TP 3: intérprete de C en LISP (alias “máquina virtual de C en LISP”)

Introducción

Se deberá programar un intérprete de C en LISP. El código que ingresará al intérprete tendrá ciertas restricciones:

- Se utilizará solo un tipo de variables: enteras.
- Se podrá utilizar las funciones “printf” y “scanf” (solo con variables enteras)
- Las estructuras de control a reconocer por el intérprete son if-then-else y while
- El código a interpretar no será estrictamente C sino que será una lista de listas con las instrucciones C (código intermedio entre C y LISP)

Ejemplo: código en C

```
int x;
int z, A = 10;
void main() {
     z = A + 1;
     printf("%d", A);
     scanf("%d", &x);
     if (A < X) {
          z += A;
     } else {
          z = 1;
     }
     while(x < 10) {
          printf("%d", A);
          x++;
     }
}
```

Código que recibirá el intérprete

```
(
     (int x)
     (int z A = 10)
     (main (
               (z = A + 1)
               (printf A z)
               (scanf x)
               (if (a < X) (
                         (z += A)
               ) else (
                         (z = 1)
               ))
               (while (x < 10) (
                         (printf A z)
                         (x++)
               ))
     ))
)
```

Código dado en clase

Función RUN

```
( DEFUN RUN ( PRG ENT &OPTIONAL ( MEM NIL ) )
    ( IF ( NULL PRG ) NIL
        ( IF ( EQ ( CAAR PRG ) `int )
            ( RUN ( CDR PRG ) ENT ( AGREGAR_MEM ( CDAR PRG ) ) ; Si antes del main hay una declaracion, la agrega a la memoria
            ( IF ( EQ ( CAAR PRG ) `main )
                ( EJEC ( CADAR ( CAR PRG ) ) ENT MEM )
            )
        )
    )
)
```

Variables y aclaraciones:

- PRG: programa a ejecutar en forma de lista
- ENT: entrada de datos que recibirá el programa
- MEM: memoria en formato de lista
- AGREGAR_MEM: funcion que agrega a la memoria el nombre y valor de una variable

Función EJEC

```
( DEFUN EJEC ( PROG ENT MEM ( &OPTIONAL ( SAL NIL ) ) )
    ( IF ( NULL PROG )
        ( REVERSE SAL )
        ( COND
            ( ( ES_VAR ( CAAR PROG ) )
                ( IF ( EQ ( CADAR PROG ) `= ) ; Si es el igual, se trata de una asignacion
                    ( EJEC ( CDR PROG ) ENT ( ASIGNAR … ) SAL )
                    ( EJEC ( CONS ( NUEVA_ASIG ( CAR PROG ) ) ( CDR PROG ) ) ENT MEM SAL )
                )
            )
            ( ( PERT ( CAAR PROG ) `(++ –) ) ; Si el primer elemento es un ++ o un  - -, se traduce
                ( EJEC ( CONS ( REVERSE ( CAR PROG ) ) ( CDR PROG ) ) ENT MEM SAL )
            )
            ( ( EQ ( CAAR PROG ) `scanf )  ; Si es un scanf se lee de la entrada y se modifica la memoria
                ( EJEC ( CDR PROG ) ( CDR ENT ) ( ASIGNAR ( CADAR PROG ) ( CAR ENT ) MEM ) SAL )
            )
            ( ( EQ ( CAAR PROG ) `printf ) ; Si es un printf se imprime en la salida
                ( EJEC ( CDR PROG ) ENT MEM ( CONS ( VALOR ( CADAR PROG ) MEM ) SAL ) )
            )
            ( ( EQ ( CAAR PROG ) `if )
                ( IF ( NOT ( EQ ( VALOR ( CADAR PROG ) MEM ) 0 ) ) ; Si el valor de la condición no es 0
                    ( EJEC ( APPEND ( CADDAR PROG ) ( CDR PROG ) ) ENT MEM SAL )
                ( IF ( EQ ( LENGTH ( CAR PROG ) ) 5 ) ; Si hay 5 elementos, es porque hay un else: IF CONDICION_IF CUERPO_IF ELSE CUERPO_ELSE
                    ( EJEC ( APPEND ( CADDDDAR PROG ) ( CDR PROG ) ) ENT MEM SAL )
                    ( EJEC ( CDR PROG ) ENT MEM SAL )
                )
            )
            ( ( EQ ( CAAR PROG ) `while )
                ( IF ( EQ ( VALOR ( CADAR PROG ) MEM ) 0 )
                    ( EJEC ( CDR PROG ) ENT MEM SAL )
                    ( EJEC ( APPEND ( CADDAR PROG ) PROG ) ENT MEM SAL )
                )
            )
        )
    )
)
```

Variables y aclaraciones:

- *Código del while*: si se encuentra un while:
  while (condicion) {
  - Código del cuerpo del while -
  }

  se evalúa la condición. Si la condicion es verdadera el programa pone a ejecutar lo siguiente:
  - Código del cuerpo del while -
  while (condicion) {
  - Código del cuerpo del while -
  }

  De esa forma ejecuta el código del while y luego vuelve a evaluar la condición (si la condición vuelve a ser verdadera ejecuta recursivamente lo mismo que antes)
- NUEVA_ASIG: guarda en la memoria la asignación que se pasa por parámetro en forma de lista

Función VALOR

```
( DEFUN VALOR ( EXP MEM &OPTIONAL ( OPERAD NIL ) ( OPERAN NIL ) )
    ( IF ( AND ( ATOM EXP ) ( NOT ( NULL EXP ) ) )
        ( IF ( NUMBERP EXP )
            EXP
            ( BUSCAR EXP MEM )
        )
        ( IF ( NULL EXP )
            ( IF ( NULL OPERAD )
                ( CAR OPERAN )
                ( VALOR EXP MEM ( CDR OPERAD ) ( CONS ( APPLY ( CAR OPERAD ) ( LIST ( CADR OPERAN) ( CAR OPERAN ) ) ( CDDR OPERAN ) ) )
            )
            (  IF ( ES_OPERADOR ( CAR EXP ) )
                ( IF ( NULL OPERAD )
                    ( VALOR ( CDR EXP ) MEM ( CONS ( CAR EXP ) OPERAD OPERAN )
                    ( IF ( < ( PESO ( CAR OPERAD ) ) ( PESO ( CAR EXP ) ) )
                        ( VALOR ( CDR EXP ) MEM ( CONS ( CAR EXP ) OPERAD ) OPERAN )
                        ( VALOR EXP MEM ( CDR OPERAD ) ( CONS ( APLICAR ( CAR OPERAD ) ( LIST ( CADR OPERAN) ( CAR OPERAN ) ) ( CDDR OPERAN ) ) )
                    )
                )
                ( VALOR ( CDR EXP ) MEM OPERAD ( CONS ( VALOR ( CAR EXP ) MEM ) OPERAN ) )
            )
        )
    )
)
```

Variables y aclaraciones:

- OPERAN: lista de operandos
- OPERAD: lista de operadores
- PESO: función que determina la prioridad de un operador. Ejemplo: en una expresión como: “a * b + c” el operador * tiene mayor prioridad que el “+” (la evaluación del producto se hace antes que la suma)
- APLICAR: función similar a APPLY. La diferencia es que en C el valor NIL y T se representan por 0 y cualquier valor que no sea 0.

Explicación sobre el algoritmo usado por la función valor

La función valor realiza la traducción desde el código intermedio C-LISP a LISP. Para ello utiliza una pila de operadores, una lista de operandos y la

expresión que se va interpretando. Los operandos se apilan o se procesan dependiendo de su prioridad o peso.

Ejemplos:

Expresión: (a + b * c)

Operadores	Operandos	Expresión
( )	()	(a + b * c)
( )	(a)	(+ b * c)
(+)	(a)	(b * c)
(*+)	(b a)	(* c)
(*+)	(c b a)	( )
(+)	((* b c) a)	( )
(+)	(+ a (* b c))	( )

Expresión: (a * b + c)

Operadores	Operandos	Expresión
( )	()	(a * b + c)
( )	(a)	(* b + c)
(*)	(a)	(b + c)
(*)	(b a)	(+ c)
( )	((* a b))	(c)
(+)	((* a b))	( )
(+)	(c (* a b))	( )
( )	(+ (* a b) c)	()

Expresión: (a + b * c < d + e)

Operadores	Operandos	Expresión
( )	( )	(a + b * c < d + e)
( )	(a)	(+ b * c < d + e)
(+)	(a)	(b * c < d + e)
(+)	(b a)	(* c < d + e)
(* +)	(b a)	(c < d + e)
(* +)	(c b a)	(< d + e)
(+)	((* b c) a)	(< d + e)
( )	(+ a (* b c))	(< d + e)
(<)	(+ a (* b c))	(d + e)
(<)	(d (+ a (* b c)))	(+ e)
(+ <)	(d (+ a (* b c)))	(e)
(+ <)	(e a (+ a (* b c)))	( )
(<)	((+ d e) (+ a (* b c)))	( )
( )	((< (+ a (* b c)) (+ d e)))	( )

Expresión: (a – b – c)

Operadores	Operandos	Expresión
( )	( )	(a – b – c)
( )	(a)	(- b – c)
(-)	(a)	(b - c)
(-)	(b a)	(- c)
( )	((- a b))	(- c)

(-)	((- a b))	(c)
(-)	(c - (a b))	( )
( )	((- (- a b) c))	( )