::= == <equalto-prim>
;;                  ::= max <max-prim>
;;                  ::= min <min-prim>
;;                  ::= > <lessthan-prim>
;;                  ::= < <greaterthan-prim>
;;                  ::= <= <lessthanorequalto-prim>
;;                  ::= >= <greaterthanorequalto-prim>
;;                  ::= append <apend-prim>
;;                           ::=<identifier> <procedure-prim>
;; <primitive>               ::= and <and-prim>
;;                  ::=  or <or-prim>
;;                  ::=  not <not-prim>
;; <primitive>               ::= cons <cons-prim>
;;                  ::=  car <car-prim>
;;                  ::=  cdr <cdr-prim>
;;                  ::=  null? <null-prim>
;;                  ::=  list <list-prim>
;;                  ::=  length <length-prim>
```

4.2. Especificación Lexica

```
(define scanner-spec
  '((whitespace (whitespace) skip)
            (comment ("//" (arbno (not #\newline))) skip)
            (comment ("/*" (arbno (or letter digit "_" "-" "?")) "*/" ) skip)
            (variable ((or "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "ñ" "o" "p" "q" "r" "s" "t"
"u" "v" "w" "x" "y" "z")
            (arbno (or letter digit "_" "-" "?")))symbol)
            (identifier ((or "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "Ñ" "O" "P" "Q"
"R" "S" "T" "U" "V" "W" "X" "Y" "Z")
                  (arbno (or letter digit "_" "-" "?"))) symbol)
            (number (digit (arbno digit)) number)
            (number ("-" digit (arbno digit)) number)
            (string ( "\"" ( arbno (or digit letter "_" "-")) "\"" ) string)
            (character ("'" (or digit letter) "'") string))
      )
```

4.3. Gramática

```
(define the-grammar
 '((program ((arbno procedure) (arbno class-decl) main-statement) a-program)
      (java-expression (expression ";") java-single-exp)
      (java-expression (statement ";") java-single-stat)
      (procedure ("proc(" type-exp identifier
            (arbno type-exp expression) ")" "{"
            (arbno java-expression) "}") proc-decl)
      (class-decl ("class" identifier "extends" identifier "{"
            (arbno (separated-list  variable ",")";") (arbno method-decl) "}")
            a-class-decl)
      (method-decl ("method" variable "(" (separated-list  variable ",") ")"
            "{" (arbno java-expression) "}") a-method-decl)
      (main-statement ("main()" "{"(arbno java-expression)"}") a-main-decl)
      (expression (number) lit-exp)
      (expression (string) string-exp)
      (expression (character) char-exp)
      (expression ("t") true-exp)
      (expression ("f") false-exp)
      (expression (variable) var-exp)
      (expression (identifier) id-exp)

      (expression ("("primitive (arbno expression)")") app-exp)
      (expression ("array" variable "=""{"(separated-list expression ",")"}")create-array)
      (expression ("[" expression expression "]") access-array-exp)

      (expression ("elementAt""(" expression expression ")") access-vector-exp)
      (expression ("addElement""(" expression expression ")") addelement-vector-exp)
```

```
                    (expression ("insertElementAt""(" expression expression expression")")
insert-vector-exp)

                    (expression ("struct" variable "(" (separated-list variable ",") ")")
structure-declaration)
                    (expression ("record" expression "(" (separated-list expression ",") ")")
struct-specifier)
                    (expression ("struct."expression"."variable) struct-selection)
                    (expression ("instanceOf""(" expression expression ")") struct-instanceof)

                    (expression ("¡""(" expression ")" expression ":" expression) ternary-exp)

                    (expression ("++" variable) increment-exp) ;increments a value by 1
                    (expression ("--" variable) decrement-exp) ;increments a value by 1

                    (expression ("new" identifier "(" (arbno expression) ")") new-object-exp)

                    (expression ("class." variable "." variable"("(arbno expression)")")
method-app-exp)

                    (expression ("super" identifier "("  (separated-list expression ",") ")")
                              super-call-exp)

                    (statement ("switch(" expression ")"
                              "{" (arbno "case" expression ":" (arbno java-expression) "break;")
                              "default" ":" (arbno java-expression) "}") switch-case-stat)
                    (statement ("for" "(" variable "=" expression ";" expression ";" expression ")"
                              "{" (arbno java-expression) "}") for-stat)
                    (statement ("while" "(" expression ")"
                              "{" (arbno java-expression ) "}") while-stat)
                    (statement ("do" "{" (arbno java-expression) "}"
                              "while" "(" expression ")") do-while-stat)
                    (statement ("set" variable operator expression) set-stat)
                    (statement ("if" "(" expression ")""{"(arbno java-expression)"}"
                              (arbno  "elseif("expression")""{" (arbno java-expression) "}")
                              "else""{"(arbno java-expression)"}")  if-stat)
                    (statement ("cases" expression "of" "{" (arbno patron) "}") cases-stat)
                    (statement ("local"(arbno variable) "in" (arbno java-expression) "," java-expression
"end") local-stat)

                    (patron (expression ":"  java-expression "break") patron-exp)

                    (operator ("-=") substract-assign-oper)
```

```
(operator ("+=") add-assign-oper)
(operator ("*=") mult-assign-oper)
(operator ("/=") div-assign-oper)
(operator ("=") assign-oper)

(primitive (identifier) procedure-prim)
(primitive ("+")            add-prim)
(primitive ("-") substract-prim)
(primitive ("*") mult-prim)
(primitive ("/") div-prim)
(primitive ("%")            modulo-prim)
(primitive ("pow")          pow-prim)
(primitive ("==")           equalto-prim)
(primitive ("max")          max-prim)
(primitive ("min")          min-prim)
(primitive ("<")            lessthan-prim)
(primitive (">")            greaterthan-prim)
(primitive ("<=")           lessthanorequalto-prim)
(primitive (">=")           greaterthanorequalto-prim)

(primitive ("&&")           and-prim)
(primitive ("||") or-prim)
(primitive ("!") not-prim)

(primitive ("newarray") array-prim)
(primitive ("setarray") setarray-prim)


(primitive ("cons") cons-prim)
(primitive ("car") car-prim)
(primitive ("cdr") cdr-prim)
(primitive ("null?") null-prim)
(primitive ("list") list-prim)
(primitive ("length") length-prim)
(primitive ("append") apend-prim)

(type-exp ("int") int-type-exp)
(type-exp ("char") int-type-exp)
(type-exp ("bool") bool-type-exp)
(type-exp ("(" (separated-list type-exp "*") "->" type-exp ")")
proc-type-exp)
(optional-type-exp ("?")
        no-type-exp)
        (optional-type-exp (type-exp)
                a-type-exp)
        ))
```

4.4. Implementación

```
(sllgen:make-define-datatypes scanner-spec the-grammar)

(define show-the-datatypes
  (lambda () (sllgen:list-define-datatypes scanner-spec the-grammar)))
;****************************************************************************************
; Scanner
; Scanner And parser
;****************************************************************************************
(define just-scan
  (sllgen:make-string-scanner scanner-spec the-grammar))
;
(define scan&parse
  (sllgen:make-string-parser scanner-spec the-grammar))
;****************************************************************************************
;Interpretador
;****************************************************************************************
(define i
  (sllgen:make-rep-loop  "<<"
                (lambda (pgm) (eval-program  pgm))
                (sllgen:make-stream-parser
                scanner-spec the-grammar)))

(define eval-program
  (lambda (pgm)
        (cases program pgm
        (a-program (procs class main)
                (let ((general-env (init-env)))
                (elaborate-class-decls! class)
                (elaborate-proc-decls! procs general-env)
                ;(eval-main main general-env)
                )
                ))))

(define eval-main
  (lambda (main env)
        (cases main-statement main
        (a-main-decl (jexps)
                (map
                (lambda (exp)
                (cases java-expression exp
                (java-single-exp (exp) (eval-expression exp env))
                (java-single-stat (stat)(eval-statement stat env)))
                ) jexps)
                ))))

(define eval-expression
  (lambda (exp env)
        (cases expression exp
        (lit-exp (datum) datum)
        (string-exp (string) string)
        (char-exp (char) char)
        (true-exp () #t)
        (false-exp () #f)
        (var-exp (var) (apply-env env var))
        (id-exp (id) (apply-env env id))
        (create-array (var rands)
                (let ((args (eval-rands rands env)))
                (setref! (apply-env-ref env var) (make-array-from-list args))))
        (access-array-exp (var field)
                (let ((array (eval-expression var env)))
                (array-ref array (eval-expression field env))
                ))
        (access-vector-exp (var field)
                (let ((evector (eval-expression var env)))
                (evector-ref evector (eval-expression field env))
                ))
        (addelement-vector-exp (var value)
                        (let ((evector (eval-expression var env)))
                        (addElement evector (eval-expression value env))))
        (insert-vector-exp (var pos value)
                (let ((evector (eval-expression var env)))
                (insertElementAt (eval-expression pos env) (eval-expression value env))))
        (structure-declaration (var ids)
                        (setref! (apply-env-ref env var)(a-struct var ids)))
        (struct-specifier (var rands)
                (let ((struct (eval-expression var env)) (args (eval-rands rands env)))
                (record-struct struct args)))
        (struct-selection (var field)
                (let ((instance (eval-expression var env)))
                (get-field instance field)))
```

```
                    (struct-instanceof (var inst)
                              (let ((struct (eval-expression var env))
                                        (instance (eval-expression inst env)))
                              (instanceof instance struct)
                              ))
                    (increment-exp (var)
                              (begin
                              (setref! (apply-env-ref env var)
                                        (apply-primitive (add-prim) (list (deref (apply-env-ref env var)) 1)
env))
                              (deref (apply-env-ref env var))
                              ))
                    (decrement-exp (var)
                              (begin
                              (setref! (apply-env-ref env var)
                                        (apply-primitive (substract-prim) (list (deref (apply-env-ref env var))
1) env))
                              (deref (apply-env-ref env var))
                              ))

                    (new-object-exp(class-name rands) (new-object class-name) )
                    (method-app-exp(obj-exp method-name rands)obj-exp)
                    (super-call-exp(method-name rands)method-name)

                    (ternary-exp (test-exp true-exp false-exp)
                              (if (eval-expression test-exp env)
                              (eval-expression true-exp env)
                              (eval-expression false-exp env)))
                    (app-exp (prim rands)
                    (let ((args (eval-rands rands env)))
                              (apply-primitive prim args)))
                    )))

          (define eval-statement
            (lambda (stat env)
                    (cases statement stat
                    (switch-case-stat (exp exps jexps jexp)
                              (cond
                              ((or (null? exps) (not (check-any-eq? exp exps jexps env)))
                              (cases java-expression (car jexp)
                                        (java-single-exp (exp) (eval-expression exp env))
                                        (java-single-stat (stat)(eval-statement stat env))))
                              (else
                    (cases java-expression (check-any-eq? exp exps (car jexps) env)
                              (java-single-exp (exp) (eval-expression exp env))
                              (java-single-stat (stat)(eval-statement stat env))))))
          (for-stat (id initexp exp2 exp3 body) id)
          (while-stat (test-exp body)
                    (if (eval-expression test-exp env)
                    (do-while test-exp body env) 0))
          (do-while-stat (body test-exp)
                    (do-while test-exp body env))
          (set-stat (var oper rhs-exp)
                    (cases operator oper
                    (assign-oper()
                              (begin (setref! (apply-env-ref env var) (eval-expression rhs-exp
env))))
                    (div-assign-oper()
                              (begin
                              (setref! (apply-env-ref env var)
                                        (/  (deref (apply-env-ref env var))(eval-expression rhs-exp
env)))))
                    (mult-assign-oper()
                              (begin
                              (setref! (apply-env-ref env var)
                                        (* (deref (apply-env-ref env var))(eval-expression rhs-exp
env)))))
                    (add-assign-oper()
                              (begin
                              (setref! (apply-env-ref env var)
                                        (+  (deref (apply-env-ref env var))(eval-expression rhs-exp
env)))))
                    (substract-assign-oper()
                                        (begin
                                        (setref! (apply-env-ref env var)
                                        (-  (deref (apply-env-ref env var))(eval-expression rhs-exp
env))))
                                        )
                    ))
          (if-stat (test-exp true-jexps elseif-exp true-elseif-jexps false-exps)
          (if (eval-expression test-exp env)
                    (map
                    (lambda (exp)
                    (cases java-expression exp
                    (java-single-exp (exp) (eval-expression exp env))
                    (java-single-stat (stat)(eval-statement stat env)))
          ) true-jexps)
          (if (null? elseif-exp)
          (map
          (lambda (exp)
          (cases java-expression exp
                    (java-single-exp (exp) (eval-expression exp env))
                    (java-single-stat (stat)(eval-statement stat env)))
          ) false-exps)
          (if (eval-expression (car elseif-exp) env)
          (map (lambda (exp)
                    (cases java-expression exp
                    (java-single-exp (exp) (eval-expression exp env))
                    (java-single-stat (stat)(eval-statement stat env)))
                    ) (car true-elseif-jexps))
          (map(lambda (exp)
                    (cases java-expression exp
                    (java-single-exp (exp) (eval-expression exp env))
                    (java-single-stat (stat)(eval-statement stat env)))
                    ) false-exps))
          )))
(cases-stat (exp patrons) (let ((element (eval-expression exp env)))
                    (let loop ((case  patrons))
                    (cases patron (car case)
                    (patron-exp (cond exp)
                              (if (eqv? (eval-expression cond env) element )
                                        (cases java-expression exp
                                        (java-single-exp (exp)(eval-expression exp env))
                                        (java-single-stat (stat)(eval-statement stat env)))
                                        (loop (cdr patrons))))
                    ))))
(local-stat (vars stats body)
          (let ((env-temp (extend-env vars (build-list (length vars)) env)))
          (map
          (lambda (exp)
          (cases java-expression exp
          (java-single-exp (exp) (eval-expression exp env-temp))
          (java-single-stat (stat)(eval-statement stat env-temp)))
          ) stats)
          (cases java-expression body
          (java-single-exp (exp) (eval-expression exp env-temp))
          (java-single-stat (stat)(eval-statement stat env-temp))))
          )
)
```

```
                    ))

          (define apply-primitive
            (lambda (prim args)
                    (cases primitive prim
                    (procedure-prim (proc-prim)(eopl:printf "prim ~a\t" proc-prim) (eopl:printf "args
~a" args) )
                    (add-prim () (apply + args))
                    (substract-prim ()(apply - args))
                    (mult-prim() (apply * args))
                    (div-prim() (apply / args))
                    (modulo-prim() (modulo (car args) (cadr args)))
                    (pow-prim() (expt (car args) (cadr args)))
                    (equalto-prim() (eqv? (car args) (cadr args)))
                    (max-prim() (let loop ((lst args))
                              (cond ((null? (cdr lst)) (car lst))
                              ((> (car lst) (loop (cdr lst))) (car lst))
                              (else (loop (cdr lst))))))
                    (min-prim() (let loop ((lst args))
                              (cond ((null? (cdr lst)) (car lst))
                              ((< (car lst) (loop (cdr lst))) (car lst))
                              (else (loop (cdr lst))))))

                    (lessthan-prim() (< (car args) (cadr args)))
                    (greaterthan-prim() (> (car args) (cadr args)))
                    (lessthanorequalto-prim() (<= (car args) (cadr args)))
                    (greaterthanorequalto-prim() (>= (car args) (cadr args)))
                    (and-prim() (and (car args) (cadr args)))
                    (or-prim() (or (car args) (cadr args)))
                    (not-prim() (not (car args)))

                    (array-prim () (simple-make-array (car args) (cadr args)))
                    (setarray-prim () (array-set! (car args) (cadr args) (cddr args)))

                    (cons-prim()(cons (car args) (cadr args)))
                    (car-prim() (car (car args)))
                    (cdr-prim() (cdr (car args)))
                    (null-prim() (if (null? (car args)) #t #f))
                    (list-prim() args)  ;; ya es una lista
                    (length-prim() args)   ;; existen dos
                    (apend-prim() (string-append (car args) (cadr args)))
                    )))
```

```
;*****************************************************************************************
;; Tipo de datos abstractos, funciones auxiliares (Ambientes,
;*****************************************************************************************
(define do-while
  (lambda (test-exp body-exp env)
          (cases java-expression (car body-exp)
          (java-single-exp (true-exp)
                  (let ((val (eval-expression true-exp env)))
                  (if (eval-expression test-exp env)
                          (do-while test-exp body-exp env)
                          val)))
          (java-single-stat (true-stat)
                  (let ((val (eval-statement true-stat env)))
                  (if (eval-expression test-exp env)
                          (do-while test-exp body-exp env)
                          val))))

          ))
;definición del tipo de dato ambiente para la ejecucion
;; <expression>          ::=
;;                  <empty-env-record>
;;                  ::=
;;                  <extended-env-record (syms vec env)
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
   (syms (list-of symbol?))
   (vec vector?)
   (env environment?))
  )

;función que busca un símbolo en un ambiente
(define apply-env
  (lambda (env sym)
          (deref (apply-env-ref env sym))))

(define apply-env-ref
  (lambda (env sym)
          (cases environment env
          (empty-env-record ()
                  (eopl:error 'apply-env-ref "No binding for ~s" sym))
          (extended-env-record (syms vals env)
                  (let ((pos (rib-find-position sym syms)))
                              (if (number? pos)
                              (a-ref pos vals)
                              (apply-env-ref env sym)))))))
;*********************************************************************************************
;Referencias

(define-datatype reference reference?
  (a-ref (position integer?)
          (vec vector?)))

(define deref
  (lambda (ref)
          (primitive-deref ref)))

(define primitive-deref
  (lambda (ref)
          (cases reference ref
          (a-ref (pos vec)
          (vector-ref vec pos)))))

(define setref!
  (lambda (ref val)
          (primitive-setref! ref val)))

(define primitive-setref!
  (lambda (ref val)
          (cases reference ref
          (a-ref (pos vec)
          (vector-set! vec pos val)))))

;^;;;;;;;;;;;;;; procedures ;;;;;;;;;;;;;;;

(define elaborate-proc-decls!
  (lambda (proc env) proc
          )
  )

(define-datatype procval procval?
  (closure
   (ids (list-of symbol?))
   (body expression?)
   (env environment?)))
```

```
(define apply-procval
  (lambda (proc args)
        (cases procval proc
        (closure (ids body env)
        (eval-expression body (extend-env ids args env))))))
;**************************************************************************************
;;
;empty-env: -> enviroment
;función que crea un ambiente vacío
(define empty-env
  (lambda ()
        (empty-env-record)))
;; extend-env: Función que crea un ambiente extendido
(define extend-env
  (lambda (syms vals env)
        (extended-env-record syms (list->vector vals) env)))
; funciones auxiliares para aplicar eval-expression a cada elemento de una
; lista de operandos (expresiones)
(define eval-rands
  (lambda (rands env)
        (map (lambda (x) (eval-rand x env)) rands)))
(define eval-rand
  (lambda (rand env)
        (eval-expression rand env)))
;init-env: -> enviroment
;Ambiente inicial
(define init-env
  (lambda ()
        (extend-env
        '(pi euler empty point)
        (list 3.14159265358979323846  2.7182818284590452354  empty (a-struct 'point
(list 'posx 'posy)))
        (empty-env))))
;definición del tipo de dato ambiente para los tipos
;; <expression>        ::=
;;                  <empty-env-record>
;;                  ::=
;;                  <extended-env-record (syms vec env)
(define-datatype type-environment type-environment?
  (empty-tenv-record)
  (extended-tenv-record
   (syms (list-of symbol?))
   (vals (list-of type?))
  (tenv type-environment?)))

(define type?  (lambda (x)  'falta_por_definir ))
(define empty-tenv empty-tenv-record)
(define extend-tenv extended-tenv-record)

(define apply-tenv
  (lambda (tenv sym)
         (cases type-environment tenv
         (empty-tenv-record ()
                 (eopl:error 'apply-tenv
                         "Variable ~s unbound in type environment" sym))
         (extended-tenv-record (syms vals tenv)
                         (let ((pos (list-find-position sym syms)))
                         (if (number? pos)
                         (list-ref vals pos)
                         (apply-tenv tenv sym))))
         )))
(define rib-find-position
  (lambda (sym los)
         (list-find-position sym los)))

(define list-find-position
  (lambda (sym los)
         (list-index (lambda (sym1) (eqv? sym1 sym)) los)))

(define list-index
  (lambda (pred ls)
         (cond
         ((null? ls) #f)
         ((pred (car ls)) 0)
         (else (let ((list-index-r (list-index pred (cdr ls))))
         (if (number? list-index-r)
                 (+ list-index-r 1)
                 #f))))))
;******************************************************************************************
;; ADT class
;; <class>  ::= <a-class(class-name super-name field-length field-ids methods)>
;;
(define-datatype class class?
  (a-class
   (class-name symbol?)
   (super-name symbol?)
   (field-length integer?)
   (field-ids (list-of symbol?))
   (methods method-environment?)))

(define elaborate-class-decls!
  (lambda (c-decls)
          (initialize-class-env!)
          (for-each elaborate-class-decl!  c-decls )))

(define elaborate-class-decl!
  (lambda (c-decl)
          (cases class-decl c-decl
          (a-class-decl (class-name super-name fields methods)

                    ;(begin

                    (add-to-class-env!
                    (a-class
                    class-name
                    super-name
                    (length (car fields))
                    (car fields)
                    (roll-up-method-decls c-decl super-name (car fields) methods)))

                    ;the-class-env)

                    )
          )
          ))

(define roll-up-method-decls
  (lambda (c-decl super-name fields methods)
          (map
          (lambda (m-decl)
          (a-method m-decl super-name fields))
          methods)))

;; The class environment
(define the-class-env '())
(define initialize-class-env!
  (lambda ()
          (set! the-class-env '())))
```

```
(define add-to-class-env!
  (lambda (class)
          (set! the-class-env (cons class the-class-env))))
;***************************************************************************************
;; ADT object
;; <object>  ::= <an-object(class-name fields)>
;;
(define-datatype object object?
  (an-object
   (class-name symbol?)
   (fields vector?)))

(define new-object
  (lambda (class-name)
          (an-object
          class-name
          (make-vector 2)))) ;\new1

;***************************************************************************************
;; ADT method
;; <method>  ::= <a-method(method-decl super-name field-ids)>
;;
(define-datatype method method?
  (a-method
   (method-decl method-decl?)
   (super-name symbol?)
   (field-ids (list-of symbol?))))

(define method-environment?
  (list-of method?))

;***************************************************************************************
;; ADT record or struct
;; <struct>  ::=
;;                    <struct(name fields)>
(define-datatype struct struct?
  (a-struct (name symbol?)
          (fields (list-of symbol?))
          ))
;Example
(define complex (a-struct 'complex '(real imag)))
(define point (a-struct 'point '(x y)))
(define polar-point (a-struct 'polar-point '(r tetha)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ADT intance
;; instance(template values) -> instance
(define-datatype instance instance?
  (a-instance
   (template struct?)
   (values list?)))
;;Example
;(define instancetest (a-instance complex (list 4 7)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;record-struct (list <struct> values) -> adt_instance
;;make a instance of a struct
(define record-struct
  (lambda (str args)
          (cases struct str
          (a-struct (name fields)
                    (if (eq? (length fields)(length  args))
                    (a-instance str args)
                    (eopl:error 'make-struct "Wrong number of arguments to struct"))))
          )
  )
;(record-struct complex (list  3 1))
;(record-struct point (list  10 12))
;(record-struct polar-point (list  2 45))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;get-field (<struct>) -> value
;;get field from struct
(define get-field
  (lambda (record field)
          (cases instance record
          (a-instance (template values)
                    (cases struct template
                    (a-struct (name fields)
                              (list-ref  values (list-find-position field fields)))
                    )))))
;;Example
;(get-field (a-instance complex (list 4 7)) 'real)
;(get-field (a-instance polar-point (list 4 45)) 'r)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;instanceof (list <struct> values) ->boolean
;;check what an element is instance of struct
(define instanceof
  (lambda (inst struct)
        (cases instance inst
        (a-instance (template values)
                (and (eqv? template struct) (not (null? values))))
        )))
;(instanceof (list (record-struct complex (list  3 1)) complex))
;(instanceof (list (record-struct point (list  10 12)) point ))
;(instanceof (list (record-struct polar-point (list  2 45)) polar-point))
;*************************************************************************************
;; ADT extensible vector <evector>
;; <evector>  ::=
;;                  <evector(currente-capacity fill automatic-expansion)>
(define-datatype evector evector?
  (a-evector (current-capacity vector?)
        (fill  vector?)
        (automatic-expansion boolean?))
  )
;;Example
(define evectest (a-evector (vector 5) (vector (list (vector 1)(vector 2)(vector 3)(vector
5)(vector 5))) #t))
(evector? evectest)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; evector-ref(evector int) -> scheme-value
;; Returns the element in slot pos of vec. The first slot is position 0,
;; and the last slot is one less than (vector-length vec).
(define evector-ref
  (lambda (evec pos)
        (cases evector evec
        (a-evector (current-capacity  fill automatic-expansion)
                (if(> pos (- (vector-ref current-capacity 0) 1))
                (eopl:error 'evector-ref:" index out of range; given: ~a" pos)
                (vector-ref (list-ref  (vector-ref fill 0) pos) 0)
                )))))
;Example
(evector-ref evectest 0)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; evector-set!(evector int scheme-value) -> int
;;Updates the slot pos of vec to contain v.

(define list-set            ;;Updates the slot pos of list to contain v.
  (lambda(list n val)
        (cond
        ((null? list) '())
        ((eq? n 0) (cons (vector-set! (car list) 0 val)(cdr list)))
```

```
                (#t (cons (car list) (list-set (cdr list) (- n 1) val))))))

        (define evector-set!
          (lambda (evec pos nval)
                (cases evector evec
                (a-evector (current-capacity  fill automatic-expansion)
                        (if(< pos 0)(eopl:error 'evector-set!:" index must be a non-negative integer;
given: ~a" pos)
                        (cond
                        [(< pos (vector-ref current-capacity 0))(begin (list-set (vector-ref fill 0) pos
nval) 0)]
                        [automatic-expansion  (begin (evector-expands-nonautomatic evectest (+
pos 1))
                                        (list-set (vector-ref fill 0) pos nval)
                                        1)]
                        [else (eopl:error 'evector-set!: "index out of range; given: ~a" pos )]
                        ))
                        ))))
        ;Example
        (evector-set! evectest 0 12)
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;; vec-expands-nonautomatic(evector int) -> int
        ;; increment size vector
        (define build-list ;;Creates a list of n elements whith 'null values by default
          (lambda (max)
                (let f ((i max)(a '()))
                (if (eq? 0 i)
                a
                (f (- i 1) (cons (vector 'null) a))))))

        (define evector-expands-nonautomatic
          (lambda (evec size)
                (cases evector evec
                (a-evector (current-capacity fill automatic-expansion)
                        (if (< size (vector-ref current-capacity 0))
                        (eopl:error 'evector-expands-nonautomatic "use
evector-reduce-nonautomatic by reduce evector")
                        (if (< size (+ (length (vector-ref fill 0))1)) (begin (vector-set!
current-capacity 0 size) 0)

                        (begin (vector-set! fill  0 (append (vector-ref fill 0) (build-list (-
size(vector-ref current-capacity 0)))))
                                (vector-set! current-capacity 0 size)
                        1)))
                ))))
        ;;Example
        (build-list 2)
        (evector-expands-nonautomatic (a-evector (vector 1) (vector (list (vector 1))) #t) 2)
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;; vec-expands-nonautomatic(evector int) -> int
        ;; Automatically incremented the double size vector when its size
        ;; becomes greater than its capacity.
        (define evector-expands-automatic
         (lambda (evec)
            (cases evector evec
            (a-evector (current-capacity fill automatic-expansion)
                (if(not automatic-expansion)(eopl:error 'evector-expands-automatic: "
vector not automatic-expansions")
                (let([size (vector-ref current-capacity 0) ])
                (if (< size (length (vector-ref fill 0)))
                (begin (vector-set! current-capacity 0 (* size 2)) 0)
                (begin (vector-set! current-capacity 0 (* size 2))
                        (vector-set! fill 0 (append (vector-ref fill 0) (build-list size)))
                        1))))
                ))))
        ;;Example
        (evector-expands-automatic (a-evector (vector 1) (vector (list (vector 80000))) #t))
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;; evector-reduce-nonautomatic(evector int) -> int
        ;; Trims the capacity of this vector to be the int size.
        ;; minimun value 1
        (define reduce-list
         (lambda (list n)
            (let f ((i n)(a '()))
            (if (eq? i (length list))
            list
            (f (+ i 1) (list-set list i 'null))
            ))))
        ;(reduce-list (list (vector 1) (vector 2)(vector 3)(vector 4)(vector 5)) 4)
        (define evector-reduce-nonautomatic
         (lambda (vec size)
            (cases evector vec
            (a-evector (current-capacity fill automatic-expansion)
                (if (> 1 size ) (eopl:error 'vec-reduce-nonautomatic "size out of range;
given: ~a minimun size 1" size)
                (begin (vector-set! current-capacity 0 size)
                                        (vector-set! fill 0 (reduce-list (vector-ref fill 0) size))
                                        1))
                              ))))
          ;;Example
          (evector-reduce-nonautomatic (a-evector (vector 4) (vector (list (vector 80000)(vector
70000)(vector 60000)(vector 50000))) #t) 2)
          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          ;; evector-length(evector) -> int
          ;; Trims the capacity of this vector to be the int size.
          ;; minimun value 1
          (define evector-length
            (lambda(vec)
                    (cases evector vec
                    (a-evector (current-capacity values automatic-expansion)(vector-ref
current-capacity 0))
                    )))
          ;;Example
          (evector-length evectest)
          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          ;; addElement(evector value) -> evector
          ;; Adds an element to the start of the vector. +The vector is automatically grown
          (define addElement
            (lambda(evec elem)
                    (cases evector evec
                    (a-evector (current-capacity fill automatic-expansion)
                              (cond
                              ((not automatic-expansion)(eopl:error 'addElement: " vector not
automatic-expansions"))
                              (else(begin
                              (vector-set! current-capacity 0 (+(vector-ref current-capacity 0)1))
                              (vector-set! fill  0 (cons (vector elem) (vector-ref fill 0))))))))))
          ;;Example
          (addElement evectest 0)
          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          ;; insertElementAt(evector pos value) -> evector
          ;;  insert the specified object s a component in this vector at the specified index
          (define list-set-append          ;;Updates the slot pos of list to contain value within change
other values.
            (lambda(list n val)
                    (cond
                    ((null? list) '())
                    ((eq? n 0) (cons (vector val) list))
                    (#t (cons (car list) (list-set-append (cdr list) (- n 1) val))))))
```

```
(define insertElementAt
  (lambda(evec pos value)
        (cases evector evec
        (a-evector (current-capacity fill automatic-expansion)
              (cond
              ((not automatic-expansion)(eopl:error 'addElement: " vector not
automatic-expansions"))
              (else(begin
              (vector-set! current-capacity 0 (+ (vector-ref current-capacity 0) 1))
              (vector-set! fill  0 (list-set-append (vector-ref fill 0) pos value)))))))))
;;Example
(insertElementAt evectest 4 4)
;*******************************************************************************************
;;Definicion del tipo del tipo de dato vector
;; <array>  ::=
;;                 <array( vec )>
(define-datatype array array?
  (an-array (vec vector?)))
;;Example
(define arrtest (an-array (vector 1 5 8 7 9 4 6)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;array-length (array) -> int
;;Returns the length of array
(define array-length
  (lambda (arr)
        (cases array arr
        (an-array (vec) (vector-length vec)))
        ))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;simple-make-array (size val) -> array
;;Constructs an empty array with an initial value in all slot.
(define simple-make-array
  (lambda (size val)
        (if (> size 0)
        (an-array (make-vector size val))
        (eopl:error 'simple-make-array:"non-negative number by size"))))
;;Example
(define arrsimple (simple-make-array 10 5))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;make-array-from-list (list) -> array
;;Constructs an array from list.
(define make-array-from-list
  (lambda (values-list)
        (if (> (length values-list) 0)
        (an-array (list->vector values-list))
        (eopl:error 'make-array-from-list:"non-negative number by size")
        )))
;;Example
(define arrfromlist (make-array-from-list (list 7 8 9 4 5 2 1)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;array-ref (array pos) -> val
;;Returns the element in slot pos of array
(define array-ref
  (lambda (arr pos)
        (cases array arr
        (an-array (vec) (vector-ref vec pos)))))
;;Example
(array-ref arrtest 3)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;array-set! (array pos value) -> <void>
;;Updates the slot pos of array to contain value.
(define array-set!
  (lambda (arr pos val)
        (cases array arr
        (an-array (vec)
                (vector-set! vec pos val)))))
;;Example
(array-set! arrtest 3 148)
```

;Auxiliar procedure check-any-eq?, receives a expression, a list of expressions and a list of java-expressions,
;;this procedure returns false if list of expressions is empty, else, looks for in the list of expressions some
;;that coincides with the expression and returns the java-expressions that is in the same position in the java-expresions list.

```
(define check-any-eq?
  (lambda (exp exps jexps env)
        (cond
        ((null? exps) #f)
        ((eq? (eval-expression exp env) (eval-expression (car exps) env)) (car jexps))
        (else (check-any-eq? exp (cdr exps) (cdr jexps) env)))))
```

4.4. Ejemplos

```
(scan&parse "
main()
{
(+ 4 5 6 7);
}")

(scan&parse "main(){
local
y
in
set y=0;
do{
 set y += pi;
}while((<= y (* pi 5)));
,y;
end;
}")

(scan&parse "main(){
local
y
in
set y=0;
while((<= y (* pi 5))){
 set y += pi;
};
,y;
end;
}")

(scan&parse "main()
{
local
x y
in
set x = 10;
set y = 5;,
if((&& (<= x 6) (== y 5)))
{
++x;
}
elseif((&& (> x 6) (<= y 4)))
{
(- x y);
}
else
{
--x;
};
end;
}")
(scan&parse "main()
{
local
x y
in
set x = 10;
set y = 5;,
¡((&& (<= x 6) (== y 5))) ++x : ¡((&& (> x 6) (<= y 4))) (- x y) : --x ;
end;
}")
(scan&parse "proc(int F int x int y int z)
{
if(x)
{
(max y z);
}
else
{
(min y z);
};
}
main()
{
local
m
in
set m = 0;,                              //aqui Error falta \",\"
(F true m -4);
end;
}")
(scan&parse "main()
{
local
x                      //Ese local es solo de creacion de variables
in
set x = 0;
set x += 10;,
x;
end;
}")
(scan&parse "main()
{
local
x y
in
set x = (cons 5 (cons 6 empty));
set y = (cdr x);,
y;
end;
}")
(scan&parse "proc(int F int x int y)
{
switch(x)
{
case 1:
(+ y 1);
break;
case 2:
(+ y 2);
break;
case 3:
(+ y 3);
break;
default:
y;
};
}
main()
{
local
m
in
set m = 5;,
(* (F 2 m) 2);                    // Dieferncia entre sentencia y expression
end;
}")

(scan&parse "proc(int Sum int x)
{
```

```
local
sum
in
for(y=1; (<= y x); ++y)
{
set sum += y;
};,                              //Dieferncia entre sentencia y expression
sum;
end;
}
main()
{
(Sum 10);                              //ejemplo con ;
}")

(scan&parse "proc(int Fact int x)
{
local
res y
in
set y=1;
set res = 1;
while((<= y x))
{
set res *= y;
--y;
};,
res;
end;
}
main()
{
(Fact 5);
}")

(scan&parse "proc(int TestCaseStruct int str)
{
cases str of {
  1:  set a = struct.str.field1; break
  str:  set istr = record str (1024 , struct.str.field2); break
  3:  set a = 1024; break
  5:  set a = 'a'; break
 };
```

```
}
main()
{
local
 str istr
in
struct str (field1,field2);
set istr = record str (4,6);,
(TestCaseStruct  istr);
end;
}")

(scan&parse "main()
{
local
 str istr
in
struct str (field1,field2);
set istr = record str (4,6);,
cases str of {
  1:  set a = struct.str.field1; break
  str:  set istr = record str (1024 , struct.str.field2); break
  3:  set a = 1024; break
  5:  set a = 'a'; break
 };
end;
}")

(scan&parse "class C1 extends Object
{
i,j;
method c1(i,j)
{
set i = 1;
set j = 1;
}
}
main()
{
local
t1 t2 o1
in
,set o1 = new C1(3);
```

```
end;
}")

(scan&parse "
main(){
local
ar val
in
array ar = {1,2,3,4,5,6,7};
//set ar = (newarray 10 2);
set val=[ar 4];
set val=[ar 6];,
val;
end;
}")

(scan&parse "main(){
local
ar val
in
array ar = {1,2,3,4,5,6,7};
set val = [ar 0];
,val;
end;
}")

(scan&parse "main(){
local
x
in
set x = 1;,
switch(x){
case 1:
(+ 1 1);
break;
case 2:
(- 2 2);
break;
case 3:
(* 3 3);
break;
case 4:
(/ 4 4);
break;
default:
x;
};
end;
}")
```