PROCESADORES DEL LENGUAJE

Práctica Final: Frontend y Backend

Grupo: 113-leal-zhu
ID de grupo de prácticas: 81

Nombre de los participantes:
Liang Ji Zhu
Ignacio Leal Sánchez
Correos electrónicos de los participantes:
100495723@alumnos.uc3m.es
100495680@alumnos.uc3m.es

Madrid, 9 de mayo de 2025

Índice

1. Introducción

La presente memoria documenta el desarrollo integral de un traductor frontend y un traductor backend. El trabajo plantea la construcción de éstas en dos fases:

- Frontend: traduce el lenguaje C, de notación infija, a código Lisp, de notación prefija.
- Backend: convierte dicho código Lisp a notación postfija, Forth, siguiendo la filosofía de máquina de pila integrada en gforth.

Se sifuió un ciclo incremental para la realización del proyecto, en cada iteración se añadió una nueva producción. Esta estrategia permitió aislar errores tempranos y dosificar la complejidad, ya que los apartados iniciales eran más asequibles y los últimos exigían mayor autonomía y un diseño gramatical más depurado.

Se puede apreciar 3 etapas de implementación aproximadas:

- Etapa 1: se implementaron las declaraciones básicas, la función `main` y las primeras sentencias de E/S (`puts`, `printf`).
- Etapa 2: se integraron operadores con su precedencia y las estructuras de control `while` e `if/else`.
- Etapa 3: se añadieron funciones con retorno estructurado y vectores.
- Etapa 4: desarrollo del backend y pruebas funcionales con la directiva `//@ (main)`, completando el flujo C → Lisp → Forth.

El analizador léxico se mantuvo sin modificaciones; toda la lógica nueva se centró en la gramática y las acciones semánticas, añadiendon los tokens necesarios.

Cada incremento semanal concluyó con una regresión completa sobre el conjunto de casos de prueba —tanto los oficiales como los diseñados ad-hoc— y con la documentación de las decisiones en esta memoria. Así, el resultado final es un traductor robusto cuya construcción refleja paso a paso la ruta marcada por las pautas del enunciado, desde las bases del frontend hasta los detalles finales del backend.

2. Transparencia

En este apartado constatamos que el uso de la IA generativa ha sido utilizado para la corrección de errores en las declaraciones de C, aunque ha sido intercalado con la asistencia de los profesores que proponen respuestas más coherentes y simples. Y ha sido imposible usar la inteligencia artificial en más partes del trabajo debido a su incapacidad para comprender los requisitos específicos del problema, y por ello dando respuestas de nula utilidad.

Asimismo, hacemos constar que la carga de trabajo se ha repartido de forma justa entre los dos. A lo largo de todo el período —y, muy especialmente, durante la fase final— la dedicación fue dinámica: cada miembro asumió más tareas cuando su disponibilidad lo permitía y cedió responsabilidad cuando sus circunstancias lo exigían, garantizando en todo momento un equilibrio global.

3. Traducción Frontend

3.1. Variables Globales

Las variables se declaran como enteros o como arrays en los que se puede dar valor o no, en este caso se usará el valor 0 para la traducción a lisp. Se permite que se declaren múltiples variables en línea separadas con ",.º en cascada.

```
1  axioma:              var_global funcion
2                       r_axioma
3                       ;
4  r_axioma:
5                       |   axioma
6                       ;
7
8  var_global:          declaracion ';'  var_global
9                       |
10                      ;
11
12 declaracion:         INTEGER  IDENTIF valor_global
      r_declaracion
13                      | INTEGER  IDENTIF '[' NUMBER ']'
                           r_declaracion
14                      ;
15
16 valor_global:
```

```
17                      | '=' NUMBER
18                      ;
19 r_declaracion:       ',' IDENTIF valor_global
      r_declaracion
20                      | ',' IDENTIF '[' NUMBER ']'
                           r_declaracion
21                      |
22                      ;
```

3.2. Función MAIN, Funciones Genéricas y Variables Locales

Se definen las funciones de dos maneras: las funciones genéricas y el main. El main es una función diferente ya que es obligatorio que exista esta función. Dentro del cuerpo de las funciones pueden declararse variables, que tanto en C como en Lisp serán variables locales. Se emplea la misma traducción de las globales, pero en este caso deben generarse dentro del cuerpo de la función. Por lo que también permitimos la declaración de variables locales que serán la variable seguida del nombre de la función en la que es variable local.

```
1  funcion:             IDENTIF  '(' argumento ')' '{'
      var_local cuerpo '}' funcion
2                       | funcion_principal
3                       ;
4
5  funcion_principal:   MAIN  '(' argumento ')' '{' var_local
       cuerpo '}'
6                       ;
7
8  argumento:           INTEGER valor resto_argumento
9                       | valor resto_argumento
10                      |
11                      ;
12
13 valor:               STRING
14                      | expresion
15                      ;
16
17 resto_argumento:     ',' argumento
18                      |
19                      ;
20
```

```
21
22 var_local:          declaracion_local ';' var_local
23                     |
24                     ;
25
26 declaracion_local:  INTEGER  IDENTIF valor_local
      r_decl_local
27                     | INTEGER  IDENTIF '[' NUMBER ']'
                          r_decl_local
28                     ;
29
30 valor_local:        /* lambda */
31                     | '=' NUMBER
32                     ;
33 r_decl_local:       ',' IDENTIF valor_local r_decl_local
34                     | ',' IDENTIF '[' NUMBER ']'
                          r_decl_local
35                     |
36                     ;
```

3.3. Impresión de Cadenas

La impresión de cadenas de texto se hace imprimiendo o un string o una variable. No existe la forma de impresión formateada como en C. Por ello cualquier impresión de variables en C se traducirá a la impresión de variables individuales en Lisp.

```
1  sentencia:          asignacion
2                      | '@' expresion
3                      | PUTS '(' STRING ')'
4                      | PRINTF printf
5                      | llamada
6                      ;
7  printf:             '(' STRING r_printf ')'
8                      ;
9  r_printf:           ',' expresion r_printf
10                     | ',' STRING r_printf
11                     |
12                     ;
```

3.4. Operadores, Precedencia, Asociatividad y Vectores

Aquí hemos cambiado la forma de operar respecto a la versión anterior. Para garantizar la mejor precedencia sin ninguna posibilidad de errores hemos dividido las diferentes operaciones o comparaciones en No Terminales específicos de forma que la precedencia pueda ser cumplida en todo momento. Para declarar un Vector podemos usar una sintaxis de Lisp extendiendo la que usamos para variables y podrán declararse como variables globales o locales.

```
1  var_global:          declaracion ';'  var_global
2                       |
3                       ;
4
5  declaracion:         INTEGER  IDENTIF valor_global
      r_declaracion
6                       | INTEGER  IDENTIF '[' NUMBER ']'
                           r_declaracion
7                       // Vector Global
8                       ;
9  ...
10 /* ==== Operadores, precedencia y asociatividad ==== */
11 asignacion:          IDENTIF '=' expresion
12                      | vector '=' expresion
13                      // Vector Local
14                      ;
15
16 expresion:           logical_or
17                      ;
18
19 llamada:             IDENTIF '(' argumento ')'
20                      ;
21
22 logical_or:          logical_and
23                      | logical_or OR logical_and
24                      ;
25 logical_and:         igualdad
26                      | logical_and AND igualdad
27                      ;
28 igualdad:            relacional
29                      | igualdad EQ relacional
30                      | igualdad NE relacional
31                      ;
32 relacional:          aditivo
```

```
33                      | relacional '<' aditivo
34                      | relacional '>' aditivo
35                      | relacional LE aditivo
36                      | relacional GE aditivo
37                      ;
38 aditivo:             multiplicativo
39                      | aditivo '+' multiplicativo
40                      | aditivo '-' multiplicativo
41                      ;
42 multiplicativo:      unario
43                      | multiplicativo '*' unario
44                      | multiplicativo '/' unario
45                      | multiplicativo '%' unario
46                      ;
47 unario:              operando
48                      | '!' unario
49                      | '+' operando %prec UNARY_SIGN
50                      | '-' operando %prec UNARY_SIGN
51                      ;
52
53 operando:            IDENTIF
54                      | IDENTIF '(' argumento ')'
55                      | NUMBER
56                      | '(' logical_or ')'
57                      |  vector
58                      ;
59
60 vector:              IDENTIF '[' logical_or ']'
61                      ;
```

Se declararon *tokens* para los operadores lógicos y de comparación con el objetivo de mejorar la legibilidad del código y establecer un acoplamiento más claro entre las fases de análisis léxico y análisis sintáctico.
En particular, los operadores de comparación fueron definidos como `nonassoc`, ya que, a diferencia de operadores como la suma o la multiplicación, las comparaciones (por ejemplo, `==`, `!=`) no son asociativas. Esto significa que una expresión como:

```
a == b == c
```

no tiene un significado bien definido en C. Existen dos posibles formas de interpretarla:

```
(a == b) == c || a == (b == c)
```

Ambas interpretaciones pueden generar ambigüedad semántica. Al declarar estos operadores como `nonassoc`, el *parser* rechaza expresiones como `a == b == c` y obliga al programador a escribirlas de manera explícita, por ejemplo:

- `(a == b) && (b == c)`
- Uso de paréntesis para aclarar el orden de evaluación

Este enfoque permite capturar errores de lógica o errores tipográficos en tiempo de compilación, evitando comportamientos ambiguos o inesperados en tiempo de ejecución.

```
...
%token EQ
%token NE
%token LE
%token GE
%token OR
%token AND

%right '='
%left OR
%left AND
%nonassoc EQ NE
%nonassoc '<' '>' LE GE
%left '+' '-'
%left '*' '/' '%'
%right UNARY_SIGN "!"
```

3.5. Estructuras de Control

Las estructuras de control se encuentran dentro del cuerpo de una función y no acaban en ";". Estas son el IF-ELSE, FOR y WHILE. Cuentan con una estructura de cuerpo diferente ya que un return de alguna de estas estructuras se interpreta diferente ya que no es un return al final de la función.

```
cuerpo:             sentencia ';' cuerpo
                    | sentencia ';'
                    | estructura cuerpo
                    | estructura
                    | RETURN expresion ';'
                    ;
estructura:         WHILE '(' expresion ')' '{'
   cuerpo_estructura '}'
```

```
8                       | IF '(' expresion ')' '{'
                           cuerpo_estructura '}'
9                       | IF '(' expresion ')' '{'
                           cuerpo_estructura '}' ELSE '{'
                           cuerpo_estructura '}'
10                      | FOR '(' declaracion_for ';'
                           expresion ';' asignacion ')' '{'
                           cuerpo_estructura '}'
11                      ;
12
13  declaracion_for:    INTEGER  IDENTIF valor_for
       r_declaracion_for
14                      |      IDENTIF valor_for
                           r_declaracion_for
15                      ;
16  valor_for:
17                      | '=' NUMBER
18                      ;
19  r_declaracion_for:      ',' IDENTIF valor_for
       r_declaracion_for
20                      |
21                      ;
22  cuerpo_estructura:  sentencia ';'
23                      | estructura
24                      | sentencia ';' cuerpo_estructura
25                      | estructura cuerpo_estructura
26                      | RETURN expresion ';'
27                      ;
```

4. Traducción Backend

Una vez acabado con la implementación del frontend que traducía de C a Lisp, vamos a realizar el correspondiente backend, de Lisp a Forth. Para ello, hemos decidido dividir la implementación en las siguientes fases:

4.1. Variables Globales

Tal y como traducimos el lisp todas las variables que pueden haber en este código serán globales. Simplemente cambiará el nombre si son específicas a una función en específico. Como ya se unificaron las formas de declarar variables esta parte de la gramática es relativamente simple.

```
var_global:          declaracion
                     | var_global declaracion
                     ;
declaracion:         '(' SETQ IDENTIF logical_or ')'
                     ;
```

4.2. Función MAIN y Funciones Genéricas

La definición de funciones se puede unificar para que haya un no terminal que acoge ambos dos tipos de posibles funciones el main que sigue siendo obligatorio y las otras que siguen siendo opcionales. Esta es una forma más simple que la utilizada en el Frontend y por la traducción de Lisp no se nos requerirá el uso de argumentos en estas funciones.

```
def_funcs:           def_funcs def_func
                     | def_func
                     | def_funcs llamada_main
                     | llamada_main
                     ;
llamada_main:        '(' MAIN ')'

def_func:            '(' DEFUN MAIN '(' ')' cuerpo ')'
                     | '(' DEFUN IDENTIF '(' ')' cuerpo ')
                       '
                     ;
```

4.3. Impresión de Cadenas

Esta función ya fue muy alterada para el cambio de C a Lisp y aquí la traducción es bastante simple ya que el programa puede necesitar imprimir strings o variables/resultados de operaciones.

```
sentencia:           '(' PRINT STRING ')'
                     | '(' PRINC logical_or ')'
                     | '(' PRINC STRING ')'
                     ...
                     ;
```

4.4. Operadores, Precedencia y Asociatividad

Estos son todos los operadores y comparadores que permiten que se sigan las reglas de asociatividad que creamos en la primera gramática. Esta parte de la gramática es muy similar a la del Frontend ya que el cambio de lenguaje no realiza ningún cambio estructural muy grande. Lo que sí que tuvimos en cuenta fue en cuanto a las operaciones en Lisp, que al llevar paréntesis hicimos que las operaciones tuvieran el mismo nivel de precedencia.

```
1  logical_or:          logical_and
2                       | '(' OR logical_or logical_and ')'
3                       ;
4  logical_and:         igualdad
5                       | '(' AND logical_and igualdad ')'
6                       ;
7  igualdad:            relacional
8                       | '(' '=' igualdad relacional ')'
9                       | '(' NE igualdad relacional ')'
10                      ;
11 relacional:          operacion
12                      | '(' '<' relacional operacion ')'
13                      | '(' '>' relacional operacion ')'
14                      | '(' LE relacional operacion ')'
15                      | '(' GE relacional operacion ')'
16                      ;
17 operacion:           unario
18                      | '(' '+' operacion operacion ')'
19                      | '(' '-' operacion operacion ')'
20                      | '(' '*' operacion operacion ')'
21                      | '(' '/' operacion operacion ')'
22                      | '(' MOD operacion operacion ')'
23                      ;
24 unario:              operando
25                      | '(' NOT unario ')'
26                      | '+' operando %prec UNARY_SIGN
27                      | '(' '-' operando %prec UNARY_SIGN
                           ')'
28                      ;
29 operando:            IDENTIF
30                      | NUMBER
31                      | '(' logical_or ')'
32                      ;
```

4.5. Estructuras de Control

Hemos logrado que en esta gramática las sentencias y estructuras de control estén unificadas gracias a la estructura de lisp y a la forma tan estandarizada de definir el código.

```
sentencia:          '(' PRINT STRING ')'
                    | '(' PRINC logical_or ')'
                    | '(' PRINC STRING ')'
                    | '(' SETF IDENTIF logical_or ')'
                    | '(' SETQ IDENTIF logical_or ')'
                    | '(' LOOP WHILE logical_or DO
                       lista_sentencia ')'
                    | '(' IF logical_or sentencia ')'
                    | '(' IF logical_or sentencia
                       sentencia ')'
                    | '(' PROGN lista_sentencia ')'
                    ;
```

5. Testing

En esta sección se presentan las pruebas realizadas para verificar el correcto funcionamiento del traductor en sus distintas etapas. Cada prueba incluye el código fuente en C, su correspondiente traducción a Lisp y finalmente la traducción a Forth.

5.1. Variables Globales

```
#include <stdio.h>
int a;
int b = 10, c, d = 5;
main() {
    puts("Variables globales");
}
//@ (main)
-------------------------------------------------
Variables globales
-------------------------------------------------
(setq a 0)
(setq b 10)
(setq c 0)
(setq d 5)
```

```
(defun main ()
        (print "Variables globales")
)
 (main)
-----------------------------------------------------
"Variables globales"
-----------------------------------------------------
variable a
0 a !
variable b
10 b !
variable c
0 c !
variable d
5 d !
: main ." Variables globales" ;
main
-----------------------------------------------------
Variables globales
```

5.2. Función MAIN

```
#include <stdio.h>
int x;

main() {
    x = x + 1;
    printf("%d\n", x);
}
//@ (main)
-----------------------------------------------------
1
-----------------------------------------------------
(setq x 0)
(defun main ()
        (setf x (+ x 1))
        (princ x)

)
 (main)
-----------------------------------------------------
1
```

```
----------------------------------------------------
variable x
0 x !
: main x @ 1 + x !
x @ . ;
main
----------------------------------------------------
1
```

5.3. PUTS

```
#include <stdio.h>
int main_var;
main() {
    puts("Hola, mundo!");
}
//@ (main)
----------------------------------------------------
Hola, mundo!
----------------------------------------------------
(setq main_var 0)
(defun main ()
        (print "Hola, mundo!")
)
 (main)
----------------------------------------------------
"Hola, mundo!"
----------------------------------------------------
variable main_var
0 main_var !
: main ." Hola, mundo!" ;
main
----------------------------------------------------
Hola, mundo!
```

5.4. PRINT Simple

```
#include <stdio.h>
int val = 42;
main() {
    printf("%d", val);
```

```
}
//@ (main)
----------------------------------------------------
42
----------------------------------------------------
(setq val 42)
(defun main ()
        (princ val)

)
 (main)
----------------------------------------------------
42
----------------------------------------------------
variable val
42 val !
: main val @ . ;
main
----------------------------------------------------
42
```

5.5. Operadores

```
#include <stdio.h>
int a = 1;
int b = 0;
int c = 2;
main() {
    // mezcla &&, ||, ==, !=, <, >, <=, >=, %, +, -, *, /
    if ((a + c * 3) % 2 == 1 && b != 0 || a <= c) {
        puts("OK");
    }
}
//@ (main)
----------------------------------------------------
OK
----------------------------------------------------
(setq a 1)
(setq b 0)
(setq c 2)
(defun main ()
```

```
        (if (or (and (= (mod (+ a (* c 3)) 2) 1) (/= b 0)
           ) (<= a c))
        (progn  (print "OK")))
)
 (main)
-----------------------------------------------------
"OK"
-----------------------------------------------------
variable a
1 a !
variable b
0 b !
variable c
2 c !
: main a @ c @ 3 * + 2 mod 1 = b @ 0 = 0= and a @ c @ <=
   or if
        ." OK"
then ;
main
-----------------------------------------------------
OK
```

5.6. WHILE

```
#include <stdio.h>
int cnt = 0;
main() {
    while (cnt < 3) {
        printf("%d\n", cnt);
        cnt = cnt + 1;
    }
}
//@ (main)
-----------------------------------------------------
0
1
2
-----------------------------------------------------
(setq cnt 0)
(defun main ()
        (loop while (< cnt 3) do
        (progn  (princ cnt)
```

```
19
20          (setf cnt (+ cnt 1))))
21 )
22  (main)
23 ----------------------------------------------------
24 012
25 ----------------------------------------------------
26 variable cnt
27 0 cnt !
28 : main begin
29          cnt @ 3 <
30 while
31          cnt @ .
32 cnt @ 1 + cnt !
33 repeat ;
34 main
35 ----------------------------------------------------
36 0 1 2
```

5.7. IF-ELSE

```
1  #include <stdio.h>
2  int v = 5;
3  int res;
4  main() {
5      if (v % 2 == 0) {
6          puts("Par");
7      } else {
8          puts("Impar");
9      }
10     if (v > 10) {
11         puts("Grande");
12     }
13 }
14 //@ (main)
15 ----------------------------------------------------
16 Impar
17 ----------------------------------------------------
18 (setq v 5)
19 (setq res 0)
20 (defun main ()
21          (if (= (mod v 2) 0)
```

```
        (progn  (print "Par"))
        (progn  (print "Impar")))
        (if (> v 10)
        (progn  (print "Grande")))
)
 (main)
-----------------------------------------------------
"Impar"
-----------------------------------------------------
variable v
5 v !
variable res
0 res !
: main v @ 2 mod 0 = if
        ." Par"
else
        ." Impar"
then
v @ 10 > if
        ." Grande"
then ;
main
-----------------------------------------------------
Impar
```

5.8. FOR

```
#include <stdio.h>
int i;
int suma = 0;
int n = 5;
main() {

    for (i = 0; i < n; i = i + 1) {
        suma = suma + i;
    }
    printf("%d\n", suma);
}
//@ (main)
-----------------------------------------------------
10
-----------------------------------------------------
```

```
(setq i 0)
(setq suma 0)
(setq n 5)
(defun main ()
        (setq i 0)
        (loop while (< i n) do
        (progn  (setf suma (+ suma i)))
        (setf i (+ i 1)))
        (princ suma)

)
 (main)
-----------------------------------------------------
10
-----------------------------------------------------
variable i
0 i !
variable suma
0 suma !
variable n
5 n !
: main 0 i !
begin
        i @ n @ <
while
        suma @ i @ + suma !
i @ 1 + i !
repeat
suma @ . ;
main
-----------------------------------------------------
10
```

5.9. Funciones (Lisp)

```
#include <stdio.h>
square (int v) {
    return v * v;
    }
sum (int x, int y) { return x + y; }
main() {
    int r1, r2;
```

```
    r1 = square(7);
    r2 = sum(3, 4);
    printf("%d %d\n", r1, r2);
    // llamada como procedimiento (valor ignorado)
    square(5);
}
//@ (main)
-----------------------------------------------------
49 7
-----------------------------------------------------
(defun square (v )
        (* v v)
)
(defun sum (x  y )
        (+ x y)
)

(defun main ()
        (setq r1 0)
        (setq r2 0)
        (setf r1 (square 7 ))
        (setf r2 (sum 3  4 ))
        (princ r1)
        (princ r2)

        (square 5 )
)
 (main)
-----------------------------------------------------
497
```

5.10. Variables Locales (Lisp)

```
#include <stdio.h>
main() {
    int g = 1;
    int a = 4, b;
    a = a + 1;
    b = a * 2;
    printf("%d %d\n", a, b);
}
//@ (main)
```

```
----------------------------------------------------
5 10
----------------------------------------------------
(defun main ()
        (setq g 1)
        (setq a 4)
        (setq b 0)
        (setf a (+ a 1))
        (setf b (* a 2))
        (princ a)
        (princ b)
)
 (main)
----------------------------------------------------
510
```

5.11. Vectores (Lisp)

```
#include <stdio.h>
int arr[5];
main() {
    int i;
    for (i = 0; i < 5; i = i + 1) {
        arr[i] = i * 2;
    }
    printf("%d %d %d %d %d\n",
           arr[0], arr[1], arr[2], arr[3], arr[4]);
}
//@ (main)
----------------------------------------------------
0 2 4 6 8
----------------------------------------------------
(setq arr (make-array 5))
(defun main ()
        (setq i 0)
        (setq i 0)
        (loop while (< i 5) do
        (progn  (setf (aref arr i) (* i 2)))
        (setf i (+ i 1)))
        (princ (aref arr 0))
        (princ (aref arr 1))
        (princ (aref arr 2))
```

```
25          (princ (aref arr 3))
26          (princ (aref arr 4))
27  )
28   (main)
29  ------------------------------------------------------
30  02468
```

6. Conclusiones

La presente memoria ha descrito en detalle el diseño, la implementación y la validación de un traductor de dos fases para un subconjunto de C: un *frontend* que convierte C en Lisp y un *backend* que convierte Lisp en Forth. A lo largo del proyecto se han abordado todos los objetivos formativos propuestos en la asignatura: la definición de gramáticas jerárquicas, la gestión de la semántica mediante código diferido, la generación de código intermedio y final, y la validación con conjuntos de pruebas diseñados ad hoc y suministrados por el profesorado.

El ciclo incremental de desarrollo permitió aislar y corregir errores de forma temprana en cada iteración. Gracias a esta estrategia, pudimos empezar por las declaraciones básicas y la impresión de cadenas, añadir luego operadores y estructuras de control, y finalmente incorporar funciones con retorno y vectores antes de abordar el backend. La generación de los ficheros `trad.y` y `back.y`, así como el conjunto de scripts de prueba automatizados, garantizó la robustez del traductor y facilitó la comparación de los comportamientos nativos en C, Lisp y Forth.

Como principales retos formativos cabe destacar la resolución de conflictos de gramática (shift/reduce y reduce/reduce), la correcta gestión de ámbitos en variables locales versus globales, y la inserción de instrucciones de salida (`bye` en GForth) para evitar bloqueos en los scripts automatizados. Estos desafíos han reforzado el entendimiento de la teoría de compiladores y la importancia de un buen diseño gramatical y semántico.

Para trabajos futuros, sería interesante ampliar el subconjunto de C soportado (por ejemplo, punteros, estructuras y llamadas con múltiples argumentos) o una implementación propio de un analizador léxico en *flex*.

En definitiva, este proyecto no solo cumple los requisitos de la práctica, sino que sienta las bases para desarrollos posteriores en el ámbito de los procesadores de lenguaje y la generación de compiladores de propósito educativo o industrial.