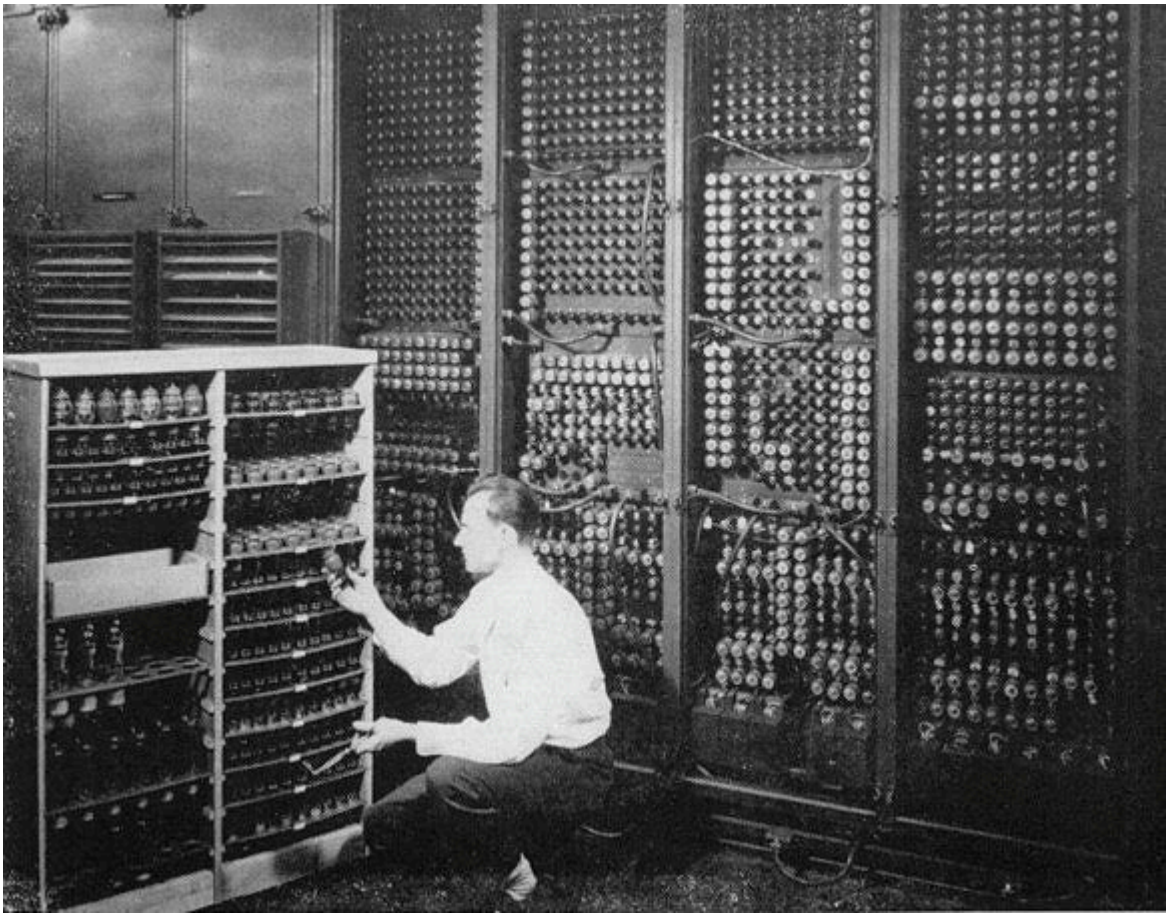
Práctica Final
Procesadores del Lenguaje

Traductor frontend: subconjunto de lenguaje C a LISP
Traductor backend: LISP a notación Postfix



Miguel Jimeno Casas - 100495932 - 100495932@alumnos.uc3m.es

Héctor Herraiz Diéz - 100499734 - 100499734@alumnos.uc3m.es

Grupo 82 - 211 - Ingeniería Informática

Índice

1 Introducción

Breve descripción del objetivo del proyecto

El objetivo de este proyecto es implementar un traductor completo dividido en dos fases: un *frontend* que traduce un subconjunto del lenguaje C a código intermedio en **LISP**, y un *backend* que convierte este código LISP en notación **postfija** (Postfix) ejecutable por un intérprete **Forth**. Con ello, buscamos simular el funcionamiento interno de un compilador simple, abordando el análisis léxico y sintáctico, la traducción estructurada y la generación de código.

Este proyecto nos ha permitido aplicar de forma práctica todos los conceptos vistos durante la asignatura de *Procesadores del Lenguaje*, en especial el uso de **gramáticas en Bison**, la definición de tokens mediante **Flex**, y la generación estructurada de código. Además, el proceso de traducir entre lenguajes tan diferentes como C, LISP y Forth nos ha obligado a reflexionar profundamente sobre la semántica de cada uno y sobre cómo adaptar estructuras imperativas a contextos funcionales o basados en pila.

Alcance del trabajo: qué hace el traductor, qué se ha implementado

Nuestro traductor es capaz de procesar programas escritos en un subconjunto definido del lenguaje C, incluyendo:

- Declaraciones de variables globales y locales (con o sin inicialización).
- Definiciones de funciones, con y sin parámetros.
- Estructuras de control como `if`, `else`, `while` y `for`.
- Instrucciones de impresión (`puts`, `printf`) y expresiones aritméticas y lógicas.
- Declaración y uso de vectores.
- Llamadas a funciones definidas por el usuario.
- Traducción de la directiva especial `//@ (main)` para ejecución.

Todo el código fuente en C se traduce primero a una versión intermedia en **LISP**, que conserva la estructura funcional del programa. A partir de ahí, el backend genera código **Forth**, empleando las primitivas correspondientes para operaciones aritméticas, asignaciones, control de flujo e impresión.

Este trabajo incluye tanto la implementación de los ficheros `trad.y` (frontend) y `back.y` (backend), como un conjunto de pruebas diseñadas por nosotros, una documentación técnica detallada, y las versiones formateadas del código fuente.

---

2. Arquitectura General del Traductor

Descripción global del diseño

“Práctica realizada de manera equitativa por ambos autores”

Frontend (Análisis)

1. **Análisis léxico**: Implementado en `yylex()`, reconoce los tokens del lenguaje fuente (C-like) y los clasifica.
2. **Análisis sintáctico**: Implementado en `yyparse()`, verifica la estructura del programa según la gramática definida.
3. **Análisis semántico**: Incluido en las acciones asociadas a las reglas gramaticales, realiza comprobaciones de tipos y contexto.

Backend (Generación de código)

1. **Generación de código intermedio**: Produce una representación en Lisp-like como paso intermedio.
2. **Generación de código objetivo**: Convierte el código intermedio a código Forth.

Diagrama general del flujo de traducción

```C/C++
[Programa Fuente en C]
       ↓ (Análisis léxico)
[Tokens]
       ↓ (Análisis sintáctico/semántico)
[Árbol Sintáctico Abstracto (AST)]
       ↓ (Generación de código)
[Código Intermedio (Lisp)]
       ↓ (Transformación final)
[Código Forth Objetivo]
```

Herramientas y lenguajes utilizados

1. **Herramientas principales**:
    - Lex/Yacc (Flex/Bison) para el análisis léxico y sintáctico
    - Compilador GCC para la construcción del traductor
2. **Lenguajes**:
    - C: Para la implementación del traductor (infija)
    - Lisp-like: Como representación intermedia (prefija)
    - Forth: Como lenguaje objetivo (postfija)
3. **Estructuras de datos clave**:
    - Tabla de símbolos para manejo de variables
    - Árboles sintácticos para representación intermedia
    - Mecanismos de gestión de memoria para generación de código
4. **Características notables**:
    - Manejo de ámbitos para variables locales/globales
    - Traducción de estructuras de control (if, while, for)
    - Soporte para arreglos y llamadas a funciones
    - Sistema de tipos básico

El traductor sigue un enfoque clásico de compilación por fases, con una clara separación entre el análisis del código fuente y la generación del código objetivo, permitiendo potenciales mejoras y extensiones futuras.

---

3. Desarrollo del Frontend (Traducción de C a LISP)

El frontend de nuestro traductor es el encargado de realizar la conversión desde un subconjunto del lenguaje C hacia una representación intermedia en **LISP**. Para ello, hemos dividido su funcionamiento en dos componentes principales: el **análisis léxico**, que identifica las unidades básicas del lenguaje de entrada (tokens), y el **análisis sintáctico y semántico**, que construye una estructura jerárquica del programa y genera el código intermedio correspondiente.

3.1. Análisis Léxico

El análisis léxico ha sido implementado mediante una función `yylex()` predefinida y proporcionada por los profesores. Esta función no ha sido modificada, siguiendo las instrucciones del enunciado. Se encarga de reconocer identificadores, números, palabras reservadas, cadenas de texto y operadores, devolviendo los tokens correspondientes al parser. Además, incluye un manejo especial de comentarios con `//` y directivas `//@`, que permiten insertar código embebido en la salida. A esto último recurrimos en las pruebas, ya que es el encargado de traducir la ejecución del main.

Cabe destacar que el lexer incluye una tabla de palabras clave (`keywords[]`), distingue correctamente entre variables y palabras reservadas, y aplica reglas de precedencia para operadores necesarios durante la práctica. Esta lista está compuesta por operadores que se especificarán más adelante como `==`, `!=`, `<=`, `>=`, `&&`, `||`, etc.

3.2. Gramática (Análisis Sintáctico y Análisis Semántico)

Nuestro enfoque ha consistido en construir la gramática de forma modular, tratando cada componente del lenguaje por separado y utilizando funciones auxiliares para gestionar variables locales, distinguir entre variables globales y locales, y dar formato a la salida en LISP. Esto nos ha permitido asegurar una traducción precisa y coherente, manteniendo la semántica de los programas originales en C.

A continuación, detallamos cómo hemos implementado cada una de las construcciones principales exigidas en el enunciado de la práctica:

- **Estructura general del programa:** división entre declaraciones globales y funciones.
- **Definición de funciones (con y sin parámetros)**, incluyendo `main`.
- **Sentencias y bloques de instrucciones**, como asignaciones, llamadas, `return`, y control de flujo (`if`, `while`, `for`).
- **Declaración y uso de variables** (globales, locales y vectores).
- **Expresiones aritméticas, lógicas y comparativas.**
- **Traducción de llamadas a `puts`, `printf`, y expresiones inline.**

Cada una de estas secciones será explicada en los siguientes apartados, incluyendo ejemplos concretos del código generado.

3.2.1. Variables Globales

La gramática contempla la declaración de variables globales con o sin inicialización, así como declaraciones múltiples. Todas se traducen a expresiones `setq` en LISP. Estas producciones se encuentran bajo el no terminal `lista_declaracion_global`, que es parte de `declaracion_variable_global`, incluida en `declaracion_variables_globales`.

Gestión en la gramática

a) Declaración sin inicialización o con inicialización constante

```
C/C++
lista_declaracion_global:
    IDENTIF {
        sprintf(temp, "(setq %s 0)", $1.code);
        $$.code = gen_code(temp);
    }
| IDENTIF '=' NUMBER {
    sprintf(temp, "(setq %s %d)", $1.code, $3.value);
    $$.code = gen_code(temp);
}
```

Código C: `int x; int y=5;`
Código Lisp: `(setq x 0) (setq y 5)`

b) Declaración múltiple (con o sin inicialización)

```
C/C++
| lista_declaracion_global ',' IDENTIF {
    sprintf(temp, "%s\n(setq %s 0)", $1.code, $3.code);
    $$.code = gen_code(temp);
}
| lista_declaracion_global ',' IDENTIF '=' NUMBER {
    sprintf(temp, "%s\n(setq %s %d)", $1.code, $3.code, $5.value);
    $$.code = gen_code(temp);}
```

Traducción resultante

Código C: `int a = 1, b, c = 3;`
Código Lisp: `(setq a 1) (setq b 0) (setq c 3)`

Estas reglas aseguran que las declaraciones múltiples se procesan en orden y que cualquier identificador sin valor asignado reciba por defecto un `0`. Como exige el enunciado, **no se permiten expresiones evaluables** en las asignaciones globales, sólo constantes (`NUMBER`), respetando la semántica del C compilado.

3.2.2. Función `main`

La función principal `main()` se reconoce en la gramática con el token reservado `MAIN`, y se traduce en Lisp mediante una definición `defun main ()`. Solo puede haber una función `main` y debe aparecer tras la declaración de variables globales. Su contenido puede incluir cualquier instrucción permitida en el lenguaje.

Gestión en la gramática

La producción encargada de detectar y traducir la función `main` es:

```
C/C++
funcion_main:
    inicializacion_main '{' instrucciones '}' {
        sprintf(temp, "(%s %s\n)", $1.code, $3.code);
        $$.code = gen_code(temp);
        clear_local_vars();  // limpia las variables locales al salir
    };
```

```
C/C++
inicializacion_main:
    MAIN '(' ')' {
        current_function = strdup("main");
        sprintf(temp, "defun main ()\n");
        $$.code = gen_code(temp);
    };
```

Traducción resultante

Código C: `int a; main() {puts(“Hola mundo”) printf(“%s”, a)}`

Código Lisp: `(setq a 0) (defun main () (print (“Hola mundo”)) (princ a))`

3.2.3. Impresión de cadenas con `puts`

La función `puts` permite imprimir cadenas literales encerradas entre comillas dobles en C. La gramática traduce directamente esta sentencia al formato Lisp equivalente `(print <string>)`.

Gestión en la gramática

Se reconoce esta estructura dentro del no terminal `sentencia_simple`:

```
C/C++
sentencia_simple:
```

```
...
| PUTS '(' STRING ')' {
    sprintf(temp, "(print \"%s\")", $3.code);
    $$.code = gen_code(temp);
}
```

La palabra clave `puts` está incluida en la tabla de palabras reservadas como:

`{ "puts", PUTS }`

Y el **análisis léxico** (función `yylex`) reconoce correctamente los `STRING` al detectar cadenas entre comillas dobles y devolver el token `STRING`:

Traducción resultante

Entrada en C: `puts("Hola mundo");`

Código Lisp: `(print "Hola mundo")`

Esta conversión es directa y sencilla, ya que no requiere interpretar formato ni procesar variables. Se utiliza `print` en lugar de `princ` para asegurar el salto de línea al final, replicando el comportamiento por defecto de `puts`.

3.2.4. Impresión de expresiones y cadenas con `printf`

La sentencia `printf(<string>, <elem1>, …, <elemN>);` permite imprimir combinaciones de cadenas y expresiones. En esta implementación:

- La **cadena de formato** se **ignora** en la traducción a Lisp.
- Cada `<elem>` se convierte en una instrucción independiente `(princ <elem>)`.
- Se preserva el orden original de los elementos.

Gestión en la gramática

Dentro de `sentencia_simple`, se define:

```
C/C++
sentencia_simple:
    ...
    | PRINTF '(' STRING ',' lista_elementos ')' {
        $$.code = $5.code;
    }
```

Esto asegura que la cadena de formato (`STRING`) se reconozca, pero no se utilice en la salida.

Construcción de los elementos

La lista de elementos se define en:

```
C/C++
lista_elementos:

    elemento { $$.code = $1.code; }

    | lista_elementos ',' elemento {

        sprintf(temp, "%s\n%s", $1.code, $3.code);

        $$.code = gen_code(temp);

    }
```

Cada elemento puede ser una expresión o cadena:

```
C/C++
elemento:
    expresion {
        sprintf(temp, "(princ %s)", $1.code);
        $$.code = gen_code(temp);
    }
    | STRING {
        sprintf(temp, "(princ \"%s\")", $1.code);
        $$.code = gen_code(temp);
    }
```

Traducción resultante

Código C: `printf("%d %s", a + 1, "hola");`

Código Lisp: `(princ (+ a 1)) (princ "hola")`

Este enfoque evita el procesamiento de la cadena de formato y se centra únicamente en imprimir los elementos que la acompañan.

3.2.5. Operadores, precedencia y asociatividad

Los operadores aritméticos, lógicos y de comparación de C se han adaptado al entorno de Lisp respetando la precedencia y la asociatividad mediante directivas `%left`, `%right` y `%prec` definidas en Bison. La gramática permite combinarlos libremente en expresiones, tal como ocurre en C.

Correspondencias entre operadores:

C	&&	||	!	!=	==	<	<=	>	>=	%
Lisp	and	or	not	/=	=	<	<=	>	>=	mod

Gestión en la gramática

La regla `expresion` define todas las combinaciones posibles. Por ejemplo, para la comparación de igualdad:

```
C/C++
expresion:
    ...
    | expresion IGUAL expresion {
        sprintf(temp, "(= %s %s)", $1.code, $3.code);
        $$.code = gen_code(temp);
    }
```

Y para el operador lógico AND:

```
C/C++
| expresion AND expresion {
    sprintf(temp, "(and %s %s)", $1.code, $3.code);
    $$.code = gen_code(temp);
}
```

Traducción resultante

Código C:: `a != 3 && b < 5`

Código Lisp: `(and (/= a 3) (< b 5))`

El resto de operadores siguen una estructura similar.

3.2.6. Estructura de control `while`

La estructura de control `while` de C ha sido adaptada directamente a la forma equivalente en Lisp utilizando la macro `(loop while (...) do (...))`. En C, este tipo de bucle no requiere `;` al final, lo cual también se refleja en la gramática.

Gestión en la gramática

La estructura `while` está contenida en el no terminal `sentencia_bloque`:

```
C/C++
sentencia_bloque:

    ...

    | WHILE '(' expresion ')' '{' instrucciones '}' {

        sprintf(temp, "(loop while %s do\n%s)", $3.code, $6.code);

        $$.code = gen_code(temp);

    }
```

Esto permite que cualquier código dentro del bloque se repita mientras se cumpla la condición `expresion`.

Traducción resultante

Código C: `while (a != 10) { printf("%d", a); a = a + 1;}`

Código Lisp: `(loop while (/= a 10) do (princ a))`

3.2.7. Estructura de control `if`

La estructura de control `if` en C se traduce a Lisp usando la forma `(if <condición> <then> [<else>])`. Esta conversión se adapta de acuerdo con las siguientes reglas:

- Si hay una sola sentencia en `then` o `else`, se incluye directamente.
- Si hay múltiples sentencias, se agrupan con `(progn ...)`.

Conflictos en Bison

La inclusión de `else` puede generar un **conflicto de ambigüedad "dangling else"** en el parser, es decir, que no se sepa a qué `if` pertenece el `else` en caso de anidamiento. Este conflicto se resuelve asegurando que el `else` siempre se asocie al `if` más cercano y que se usen siempre llaves `{}` en ambas ramas del `if`.

Gestión en la gramática

Ya están integradas en el no terminal `sentencia_bloque`:

```
C/C++
if_sin_else:
    IF '(' expresion ')' '{' instrucciones '}' {
        if (strchr($6.code, '\n') != NULL)
            sprintf(temp, "(if %s\n(progn %s))", $3.code, $6.code);
        else
            sprintf(temp, "(if %s\n%s)", $3.code, $6.code);
        $$.code = gen_code(temp);
    }
;
```

```
C/C++
if_con_else:
    IF '(' expresion ')' '{' instrucciones '}' ELSE '{' instrucciones '}' {
        if (strchr($6.code, '\n') != NULL && strchr($10.code, '\n') != NULL)
                sprintf(temp, "(if %s\n(progn\n%s)\n(progn\n%s))", $3.code,
$6.code, $10.code);
        else if (strchr($6.code, '\n') != NULL)
                sprintf(temp, "(if %s\n(progn\n%s)\n%s)", $3.code, $6.code,
$10.code);
        else if (strchr($10.code, '\n') != NULL)
                sprintf(temp, "(if %s\n%s\n(progn\n%s))", $3.code, $6.code,
$10.code);
        else
            sprintf(temp, "(if %s\n%s\n%s)", $3.code, $6.code, $10.code);
        $$.code = gen_code(temp);
    }
;
```

Estas reglas cubren los casos con y sin `else`, y usan `progn` automáticamente si detectan múltiples sentencias.

Traducción resultante

Código C:: `if (a == 0) { b = 1;} else { b = 2;}`

Código Lisp: `(if (= a 0) (setq b 1)(setq b 2))`

Si hay varias instrucciones en una rama:

Código C:: `if (a == 0) { puts("hola"); b = 1;}`

Código Lisp: `(if (= a 0)(progn (print "hola")(setq b 1)))`

3.2.8. Estructura de control `for`

El bucle `for` en C sigue una estructura tripartita:

```
C/C++
for (<inicialización>; <condición>; <incremento>) {
    <código>
}
```

Y debe traducirse en Lisp de la siguiente forma:

```
C/C++
<inicialización>
(loop while <condición> do
  <código>
  <incremento>)
```

Gestión en la gramática

```
C/C++
sentencia_bloque:
…
| FOR '(' inicializacion ';' expresion ';' incremento ')' '{' instrucciones
'}' {
```

```
    sprintf(temp, "%s\n(loop while %s do\n%s\n%s)",
            $3.code, $5.code, $10.code, $7.code);
    $$.code = gen_code(temp);
}
```

```
C/C++
inicializacion:
    INTEGER IDENTIF '=' operando {
       sprintf(temp, "(setq %s_%s %s)", current_function, $1.code, $3.code);
        $$.code = gen_code(temp);
    }
    | IDENTIF '=' operando {
              sprintf(temp, "(setf %s_%s %s)", current_function, $1.code,
$3.code);
        $$.code = gen_code(temp);
    }
;
```

```
C/C++
incremento:
    IDENTIF '=' expresion {
        sprintf(temp, "(setf %s_%s %s)", current_function, $1.code, $3.code);
        $$.code = gen_code(temp);
    }
;
```

Traducción resultante

Código C:: `for (a = 0; a < n; a = a + 2) {puts("Iteración");}`

Código Lisp: `(setq a 0) (loop while (< a n) do (print "Iteración")(setf a (+ a 2)))`

3.2.9. Variables globales

En C, dentro del cuerpo de las funciones se pueden declarar variables locales. Estas deben tratarse de manera diferente a las variables globales para evitar colisiones. En Lisp, toda variable definida con `setq` es global, por lo que para simular variables locales, se emplea una convención basada en **renombrar** las variables concatenando el nombre de la función

con el de la variable. Por ejemplo, la variable `a` dentro de la función `main` se convierte en `main_a`.

Uso del nombre prefijado

Para distinguir las variables locales de las globales, el analizador mantiene una **tabla de variables locales**. Al declarar una variable local, se añade a esta tabla. Posteriormente, cada vez que una variable es usada o modificada, se consulta esta tabla. Si la variable está en ella, se genera el nombre `funcion_variable` en la traducción; si no, se trata como global.

Gestión en la gramática

La gramática ya implementa este comportamiento. A continuación, se destacan las partes más relevantes:

En la declaración de variables locales (dentro de funciones), se utiliza la regla lista_declaracion, que incluye:

```
C/C++
IDENTIF {
    add_local_var($1.code);
    sprintf(temp, "(setq %s_%s 0)", current_function, $1.code);
    $$.code = gen_code(temp);
}
```

- Se registra la variable en la tabla y se traduce con `setq` y nombre compuesto.

En las asignaciones dentro de funciones:

```
C/C++
sentencia_simple:

IDENTIF '=' expresion {
    if(is_local_var($1.code)) {
              sprintf(temp,  "(setf  %s_%s  %s)",  current_function,  $1.code,
$3.code);
    } else {
        sprintf(temp, "(setf %s %s)", $1.code, $3.code); }
    $$.code = gen_code(temp);
}
```

Se comprueba si es una variable local y se elige el nombre adecuado para `setf`.

En las expresiones donde se usa una variable:

```
C/C++
IDENTIF {
    if(current_function != NULL && is_local_var($1.code)) {
        sprintf(temp, "%s_%s", current_function, $1.code);
    } else {
        sprintf(temp, "%s", $1.code);
    }
    $$.code = gen_code(temp);
}
```

- Esta lógica asegura que incluso en operaciones o condiciones se respete el ámbito local/global.

Se han realizado y se van a realizar diversos ejemplos sobre esta implementación durante la memoria.

Este enfoque garantiza que las variables locales sean únicas en su contexto y evita interferencias con otras funciones o variables globales.

3.2.10. Funciones

En este apartado se debe implementar la definición y uso de funciones en C traducidas a Lisp, comenzando por una versión simplificada sin tipo de retorno explícito, sin parámetros y sin necesidad de devolver un valor (funciones usadas como procedimientos). Posteriormente, se amplía para permitir funciones con uno o varios argumentos, manteniendo el orden en su uso y definición. En cuanto al valor de retorno, si la sentencia `return` aparece como la última instrucción de la función, se traduce simplemente como el valor devuelto al final del cuerpo de la función; en cualquier otro caso, se traduce como `(return-from <nombre-función> <expresión>)`. También se contempla que una función pueda llamarse sin usar su resultado, es decir, como una sentencia independiente. Se evita incluir el tipo en la definición para prevenir conflictos gramaticales, y se mantiene el seguimiento del nombre de la función actual para poder gestionar correctamente los retornos.

Gestión en la gramática

Vamos a dividir en partes este apartado:

1. Cabecera de la función

Define el nombre y parámetros con `defun`:

```
C/C++
inicializacion_funcion:
    IDENTIF '(' parametros ')' {
        current_function = $1.code;
        sprintf(temp, "defun %s (%s)\n", $1.code, $3.code);
        $$.code = gen_code(temp);
    }
```

2. Cuerpo y cierre de función

Traduce el cuerpo, detectando si el `return` está al final:

```
C/C++
funcion:
    inicializacion_funcion '{' instrucciones '}' {
        char *last_instr = strrchr($3.code, '\n');
        if (last_instr != NULL) last_instr++;
        else last_instr = $3.code;

        if (strstr(last_instr, "return") != NULL) {
            sprintf(temp, "(%s %s)", $1.code, $3.code);
        } else {
            sprintf(temp, "(%s %s\n)", $1.code, $3.code);
        }

        $$.code = gen_code(temp);
        free(current_function);
    }
```

3. Retorno explícito dentro de funciones:

```
C/C++
sentencia_simple:
...
    RETURN expresion {
        if (current_function != NULL) {
            if (is_local_var($2.code)) {
                   sprintf(temp, "(return-from %s %s_%s)", current_function,
current_function, $2.code);
```

```
            } else {
                    sprintf(temp, "(return-from %s %s)", current_function,
$2.code);
            }
        }
        $$.code = gen_code(temp);
    }
```

Traducción resultante

Código C:: `int suma(int x, int y) { return x + y;}`

Código Lisp: `(defun suma (x y)(+ x y))`

3.2.11. Implementación de Vectores

Este apartado permite declarar y utilizar vectores tanto globales como locales en C, traducidos a estructuras `make-array` en Lisp. Se gestiona su creación, acceso y modificación según el ámbito (global/local).

Gestión en la gramática

Manejamos primero la declaración:

```
C/C++
lista_declaracion_global:

IDENTIF '[' NUMBER ']' {
  sprintf(temp, "(setq %s (make-array %d))", $1.code, $3.value);
  $$.code = gen_code(temp);
}
```

Haríamos lo mismo en la lista de declaraciones local, pero añadiendo el prefijo.

Acceso a elementos del vector (usado como operando):

```
C/C++
operando:
IDENTIF '[' expresion ']' {
  if(current_function != NULL && is_local_var($1.code)) {
    sprintf(temp, "(aref %s_%s %s)", current_function, $1.code, $3.code);
  } else {
    sprintf(temp, "(aref %s %s)", $1.code, $3.code);
  }
  $$.code = gen_code(temp);
}
```

Asignación a elementos del vector:

```
C/C++
sentencia_simple:
IDENTIF '[' expresion ']' '=' expresion {
  if(is_local_var($1.code)) {
      sprintf(temp, "(setf (aref %s_%s %s) %s)", current_function, $1.code,
$3.code, $6.code);
  } else {
    sprintf(temp, "(setf (aref %s %s) %s)", $1.code, $3.code, $6.code);
  }
  $$.code = gen_code(temp);
}
```

Traducción resultante

Código C:: `int v[3]; main() { v[0] = 10; printf("%d", v[0]);}`

Código Lisp: `(setq v (make-array 3)) (defun main () (setf (aref v 0)10) (princ (aref v 0))) (main)`

---

4. Desarrollo del Backend (Traducción de LISP a FORTH)

Introducción

El backend de nuestro traductor tiene como objetivo transformar código intermedio en LISP (generado a partir de un subconjunto del lenguaje C) a código en lenguaje FORTH, una

notación de tipo postfijo usada frecuentemente en sistemas embebidos y entornos con arquitecturas de pila. Esta traducción permite ejecutar programas originalmente escritos en C en un entorno que utiliza una máquina de pila como modelo de computación, como es el caso de Gforth.

La principal dificultad del backend reside en la conversión de expresiones prefijas, propias de LISP, a notación postfija, requerida por FORTH. Además, se han de respetar las convenciones de impresión, declaración de variables, definición de funciones y estructuras de control que FORTH impone.

Este backend está implementado usando Bison, definiendo una gramática que reconoce construcciones LISP y emite su equivalente FORTH a medida que se analizan.

Estructura General

La gramática del backend comienza reconociendo programas como una secuencia de expresiones LISP. Cada una de estas expresiones se analiza y se traduce inmediatamente a su equivalente en FORTH. El axioma del parser no produce salida directa; el código se imprime conforme se reconocen definiciones y expresiones, siguiendo una aproximación inmediata (traducción directa).

Acceso y Asignación de Variables

Este apartado del backend traduce las instrucciones de asignación de valores y el acceso a variables escritas en LISP a su correspondiente notación en FORTH. En FORTH, el acceso y almacenamiento se hace explícito con los operadores @ (lectura) y ! (escritura).

Gestión en la gramática

Existen dos formas principales de asignación en LISP que se traducen en la gramática:

Asignación con `setq`:
Se utiliza para inicializar una variable (equivalente a una declaración con asignación en C). En FORTH, se traduce como:

```
variable x

valor x !
```

Esto primero declara la variable y luego le asigna el valor.

Implementado en la gramática:

```
C/C++

sentencia_simple:

    SETQ IDENTIF expresion_gen {

        sprintf(temp, "variable %s\n%s %s !\n", $2.code, $3.code, $2.code);

        $$.code = gen_code(temp);

    }
```

Asignación con `setf:`
Consiste en hacer lo mismo pero quitando la parte de la declaración.

```
C/C++

sentencia_simple:

    SETF IDENTIF expresion_gen {

        sprintf(temp, "%s %s !\n", $3.code, $2.code);

        $$.code = gen_code(temp);

    }
```

Acceso a variables (uso en expresiones):
Cuando una variable aparece como operando, se debe recuperar su valor con `@`. Por ejemplo, `x` se convierte en `x @ .`

```
C/C++

operando: IDENTIF {

    sprintf(temp, "%s @", $1.code);

    $$.code = gen_code(temp);

}
```

Traducción resultante

Código LISP: `(setq x 5) (setf x (+ x 1)) (princ x)`

Traducción a FORTH: `variable x | 5 x ! | x @ 1 + x ! | x @ .`

(Obviar los '|', son para que se entienda mejor)

Funciones (MAIN)

Aunque la implementación de funciones no era obligatoria en el backend, se ha decidido incluir una versión simplificada que permite reconocer y traducir definiciones básicas de funciones, en concreto, la función principal `main`. Esto permite estructurar el código generado de forma más ordenada y modular, acercándolo al estilo de programación imperativo de FORTH, donde se permite definir nuevas palabras (funciones) mediante la sintaxis `: nombre ... ;`.

Gestión en la gramática

Se ha añadido una producción que reconoce la forma básica de definición de una función `main` en LISP `(defun main () <bloque>)` y la traduce a la sintaxis de FORTH utilizando la notación `: main ... ;`.

Regla relevante:

```
C/C++
sentencia_bloque:
    DEFUN MAIN '(' ')' bloque_codigo {
        sprintf(temp, ": main \n %s ;", $5.code);
        $$.code = gen_code(temp);
    }
```

Esta producción traduce el cuerpo de la función `main` al contenido de un bloque FORTH, encerrado entre `: main` y `;`, como es habitual para la definición de nuevas palabras en dicho lenguaje.

Traducción resultante

Código LISP: `(defun main () (setq x 1) (setf x (+ x 2)) (princ x))`

Traducción a FORTH: `: main | variable x | 1 x ! | x @ 2 + x !| x @ .| ;`

(Obviar los '|', son para que se entienda mejor)

Expresiones Aritméticas y Comparativas

Las expresiones aritméticas de LISP, pasan a traducirse a orden inverso (postfijo), como ya hemos vistos en ejemplos mostrados anteriormente.
Los operadores se mantienen respecto a lo anterior, pero el orden se altera.

Gestión en la gramática

Esta parte se maneja al completo en el no terminal expresión, el cuál reconoce todas las entradas de LISP y las ordena a postfija. Por poner un par de ejemplos:

Regla relevante:

```
C/C++
expresion:
...
| '/' expresion_gen expresion_gen {
            sprintf(temp, "%s %s /", $2.code, $3.code);
            $$.code = gen_code(temp);
            }
 | '>' expresion_gen expresion_gen {
            sprintf(temp, "%s %s >", $2.code, $3.code);
            $$.code = gen_code(temp);
            }
```

Traducción resultante

Código LISP: `(setq x (+ 2 3))`

Traducción a FORTH: `: variable x | 2 3 + x !`

(Obviar los ‘|’, son para que se entienda mejor)

Estructuras de control

Las estructuras de control que hemos tenido que manejar en esta parte del trabajo, son tanto el `if-else`, como el `while`.

IF-ELSE:

Debido a que en LISP, la salida del if depende de lo que lleve este dentro, hemos tenido que manejar muchos casos diferentes que se ven reflejados en la gramática que se proporciona a continuación.

Dentro de estos casos están: Que solo tenga if y con una instrucción, solo if con varias instrucciones (PROGN), if con else con varias instrucciones…

Gestión en la gramática

Dentro de la sentencia de bloque, tenemos el if_general, que contempla todos los casos:

```
C/C++
if_general:

       IF expresion_gen possible_progn_if possible_progn_else
                          {  if (strlen($4.code) > 0) {
                          //  Si  el  segundo  possible_progn  tiene  código
                          sprintf(temp,  "%s  IF  \n%s  ELSE\n%s  THEN  ",
                          $2.code,            $3.code,            $4.code);
                          }
                          else {  // Si el segundo possible_progn está vacío
                          sprintf(temp,  "%s  IF  \n%s\n  THEN",  $2.code,
                          $3.code);
                          $$.code = gen_code(temp);}
                  ;
```

Una vez manejado si es if o if-else, pasamos a manejar el PROGN

```
C/C++
possible_progn_if:
                     '('  PROGN  bloque_codigo  ')'  {sprintf  (temp,  "%s",
                     $3.code);
                     $$.code = gen_code(temp);}
                     | '(' sentencia ')'       {sprintf (temp, "%s",$2.code);
                                               $$.code = gen_code(temp); }
                     ;


possible_progn_else:
                           '('  PROGN  bloque_codigo  ')'  {sprintf  (temp,
                           "%s",$3.code);
                             $$.code = gen_code(temp);}
                     | '(' sentencia ')'       {sprintf (temp, "%s",$2.code);
                                               $$.code = gen_code(temp); }
                     | /*vacio*/               { $$.code = gen_code("");
                                                   }
                     ;
```

Traducción resultante

Código LISP: `(if (= c 1)(progn (princ a)(princ " ")))`

Traducción a FORTH: `: c@ 1 = IF | a @ . | ." " | THEN a @ 1 + a ! | REPEAT`

WHILE:

Igual que con el if-else había que controlar muchas cosas diferentes, este fue mucho menos costoso, ya que siempre llegaba igual. `LOOP WHILE` <expresion_gen> `DO` <bloque_codigo>
Bastó con traducirlo siguiendo las consideraciones del pdf.

Gestión en la gramática

Creamos otro tipo de sentencia de bloque:

```
C/C++
sentencia_bloque:
...
| LOOP WHILE expresion_gen DO bloque_codigo {sprintf (temp, "BEGIN %s WHILE
\n %s REPEAT\n", $3.code, $5.code);

$$.code = gen_code(temp);

}
```

Traducción resultante

Código LISP: `(loop while (< a n) do (setf a (+ a 2)))`

Traducción a FORTH: `: BEGIN | a n < | WHILE | a @ 2 + a ! | REPEAT`

(Obviar los '|', son para que se entienda mejor)

Impresión

Como se ha ido viendo a lo largo de los ejemplos puestos durante la memoria del backend, la impresión no se realiza mediante palabras como print o princ. En este caso, se sabe que se quiere imprimir algo, ya que se encuentra un `'.'`.
Depende de la localización del punto la impresión será de una manera o de otra.
Si se recibe un `STRING`, el punto irá anterior a lo que queramos imprimir, mientras que si se recibe otro tipo de impresión, el punto irá posterior.

Gestión en la gramática

Dentro de sentencia_simple, añadimos:

```
C/C++
sentencia_simple:

|  PRINT STRING { sprintf (temp, ".\"%s\"\n", $2.code);
                                          $$.code = gen_code(temp);}
|  PRINC expresion_gen  { sprintf (temp, "%s .\n", $2.code);
                                          $$.code = gen_code(temp);}
| PRINC STRING          { sprintf(temp, ".\"%s\"\n", $2.code);
                                          $$.code = gen_code(temp); }
```

Traducción resultante

Código LISP: `(princ a)(princ " ")`

Traducción a FORTH: `: a @ . | ." "`

(Obviar los ‘|’, son para que se entienda mejor)

Llamada a main

Como se especifica en el enunciado, al backend, hemos decidido que le llegue (main) y que este se encargue de convertirlo en `main`, para ejecutar el programa correctamente.

---

5. Gestión de Errores

Tipos de errores manejados

Errores léxicos:

- **Símbolos no reconocidos**: Caracteres que no pertenecen al alfabeto del lenguaje
- **Strings demasiado largos**: Límite de 255 caracteres
- **Comentarios no cerrados**: Detectados por el analizador léxico

Errores sintácticos:

- **Estructuras mal formadas**: Paréntesis no balanceados, puntos y coma faltantes
- **Órdenes incorrectas**: Expresiones mal construidas según la gramática
- **Conflictos shift/reduce y reduce/reduce**: Identificados durante el desarrollo de la gramática

Errores semánticos (limitados):

- **Variables no declaradas**: En algunos contextos mediante la tabla de símbolos
- **Tipos incompatibles**: En operaciones aritméticas y comparaciones

Estrategias de recuperación

1. **Modo pánico**: En errores graves, se salta hasta el siguiente punto de sincronización (normalmente ';' o '}')
2. **Inserción de tokens faltantes**: En algunos casos se asume la presencia de tokens esenciales
3. **Reporte múltiple**: Se intenta continuar el análisis para detectar más errores después del primero

Conflictos shift/reduce y reduce/reduce

Durante el desarrollo del analizador sintáctico, identificamos varios conflictos:

Conflictos shift/reduce:

1. **En expresiones condicionales**: Entre la regla del `if` simple y el `if-else`
    - Solución: Especificamos precedencia y reorganizamos la gramática
2. **En operadores aritméticos**: Entre operadores de misma precedencia
    - Solución: Añadimos reglas de asociatividad (`%left`, `%right`)

Conflictos reduce/reduce:

1. **En declaraciones de variables**: Entre declaraciones simples y con inicialización
    - Solución: Unificamos las producciones gramaticales
2. **En listas de parámetros**: Entre listas vacías y con elementos
    - Solución: Simplificamos la estructura gramatical

Identificación con -Wcounterexamples:

Utilizamos la opción `-Wcounterexamples` de Bison para:

1. Generar casos concretos que provocan los conflictos
2. Analizar los estados del autómata donde ocurrían
3. Probar soluciones alternativas hasta eliminar los conflictos

---

6. Pruebas Diseñadas (Anexo)

Se han realizado múltiples pruebas para probar tanto el código del trad, como el del back. Las pruebas están en formato C todas, ya que para las pruebas del back, la idea es primero pasarlas por el trad y encadenar directamente esa salida.
Cada prueba cubre diferentes ámbitos para los que el código debe de estar capacitado.

Pruebas Trad

Formato ejecución

./trad < nombre_prueba.c' para ver el código Lisp.

./trad < 'nombre_prueba.c' | clisp para ver la salida de CLisp.

Conjunto de pruebas

- **mcd.c:** función con if y else
- **potencia.c:** función con if y else
- **suma_n.c:** función con bucle for
- **suma_recursiva.c:** función con if y else
- **tabla_multiplicar.c:** función con doble bucle for
- **factorial.c:** función con while
- **posneg.c:** función con 3 if
- **conjunto_funciones.c:** implementación de varias funciones simultáneas

Pruebas Back

Formato ejecución

./trad < nombre_prueba.c' | ./back para ver el código Forth.

./trad < 'nombre_prueba.c' | ./back | gforth para ver la salida de GForth.

Conjunto de pruebas

- **potencias 2.c:** bucle while
- **primos.c:** bucle while dentro de otro bucle while
- **posneg.c:** conjunto de if y else anidados
- **suma_acumulada.c:** bucle while

---

7. Conclusiones

Durante el desarrollo de este proyecto, aprendimos a construir un traductor de lenguaje C a ensamblador paso a paso, lo cual nos permitió comprender más profundamente cómo se representan las instrucciones de alto nivel en un entorno de bajo nivel. Uno de los principales aprendizajes fue cómo analizar y descomponer estructuras sintácticas comunes en C, como funciones, condicionales y bucles, y mapearlas adecuadamente a instrucciones ensamblador.

Entre los retos más importantes que enfrentamos estuvieron el manejo correcto de los registros y la pila, así como asegurar que las variables locales y los flujos de control (como if y while) se tradujeran de forma precisa y sin errores lógicos. También tuvimos que tener mucho cuidado al manejar etiquetas únicas para saltos, especialmente al anidar estructuras de control.

En retrospectiva, podríamos mejorar la modularidad del traductor, facilitando la incorporación de nuevas estructuras. Asimismo, podríamos optimizar la generación de código ensamblador para que sea más eficiente, en lugar de simplemente correcto.

En el futuro, nos gustaría extender este traductor para soportar más tipos de estructuras del lenguaje C, como `switch`, `struct`, arreglos y punteros, lo cual implicaría desafíos adicionales tanto en el análisis sintáctico como en la generación de código.