# UNIVERSIDAD TECNOLÓGICA NACIONAL

# FACULTAD REGIONAL DE BUENOS AIRES

## INGENIERÍA EN SISTEMAS DE INFORMACIÓN

### Sintaxis y Semántica de los Lenguajes - K2102

**Profesor:** Pablo Méndez

# Trabajo Práctico Final

## Grupo N°10



| Apellido y Nombre | Legajo | Correo | GitHub |
|---|---|---|---|
| Baudo Sofia | 213.498-6 | sbaudo@frba.utn.edu.ar | SofiaBaudoUTN |
| Castro Planas Ignacio | 213.579-6 | icastroplanas@frba.utn.edu.ar | nacho-castro |
| Petrocelli Azul Martin | 213.999-6 | apetrocelli@frba.utn.edu.ar | AzulPetrocelliUTNBA |
| Bertella Tomas Emiliano | 213.511-5 | tbertella@frba.utn.edu.ar | berty-tb |



**Fecha de entrega:** 03/12/2024

# Índice

# Introducción

El presente trabajo consiste en realizar un intérprete del lenguaje MICRO. Para ello se utilizó FLEX y BISON (cualquier producto que lo implemente) y lenguaje C.

El usuario debe poder seleccionar el tipo de input que realizará(si será a través de un archivo o si lo quiere ingresar a mano desde el teclado).

El programa debe compilar, es decir deben reducirse las expresiones y se deben ejecutar las sentencias. Además se deben leer y sacar por pantalla los valores correspondientes cuando las sentencias lo requieran y en caso de error mostrar en pantalla el tipo de error(lexico o sintactico) y la linea en la que se encuentra.

## a. Descripción Informal del Lenguaje Micro (Modificado)

MICRO es un lenguaje reducido, con objetivos puramente didácticos.

1. Existen dos tipos de datos: Entero y String.
2. Todos los identificadores son declarados explícitamente, con una longitud máxima de 32 caracteres.
3. Los identificadores deben comenzar con una letra o guión bajo, seguido opcionalmente de una combinación de letras, dígitos o guiones bajos.
4. Las constantes son secuencias de dígitos (números enteros) o texto literal (hardcoded) en el caso de una invocación a “escribir”.
5. Hay 3 tipos de sentencias:
    a. Declaración
    b. Asignación
    c. Entrada/Salida
6. Cada sentencia termina con un punto y coma.
7. El cuerpo de un programa está delimitado por las palabras reservadas inicio y fin.
   inicio, fin, leer y escribir son palabras reservadas y deben escribirse en minúscula.

Las funciones leer y escribir funcionan de la siguiente manera:

leer(lista de IDs) asigna los valores dados a los IDs proporcionados. Por ejemplo:

leer(a,b,c);

// se ingresa 14 27 49

// ahora a==14, b==27, c==49

escribir(lista de sentencias) se muestra en pantalla el valor de las sentencias indicadas. Ejemplo:

a := 14;

b := 27;

escribir (a, b, b + a*2);

14 27 55

Es importante aclarar que este informe tiene como objetivo servir como guía de usuario a la hora de utilizar el programa. Se incluyen nuestras conclusiones, además de la información relevante a la materia. En caso de querer conocer en extensión el funcionamiento interno, se recomienda ver el código disponible en GitHub: https://github.com/utn-frba-ssl/24-102-10

## 1. Flex

Comenzamos con el analizador léxico. Flex se usa para analizar léxicamente el código, es decir, para dividirlo en tokens.

Los tokens son los terminales de la gramática dada:

| Expresión Regular | TOKEN |
|---|---|
| inicio | INICIO |
| fin | FIN |
| leer | LEER |
| escribir | ESCRIBIR |
| [a-zA-Z_][a-zA-Z0-9_]* | ID |
| [0-9]+ | CONSTNUM |
| \"[^\"]*\" | CONSTCADENA |
| const | CONST |
| int | TIPO_INT |
| string | TIPO_STRING |
| ( | PARENIZQUIERDO |
| ) | PARENDERECHO |
| , | COMA |
| ; | PUNTOYCOMA |
| + | SUMA |
| - | RESTA |
| := | ASIGNACION |

Para ello se creó el archivo '*scanner.l*'. Se encargará de identificar los tokens, además de marcar errores léxicos.

int yylex();

//Función del analizador léxico. Es el núcleo del scanner generado por flex. Su propósito es leer el código fuente de entrada y devolver los tokens que el analizador sintáctico (bison) necesita para construir la estructura sintáctica del programa.

int yylineno;

//Indica la línea actual del archivo. Es una variable global que lleva el conteo de la línea en la que se encuentra el scanner mientras analiza el archivo.

int yylexerrs;

//Contador de errores léxicos. Lleva un registro de la cantidad de errores que ocurren durante el análisis léxico. Se incrementa cada vez que el scanner encuentra un token no reconocido o inválido.

*Código:*

. *{printf("Error léxico: carácter '%s' inválido\n", yytext); yylexerrs++;}*

## 2. Bison

Bison se encarga del análisis sintáctico. Se definió cómo deben organizarse los tokens (de Flex) en una estructura coherente de sentencias y expresiones. La gramática debe representar las reglas del lenguaje MICRO. Se buscó utilizar recursión a izquierda siempre.

**Gramática:**

<programa> -> INICIO <listaSentencias> FIN

<listaSentencias> -> <sentencia> {<sentencia>}

<sentencia> -> TIPO_INT ID PUNTOYCOMA

TIPO_INT ID ASIGNACION <expresion> PUNTOYCOMA

CONST TIPO_INT ID ASIGNACION <expresion> PUNTOYCOMA

TIPO_STRING ID PUNTOYCOMA

TIPO_STRING ID ASIGNACION CONSTCADENA PUNTOYCOMA

CONST TIPO_STRING ID ASIGNACION CONSTCADENA PUNTOYCOMA

ID ASIGNACION <expresion> PUNTOYOCOMA

LEER PARENIZQUIERDO <listaIdentificadores> PARENDERECHO PUNTOYCOMA

ESCRIBIR PARENIZQUIERDO <listaExpresiones> PARENDERECHO PUNTOYCOMA

<listaIdentificadores> -> ID {COMA ID}

<listaExpresiones> -> <expresion> {COMA <expresion>}

<expresion> -> <primaria>

{SUMA <primaria>}

{RESTA <primaria>}

<primaria> -> ID | CONSTNUM | CONSTCADENA

| PARENIZQUIERDO <expresion> PARENDERECHO

El archivo ‘parser.y’ será el encargado de inicializar el análisis y coordinar entre Flex y la Tabla de Símbolos. Se maneja la entrada del usuario (desde archivo o entrada estándar). Es necesario integrar el lexer generado por Flex y la Tabla de Símbolos en lenguaje C.

```
int main() {
  int opcion;
  printf("Seleccione entrada:\n 1. Archivo\n 2. Teclado\n");
  scanf("%d", &opcion);
  if (opcion == 1) { // Cargar un archivo
      char archivo[100];                  // Nombre del archivo
      printf("Ingrese el nombre del archivo: ");
```

```
      scanf("%s", &archivo);
      yyin = fopen(archivo, "r");
  } else
        yyin = stdin;
    inicializarTabla(); // Inicializa la tabla con todo en -1
    // Parser
    switch (yyparse()) {
        case 0:
          printf("\nCompilado exitosamente\n");
          break;
        case 1:
          printf("Error de compilacion\n");
          break;
        case 2:
          printf("Memoria insuficiente\n");
          break;
    }
    printf("Cantidad de errores Lexicos: %i\n", yylexerrs);
    printf("Cantidad de errores Sintacticos: %i\n", yynerrs);
    return 0;
}
```

El código main solicita al usuario indicar el tipo de entrada, ya sea por archivo o teclado. En caso de ingresar por teclado, el usuario debe ingresar cada línea de código seguida de salto de línea. Bison se encargará de revisar línea por línea. Cuando identifica un error se detiene y emite un mensaje con el tipo de error.

Bison lleva una cuenta de la cantidad de errores sintácticos y léxicos encontrados. También realiza los llamados correspondientes a la tabla de símbolos, la cual lleva el registro de los identificadores y sus valores asignados.

int yyerror(char *s);

//Función de manejo de errores. Se invoca automáticamente cada vez que el analizador sintáctico (bison o yacc) detecta un error en la entrada que no coincide con ninguna regla definida en la gramática.

```
int yyerror(const char* s) {
  extern int yylineno;  // Línea actual del analizador léxico
  extern char* yytext;  // Texto del token actual
```

```
  printf("Error Sintáctico. Línea %d: token invalido '%s'.\n",
yylineno, yytext);
  return 0;
}
```

## 3. Archivo TS.c

La Tabla de Símbolos es una estructura de datos compleja que es utilizada para el almacenamiento de todos los identificadores del programa a compilar.

Cada elemento de la TS (que llamaremos 'símbolo'), está formado por una cadena y sus atributos. En nuestro caso, tendrá como atributo un código que registre si el identificador es de tipo entero o texto, indicando también si es una constante.

Se decidió de forma arbitraria que la Tabla de Símbolos podrá contener hasta 50 identificadores. Mientras que los identificadores tendrán un largo de 32 caracteres. Las cadenas literales son un vector de hasta 255 caracteres, incluyendo el '\0'.

```
typedef struct {
    char id[32];
    int tipo; //1.Entero. 0.Cadena
    int entero;
    char cadena[255];
    int esConst; //1.True 0.False
} simbolo;

simbolo TS[50];//TABLA DE SIMBOLOS. 50 DE TAMAÑO
```

La Tabla cuenta con los siguientes métodos:

void inicializarTabla();

//Inicializa la tabla con todo en -1.

int indiceTabla(char* id);

//Busca un id por su nombre y nos devuelve su posición en la Tabla. En caso de no encontrarlo devuelve -1.

char* retornarValorCad(char* id);

int retornarValorInt(char* id);

//Ambos obtienen el valor (entero o cadena) asociado al identificador. Devuelve el valor si el identificador existe, o muestra un error si no está definido o no es del tipo correcto.

void escribirIntTabla(char* s, int valor, int esConst);

void escribirCadTabla(char* s, char* valor, int esConst);

//Ambos escriben un valor en la tabla de símbolos:

- Si el identificador no existe, lo agrega a la Tabla.
- Si es una constante, no permite modificarlo.
- Si ya existe como variable, actualiza su valor.

int tipoVariable(char* s);

//Devuelve el tipo del identificador:

- 1 para enteros.
- 0 para cadenas.
- -1 si el identificador no está definido.

void cargarEntradas(char* p1); //leer(id);

//Solicita al usuario un valor para un identificador:

- Si el valor es numérico, lo almacena como entero.
- Si es texto, lo almacena como cadena.

void imprimir(char* id); //escribir(id);

//Muestra en pantalla el valor asociado a un identificador:

- Si es entero, imprime el número.
- Si es cadena, imprime el texto.
- Muestra un error si no encuentra el identificador.

# 4. Compilación

Se debe ejecutar:

- flex scanner.l
- bison -d parser.y
- gcc parser.tab.c lex.yy.c TS.c -o micro

Una vez generado el ejecutable ‘micro.exe’ podemos iniciarlo.

- ./micro

El programa emitirá por pantalla un mensaje donde solicita el tipo de entrada al usuario.

# Parte teórica

### 1. Enumere las fases de compilación de un programa C. Identifique en qué fases interactúa la tabla de símbolos.

El proceso de compilación en C se puede dividir en varias fases, cada una con tareas y objetivos específicos. Estas fases garantizan que el código fuente se traduzca a una forma eficiente y ejecutable, al mismo tiempo que se comprueban los errores y se permite la modularidad. Las fases del proceso son:

1. Preprocesamiento
2. Compilación
3. Ensamblado
4. Enlazado

Además, el proceso de compilación en sí puede subdividirse en más etapas como se vió a lo largo de la materia:

2.1 Fase de análisis
    2.1.1 Análisis Léxico
    2.1.2 Análisis Sintáctico
    2.1.3 Análisis Semántico
2.2 Fase de Síntesis
    2.2.1 Generación de código intermedio
    2.2.2 Optimizador de código
    2.2.3 Generación de código en lenguaje ensamblador

En cuanto a la tabla de símbolos, esta interactúa únicamente con la fase de compilación, específicamente con el análisis sintáctico y semántico, siendo el primero el que va guardando en ella las variables con sus correspondientes datos brindados, mientras que el segundo se encarga de verificar que semánticamente, la ubicación de cada uno de los símbolos con su tipo de dato sea correcta, además de ejecutar cada procedimiento semántico asociado usando esa tabla de símbolos para resolver las referencias de cada variable.

**2. Describa el concepto de “espacio de nombres” en C (namespaces) y dé ejemplos. ¿Qué rol cumple la tabla de símbolos con este concepto?**

El espacio de nombres en C se refiere a la manera en que se gestionan los identificadores (como variables, funciones, y tipos) dentro de diferentes contextos o ámbitos (como dentro de una función, un archivo o un programa completo). Este concepto se puede entender como el conjunto de reglas que definen cómo se organizan y acceden a los identificadores en un programa para evitar colisiones de nombres y gestionar correctamente su visibilidad.

En cuanto a la tabla de símbolos, esta debe ser capaz de identificar en qué espacio de nombre se encuentra un identificador, para que al momento de realizarse el análisis semántico, el parser sepa que si encuentra dos identificadores iguales, pero en espacios distintos, entonces no deberá producirse un error semántico. En cambio si durante el análisis semántico se encontraran dos identificadores iguales pertenecientes al mismo espacio de nombres, debe producirse un error semántico.

**3. Investigue y describa el makefile de C. Enumere sus parámetros y dé un ejemplo de uso utilizando un programa con bibliotecas.**

El makefile en C se utiliza para crear y construir programas. El makefile es un archivo utilizado por el make. Es una herramienta utilizada para poder compilar el programa. Describe las relaciones entre los archivos y las operaciones a ejecutarse en los mismos. Además, el makefile es necesario para estar seguro que se recompile únicamente cuando cambios fueron realizados y no constantemente.

El makefile está compuesto por la siguiente estructura:

nombreArchivo: dependencias…

comando1

comando2

comandoN

Un ejemplo de uso de un programa que utiliza una biblioteca, en este caso "math" podría ser la siguiente:

*Archivo main.c :*

```
#include <math.h>

int main{...}
```

*Makefile:*

```
programaEjemplo: main.o

gcc -o programaEjemplo main.o -lm
```

**4. Describa los pasos que realiza make para llegar al programa ejecutable. Identifique en qué paso se confeccionan los programas objeto y explique qué función cumple el linker.**

El comando make está compuesto por una serie de pasos. Estos son:

1. Leer el makefile
2. Identificar las dependencias, los objetos etc.
3. Verificar que archivos en las dependencias deben ser compilados otra vez.
4. Se compilan los archivos fuentes a objetos.
5. Se “enlazan” los archivos mediante el linker.

Los programas objetos se confeccionan en el paso 4 en la parte de compilación.

El linker es un programa del sistema que une diferentes módulos del programa en un solo archivo. Este, colecta y mantiene los datos y el código para almacenarlos en un único archivo. Se utiliza en las últimas partes del proceso de compilación.

# Conclusión

El programa implementa un analizador léxico, sintáctico y semántico para el lenguaje Micro modificado, que gestiona enteros y cadenas. Es capaz de procesar declaraciones, asignaciones y operaciones aritméticas, además de realizar la entrada/salida de datos. Utiliza una Tabla de Símbolos (TS) para almacenar las variables y constantes del programa, asegurando que se respeten los tipos de datos y las restricciones (como las constantes que no se pueden modificar).

La detección y manejo de errores léxicos y sintácticos informan al usuario de manera clara sobre inconsistencias o mal uso del lenguaje. Además, el programa es escalable, permitiendo agregar nuevas funcionalidades.

En resumen, el programa logra abarcar los conceptos de compiladores, con un manejo eficiente de las estructuras de datos. Es capaz de interpretar y validar programas escritos en lenguaje Micro.