Universidad Nacional de Quilmes

Parseo y generación de código

Trabajo Práctico
Intérprete reducido de Eiffel

B. Emmanuel Pericon

Diciembre 3, 2025

Indice

1 Introducción **3**

2 Diseño del analizador léxico **3**

3 Diseño del analizador sintáctico **4**

4 Estructura del AST **6**

5 Gestión de entornos y tabla de símbolos **7**

6 Intérprete **8**

7 Decisiones de diseño y limitaciones **9**

8 Compilación y ejecución **10**

9 Conclusion **10**

10 Reseña del lenguaje Eiffel y su origen **11**

11 Re-entrega **12**
11.1 Comentarios 12
11.2 Concatenación de strings 12
11.3 Multiples lineas de declaración de variables 13
11.4 Números negativos 13
11.5 Conclusión de re-entrega 14

1 Introducción

A partir del analizador léxico entregado en la parte 1, para esta entrega se optó por la Opción A (Intérprete directo), utilizando Flex para el análisis léxico, Bison para el análisis sintáctico y un entorno en C para la construcción y evaluación del Árbol de Sintaxis Abstracta (AST).

2 Diseño del analizador léxico

El analizador con Flex (**lexer.l**) se describe mediante expresiones regulares, que Flex traduce internamente en autómatas finitos determinísticos. Se trato de obtener la mayor cantidad de tokens posibles con las expresiones a continuación:

```
"class"            { return CLASS; }
"feature"          { return FEATURE; }
"do"               { return DO; }
"from"             { return FROM; }
"until"            { return UNTIL; }
"loop"             { return LOOP; }
"end"              { return END; }
"if"               { return IF; }
"then"             { return THEN; }
"else"             { return ELSE; }
"print"            { return PRINT; }
{RESERVED_TYPE}    { return RESERVED_TYPE; }
"+"                { return PLUS; }
"-"                { return MINUS; }
"*"                { return TIMES; }
"/"                { return DIVIDE; }
">"                { return GREATER; }
"<"                { return LESS; }
"="                { return EQUAL; }
"!="               { return NOT_EQUAL; }
">="               { return GREATER_EQUAL; }
"<="               { return LESS_EQUAL; }
"not"              { return NOT; }
"and"              { return AND; }
"or"               { return OR; }
"true"             { return TRUE; }
"false"            { return FALSE; }
":="               { return ASSIGN; }
":"                { return COLON; }
","                { return COMMA; }
"("                { return PARENTHESIS_OPEN; }
{STR}              { return STRING; }
")"                { return PARENTHESIS_CLOSE; }
{COMMENT}          { return COMMENT; }
{IDENTIFIER}       { return IDENTIFIER; }
{NUMBER}           { return NUMBER; }
{REAL}             { return REAL; }

[ \t\n\r]+         ;
.                  { return UNKNOWN; }
```

Algunos expresiones particulares fueron:

```
\\ Toda palabra con letras minusculas/mayusculas y/o guion bajo
IDENTIFIER [a-zA-Z][0-9_a-zA-Z]*

\\ Un numero
DIGIT [0-9]

\\ Uno o mas numeros enteros
NUMBER [0-9]+

\\Uno o mas numeros reales
REAL ({DIGIT}+\.[0-9]*([eE][+-]?{DIGIT}+)?|{DIGIT}+[eE][+-]?{DIGIT}+)

\\ Todo string, cualquier caracter que se encuentre entre comillas
STR \"[^\"]*\"

\\ Algunas palabras reservadas para tipos
RESERVED_TYPE (INTEGER|BOOLEAN|STRING)

\\ Los comentarios, estos comienzan con doble guion medio
COMMENT \-{2}.*

\\ Ignorar espacios y saltos de linea
[ \t\n\r]+

\\ El punto es para lo no reconocido por el resto de expresiones
.
```

3 Diseño del analizador sintáctico

El analizador con Bison (**parser.y**), es el que verifica que la secuencia de tokens cumpla con la gramática definida para el subconjunto de Eiffel. En lugar de ejecutar el código inmediatamente, las acciones semánticas de Bison construyen nodos de un árbol (AST).

```
PROGRAM
    : CLASS_DEF
    ;

CLASS_DEF
    : CLASS IDENTIFIER FEATURE_LIST END
    ;

FEATURE_LIST
    : FEATURE_DEF FEATURE_LIST
    | FEATURE_DEF
    ;

FEATURE_DEF
    : FEATURE IDENTIFIER BLOCK
    ;

BLOCK
    : LOCAL_DECL DO STATEMENT_LIST END
```

```
    | DO STATEMENT_LIST END
    ;

LOCAL_DECL
    : LOCAL VAR_DECL
    ;

VAR_DECL
    : MULT_ID COLON RESERVED_TYPE
    ;

MULT_ID
    : MULT_ID COMMA ID
    | ID
    ;

STATEMENT_LIST
    : STATEMENT STATEMENT_LIST
    | STATEMENT
    ;

STATEMENT
    : IDENTIFIER ASSIGN EXPR
    | PRINT PARENTHESIS_OPEN EXPR PARENTHESIS_CLOSE
    | IF EXPR THEN STATEMENT_LIST ELSE STATEMENT_LIST END
    | FROM STATEMENT_LIST UNTIL EXPR LOOP STATEMENT_LIST END
    ;

EXPR
    : EXPR PLUS EXPR
    | EXPR MINUS EXPR
    | EXPR TIMES EXPR
    | EXPR DIVIDE EXPR
    | EXPR EQUAL EXPR
    | EXPR NOT_EQUAL EXPR
    | EXPR LESS EXPR
    | EXPR GREATER EXPR
    | EXPR LESS_EQUAL EXPR
    | EXPR GREATER_EQUAL EXPR
    | EXPR AND EXPR
    | EXPR OR EXPR
    | NOT EXPR
    | FACTOR
    ;

FACTOR
    : NUMBER
    | STRING
    | REAL
    | TRUE
    | FALSE
    | ID
    | PARENTHESIS_OPEN EXPR PARENTHESIS_CLOSE
    ;

ID
    : IDENTIFIER
    ;
```

4 Estructura del AST

Se diseñó un AST genérico utilizando struct AST en C (ast.c, ast.h), lo que permite una gestión uniforme de todos los elementos del lenguaje. Para manejar la variedad de tipos de datos de Eiffel (INTEGER, REAL, STRING) con una única estructura de nodo, se empleó una union dentro de struct, mientras campo kind determina qué miembro de la union debe ser accedido por el intérprete.

```
typedef struct AST {
    NodeKind kind;
    char *name;
    struct AST *left;
    struct AST *right;
    union {
        long ival;
        double rval;
        char *sval;
        int bval;
    } value;
} AST;
```

Se definió un conjunto de tipos de nodo que reflejan la estructura del subconjunto de Eiffel, no en su totalidad, pero si cubriendo lo básico y funcional.

```
typedef enum {
    N_PROGRAM,
    N_CLASS,
    N_FEATURE,
    N_BLOCK,
    N_VARDECL,
    N_SEQ,
    N_INT,
    N_REAL,
    N_STRING,
    N_BOOL,
    N_VAR,
    N_ASSIGN,
    N_BINOP,
    N_UNARYOP,
    N_IF,
    N_IF_BLOCK,
    N_COND,
    N_LOOP,
    N_INIT,
    N_BODY,
    N_UNTIL,
    N_PRINT,
    N_NOOP
} NodeKind;
```

5 Gestión de entornos y tabla de símbolos

La gestión correcta del ámbito léxico (scoping) es crucial para variables locales. Esto se implementó mediante una **tabla de símbolos** basada en **entornos enlazados** (**symtab.c, symtab.h**).

El sistema se basa en tres estructuras clave:

- Value: Define los tipos de datos en tiempo de ejecución (*INT*, *BOOL*, etc.) y utiliza una *union* para almacenar el dato real, asegurando un manejo tipado y eficiente de la memoria.

```
    typedef struct {
        ValueType type;
        union {
            int int_val;
            int bool_val;
            char char_val;
            char* str_val;
        } value;
    } Value;
```

- Map: Es un nodo de una lista enlazada que almacena una variable dentro de un entorno específico.

```
typedef struct Map {
    char* key;              // Nombre de la variable (identificador)
    Value value;            // Valor de la variable
    struct Map* next;
} Map;
```

- Environment: Representa un scope o ámbito léxico. Contiene la tabla de símbolos local (variables) y un puntero a su ámbito contenedor (parent).

```
typedef struct Environment {
    Map* variables;              // Lista de variables locales (Tabla de Simbolos)
    struct Environment* parent;  // Puntero al entorno padre (ambito exterior)
} Environment;
```

La distinción entre cómo se busca una variable para declararla versus cómo se busca para usarla es fundamental para evitar errores semánticos como la re-declaración o el acceso a variables inexistentes.

La declaración (*env_add_variable*) utiliza una búsqueda local, solo se busca en el entorno actual. Si la variable ya existe, se considera un error de re-declaración.

El uso/asignación (*env_lookup_variable*), una búsqueda recursiva (ámbito léxico), se busca primero en el entorno actual. Si no se encuentra, la búsqueda se delega recursivamente al parent environment, siguiendo la cadena de ámbitos hasta la raíz global.

La actualización (*env_update_variable*) también utiliza la misma lógica de búsqueda recursiva de *env_lookup_vari* para garantizar que solo se actualice una variable que ya haya sido previamente declarada en algún ámbito accesible.

Este diseño asegura que el intérprete respeta el principio de localidad de Eiffel: las variables declaradas dentro de un bloque (local) son accesibles por el código de ese mismo y cualquier bloque hijo, pero no por el ámbito padre.

6 Intérprete

El intérprete se implementa en el archivo **interpreter.c** mediante la función principal *Value evaluate_ast(AST node, Environment env)*, la cual ejecuta un recorrido recursivo en profundidad sobre el AST, evaluando las expresiones y ejecutando las sentencias en el contexto del entorno de variables actual.

- Recorrido y control de secuencia: Los nodos contenedores (como *N_PROGRAM*, *N_FEATURE*, *N_BLOCK*) y el nodo de secuencia *N_SEQ* siguen un patrón de recorrido simple:

```
evaluate_ast(node->left, env);
if (node->right)
    evaluate_ast(node->right, env);
```

Este patrón asegura que el código se ejecuta de forma secuencial de izquierda a derecha, simulando la ejecución lineal del código fuente.

- Declaración de variables anidadas: Para manejar declaraciones múltiples (*N_VARDECL*) como *x, y, z: INTEGER*, la función *declare_var_list* se utiliza como un helper estático.

Hay que marcar como importante, que en esta version en el scope de local solo se puede declarar una sola seguidilla de variables con un solo tipo.

```
class MAIN
feature
    make
        local
            x, y: INTEGER // Permitido
        do
            // ...
        end
end


class MAIN
feature
    make
        local
            x, y: INTEGER
            z: BOOLEAN // No permitido
        do
            // ...
        end
end
```

- Control de flujo: La estructura de bucle LOOP de Eiffel (from...until...loop...end) se implementa siguiendo su semántica específica: Se ejecuta la expresión de inicialización (*N_INIT*), la condición de salida (*N_UNTIL*) se evalúa antes de cada iteración del cuerpo.

- Errores semánticos en tiempo de ejecución: La seguridad del intérprete está garantizada por comprobaciones en tiempo de ejecución. Estas verificaciones se realizan utilizando el contador de línea (*extern int yylineno*) para proveer mensajes de error legibles y localizados.

```
if (existing == NULL) {
    fprintf(stderr, "\n Runtime/Semantic error: variable '%s' not declared (line %d
   )\n", ...);
    exit(1);
}
```

7 Decisiones de diseño y limitaciones

La elección de la opción A (intérprete directo) fue una decisión estratégica basada en la complejidad del proyecto, el tiempo disponible y la necesidad de priorizar la correcta implementación del ámbito léxico y las estructuras de control, ademas de ya tener algunas tecnologías conocidas de la primer entrega (Flex y Bison).

- Mapeo directo entre sintaxis y semántica (transparencia): En un intérprete de AST, la ejecución es un recorrido directo sobre el árbol generado por Bison. Esto significa que si el AST representa correctamente una sentencia (*N_ASSIGN*, *N_LOOP*), su ejecución semántica es inmediata y fácil de trazar. Durante el desarrollo, al encontrar un error (ej. una asignación incorrecta), podemos debuguear el intérprete simplemente inspeccionando el nodo AST fallido, sin tener que preocuparnos por una capa intermedia de Bytecode.

- Facilidad en la gestión de memoria (variables): Nuestra implementación de entornos enlazados (*struct Environment*) se integra de manera natural con el recorrido recursivo del AST. Cuando se entra a un nuevo bloque de código (ej. un *N_BLOCK* o *N_FEATURE*), se crea un nuevo entorno que apunta al padre. Al salir de la función *evaluate_ast* para ese bloque, el sistema de memoria de C (la pila de llamadas) se encarga de manera implícita de manejar el retorno, y solo necesitamos liberar explícitamente la memoria del entorno que creamos (*env_free*).

En resumen, la ppción A nos permitió enfocarnos en la semántica del lenguaje y la gestión de ámbitos, utilizando la recursión de C como el motor de ejecución, minimizando la sobrecarga de generar Bytecode e implementar una máquina virtual desde cero.

8 Compilación y ejecución

El proyecto incluye un Makefile automatizado para facilitar la compilación y las pruebas.

Requisitos:

- GCC (Compilador de C)
- Flex (Analizador léxico)
- Bison (Analizador sintáctico)
- Make

Comandos

- Compilar todo:
Esto genera los archivos **lex.yy.c**, **parser.tab.c** y compila el ejecutable executable.

```
username:path$  make build
```

- Ejecutar un archivo específico:

```
username:path$  make run FILE=examples/01_hello.e
```

- Correr pruebas:
Este comando busca automáticamente todos los archivos **.e** en la carpeta **examples/** y los ejecuta secuencialmente, mostrando los resultados en consola.

```
username:path$  make test
```

- Limpiar:

```
username:path$  make clean
```

9 Conclusion

Se ha logrado implementar un intérprete funcional para un subconjunto significativo de Eiffel. El sistema es capaz de realizar operaciones de entrada/salida, aritmética compleja, manejo de lógica booleana y estructuras de control de flujo. Creemos que la implementación esta completamente abierta a modificaciones para crecer significativamente rápido y fácil.

10 Reseña del lenguaje Eiffel y su origen

Lenguaje Eiffel

Eiffel es un lenguaje de programación orientado a objetos, creado a mediados de los años 80 por **Bertrand Meyer**, un informático francés, mientras trabajaba en la empresa *Interactive Software Engineering* (ISE).
Su diseño está fuertemente ligado al ámbito académico, ya que Meyer era profesor e investigador y buscaba un lenguaje que sirviera como vehículo académico para difundir y aplicar los principios de *programación por contrato* (*Design by Contract*), una metodología introducida por Meyer que establece que cada componente de software debe definirse con **precondiciones, postcondiciones e invariantes**. Esto promueve la confiabilidad, reutilización y mantenibilidad del código.
El nombre Eiffel hace referencia a la *torre Eiffel* en París, símbolo de ingeniería sólida y elegante, lo cual refleja el espíritu con el que fue concebido: construir sistemas de software tan robustos y bien diseñados como una obra de ingeniería.

Principios fundamentales

- **Orientación a objetos pura**: todo en Eiffel es objeto, incluso los tipos básicos.
- **Diseño por contrato (Design by Contract, DbC)**:
  - Una de sus mayores innovaciones.
  - Cada clase, método o módulo se especifica con precondiciones, postcondiciones e invariantes, como un contrato formal entre el código y sus usuarios.
  - Esto fomenta la confiabilidad y la verificación del software.
- **Simplicidad y legibilidad**: la sintaxis busca ser clara y cercana al lenguaje natural.
- **Soporte de herencia múltiple y polimorfismo**.

Origen universitario

Bertrand Meyer desarrolló Eiffel inicialmente en el IT Department de *Interactive Software Engineering* (hoy Eiffel Software), pero el lenguaje alcanzó su madurez y estandarización en el ámbito universitario, sobre todo en el **ETH Zürich** (Escuela Politécnica Federal de Zúrich, Suiza), donde Meyer ejerció como profesor de informática desde los años 90.
Desde entonces, Eiffel ha sido un lenguaje estrechamente ligado a la academia, aunque también tuvo aplicaciones en la industria en sectores donde la confiabilidad del software es crítica (banca, aeroespacial, defensa).

Legado

Aunque no alcanzó la popularidad masiva de Java o C++, Eiffel influyó enormemente en la evolución del software: el concepto de *Design by Contract* fue adoptado por otros lenguajes (Ada 2012, C#, Java mediante librerías, Python mediante decoradores, etc.).
Sigue siendo un referente en cursos universitarios de teoría de lenguajes y compiladores, justamente porque combina **rigor formal** con un **diseño práctico**.

11 Re-entrega

11.1 Comentarios

Para lograr agregar esta funcionalidad de Eiffel al interprete, basto con lograr que el **lexer.l** identifique los comentarios y no genere ningún token. Ademas se borro del **parser.y** el token ya que no iba a ser consumido.

```
...

{COMMENT}          { }    // <-- Identifica pero no genera token

...
```

11.2 Concatenación de strings

Antes de los cambios el analizador léxico estaba obteniendo bien al operando '+', pero solo como la operación aritmética de suma, no funcionando correctamente cuando los operandos son de tipo *STRING*. Para lograr que ademas de la operación aritmética logre la concatenación de strings se realizo un cambio sobre el interprete. Cuando la función **evaluate_ast** identifica el nodo **N_BINOP** en el AST para la operación/signo '+', se decide si realizar la suma o concatenación dependiendo del tipo de cada operando. En caso de no tener tipos admisibles se lanza un error.

```
Value evaluate_ast(AST *node, Environment *env) {
    //...
    switch (node->kind) {
        case N_BINOP: {
            Value l = evaluate_ast(node->left, env);
            Value r = evaluate_ast(node->right, env);
            result.type = INT_T;
            if (strcmp(node->name, "+") == 0){
                if(l.type == INT_T && r.type == INT_T){
                    result.value.int_val = l.value.int_val + r.value.int_val;
                }else if (l.type == STR_T && r.type == STR_T){
                    int str_len_l = strlen(l.value.str_val) > 1? strlen(l.value.
                        str_val)-1 : 0;
                    int str_len_r = strlen(r.value.str_val) > 1? strlen(r.value.
                        str_val)-1 : 0;
                    int str_len = str_len_l + str_len_r;
                    char *str = (char *)malloc(str_len + 1);
                    strncpy(str, l.value.str_val, str_len_l);
                    strncat(str, r.value.str_val +1, str_len_r);
                    result.type = STR_T;
                    result.value.str_val = str;
                }else{
                    throw_error_types(l.type, r.type);
                }
            }
        }
        // Other cases...
    }
    //...
}
```

11.3 Multiples lineas de declaración de variables

Al realizar la declaración de variables antes solo se permitía una linea y un tipo solo.

```
class MAIN
feature
    make
        local
            a, b, c: INTEGER
        do
            ...
        end
end
```

El cambio actual permite multiples lineas cada una con sus tipos particulares.

```
class MAIN
feature
    make
        local
            a, b, c: INTEGER
            bool: BOOLEAN
            x: INTEGER
        do
            ...
        end
end
```

Esto se logró sumando producciones que contemplan multiples declaraciones.

```
LOCAL_DECL
    : LOCAL VAR_DECL_LIST { $$ = $2; }
    ;

VAR_DECL_LIST
    : VAR_DECL VAR_DECL_LIST { $$ = ast_new_node(N_SEQ, "VAR_DECL_LIST", $1, $2); }
    | VAR_DECL               { $$ = $1; }
    ;
```

11.4 Números negativos

Se agregó en el **lexer.l** la obtención del token para números negativos.

```
...
NEG_NUMBER \-{1}{NUMBER}
...

{NEG_NUMBER}   {yylval.str = strdup(yytext); yycolumn += yyleng; return NEG_NUMBER;
    }

...
```

Y también su lectura en el **parser.y** para crear el nodo **INTEGER** correspondiente, pero con un valor negativo.

```
FACTOR
    : NUMBER         { $$ = ast_new_int(atol($1)); }
    | NEG_NUMBER     { $$ = ast_new_int(atol($1)); }

    ...
```

11.5 Conclusión de re-entrega

Con pequeños cambios y agregado de funcionalidades no tan complejas, el interprete se ha vuelto un poco más robusto frente a distintas pruebas.

Se agregan nuevos test:
11_comments.e, **12_concat_strings.e**, **13_multi_decl_vars.e**, **14_negative_numbers.e**