Universidad Tecnológica Nacional Facultad Buenos Aires

Sintaxis y Semántica de los Lenguajes,

Curso: K2102

Profesor: Esp. Ing. Pablo D. Méndez

Trabajo Práctico Nro. 3

Compilador

Fecha de entrega: 03/12/2024

Integrantes:

- Albornoz, Tobias Abel (legajo: 209.930-5)

  @tobias-albornoz <talbornoz@frba.utn.edu.ar>

- Figueredo Chirinos, Jesus Antonio (legajo: 176.605-3)

  @phosph <jfigueredochirinos@frba.utn.edu.ar>

- Scarfo, Ignacio Alejo (legajo: 214.123-1)

  @iscarfo <iscarfo@frba.utn.edu.ar>

- Venero Alosilla, Nicolás Alejandro (legajo: 202.792-6)

  @nickvenero <nveneroalosilla@frba.utn.edu.ar>

Índice

Introducción

El presente trabajo tiene como objetivo desarrollar un intérprete para el lenguaje de programación didáctico MICRO. Para su implementación se utilizarán las herramientas FLEX y BISON, que permitirán construir el analizador léxico y sintáctico, respectivamente, y el lenguaje de programación C. MICRO es un lenguaje simplificado, ideal para el aprendizaje de los principios fundamentales de programación y diseño de intérpretes, en el cual se manejan únicamente dos tipos de datos: enteros y cadenas de caracteres.

El proyecto busca dotar al intérprete de las capacidades necesarias para recibir sentencias desde la entrada estándar o desde un archivo, según la preferencia del usuario. Estas sentencias se procesarán, se ejecutarán y, en caso de requerirse, se mostrarán los resultados en pantalla. A diferencia del MICRO original propuesto por Muchnik, en esta implementación se requerirá que las variables sean declaradas explícitamente, así como también se habilitará la creación de constantes a través de sentencias const.

A la hora de la división de tareas, al principio designamos diferentes tareas para cada uno, por ejemplo designamos un punto teórico para cada uno, Albornoz y Scarfo se encargaban de la declaración e inicialización de variables, Figueredo Chirinos de estructurar el código, manejar el makefile y desarrollar la tabla de símbolos y Venero Alosilla de las operaciones escribir y leer, cada uno en su respectiva rama del git. Sin embargo, al realizar la solicitud de pull todos terminamos pensando y realizando cambios sobre el trabajo de los otros, por ende, a pesar de realizar una división de tareas, terminamos ocupándonos todos de todo generando un trabajo mucho más integral y colaborativo

Propósito General

Verificar la validez de un programa escrito en lenguaje MICRO, asegurándose de que cumpla con las reglas léxicas, sintácticas y semánticas establecidas.

Hipótesis Consideradas

En el desarrollo de este sistema, se asumieron una serie de hipótesis para definir el alcance y las reglas del lenguaje, asegurando un funcionamiento claro y consistente. Estas hipótesis guían la interpretación de las entradas y los errores que el sistema puede manejar.

1. Sobre la Declaración y Uso de Variables

Declaración explícita: Todas las variables y constantes deben declararse antes de ser usadas. No se permite la redeclaración en el mismo programa.

Tipos asociados: Tanto variables como constantes tienen un tipo fijo (`int` o `string`) que no puede cambiar durante la ejecución.

Constantes:

- Deben declararse con la palabra clave `const` y asignarse un valor al momento de la declaración.
- Su valor es inmutable tras la declaración.
- Están sujetas a las mismas restricciones de tipos que las variables.

Compatibilidad de asignaciones:

- Variables y constantes de tipo `int` sólo pueden almacenar números o expresiones aritméticas válidas.
- Variables y constantes de tipo `string` sólo pueden almacenar cadenas o concatenaciones de cadenas.

2. Sobre las Operaciones

Aritméticas: Las operaciones (+, -) se aplican únicamente a valores numéricos.

Cadenas: La única operación permitida para `string` es la concatenación (`+`).

Errores semánticos: Las operaciones con tipos incompatibles se consideran inválidas.

3. Sobre los Literales

- **Numéricas:** Sólo se aceptan literales enteros, excluyendo números con decimales.

- **Cadenas:** Deben estar delimitadas por comillas dobles (`"`) y pueden incluir caracteres escapados (por ejemplo, `\n` para salto de línea o `\t` para tabulación).
- Literales que no cumplan con estos formatos son rechazados durante el análisis léxico.

4. Sobre las Sentencias de Entrada y Salida

- **Leer:** Los identificadores deben haber sido previamente declarados, de lo contrario se genera un error.
- **Escribir:** Evalúa y muestra las expresiones dadas, sin alterar el estado de las variables.

5. Sobre la Estructura del Lenguaje

El programa debe iniciar con `inicio` y finalizar con `fin`. Cualquier desviación se considera un error.

Las sentencias deben terminar con un punto y coma (`;`).

No se permite la anidación de estructuras.

6. Sobre la Tabla de Símbolos

-La tabla de símbolos es única para todo el programa (es decir, no se considera un entorno de ejecución con ámbitos locales o anidados).

Cada identificador tiene un único nombre, sensible a mayúsculas y minúsculas, que debe ser único dentro del programa.

La eliminación de símbolos o el manejo de ciclos de vida de variables (por ejemplo, variables temporales) no están contemplados en esta implementación.

7. Sobre los Errores

Los errores detectados pueden clasificarse en:

- **Léxicos:** Identifican tokens inválidos, como caracteres no permitidos.
- **Sintácticos:** Detectan incumplimiento de reglas gramaticales, como la falta de `;`.
- **Semánticos:** Señalan inconsistencias lógicas, como operaciones con tipos incompatibles.

Los errores se reportan indicando su tipo y posición, pero no se corrigen automáticamente.

8. Sobre la Modularidad

Las funcionalidades del intérprete están interconectadas pero diseñadas de forma independiente

El analizador léxico asume que los patrones proporcionados son suficientes para identificar todos los componentes válidos del lenguaje.

El analizador sintáctico confía en que los tokens generados por el léxico son correctos y utiliza las reglas gramaticales para validar su estructura basado en reglas gramaticales.

El análisis semántico asume una tabla de símbolos correctamente configurada.

9. Alcance y Limitaciones

Alcance: El lenguaje admite variables, constantes, y operaciones básicas, pero carece de estructuras avanzadas como funciones, ciclos o condicionales.

Limitaciones: No se incluye un sistema de ciclos de vida para variables temporales.

Componentes Principales

Definición Léxica (FLEX)

Se diseñó un analizador léxico para reconocer los tokens definidos en la gramática de MICRO, como palabras reservadas (`inicio`, `fin`, `leer`, `escribir`), identificadores, constantes numéricas y cadenas de texto.

- **Expresiones Regulares:**
  - Identificadores (`ID`): comienzan con una letra o guión bajo (`_`), seguidos opcionalmente por letras, dígitos o guiones bajos.
  - Constantes numéricas (`CONSTANTE`): secuencias de dígitos opcionalmente precedidas por un signo negativo.
  - Literales de texto (`CONSTANTE`): cadenas delimitadas por comillas dobles, con soporte para secuencias de escape (`\n`, `\t`, etc.).

Analizador Sintáctico (BISON)

El analizador sintáctico de MICRO, construido con **BISON**, establece las reglas gramaticales que definen la estructura jerárquica y válida de los programas escritos en este lenguaje. Estas reglas aseguran que el programa cumpla con los estándares sintácticos establecidos para MICRO. A continuación, se detalla cada uno de los aspectos fundamentales:

1. Estructura del Programa

Se define una regla general para delimitar el inicio y fin del programa utilizando las palabras clave `inicio` y `fin`. Esta estructura garantiza que todos los programas sigan un formato

consistente, comenzando y terminando correctamente. Esta gramática asegura que el programa contenga un cuerpo válido compuesto por sentencias bien formadas.

2. Declaraciones y Asignaciones

Se definen reglas que validan que las variables sean declaradas explícitamente antes de su uso. También se incluyen las estructuras necesarias para evaluar expresiones matemáticas y realizar asignaciones.

3. Entrada y Salida

Se incorporan las reglas para manejar sentencias de entrada y salida, permitiendo al usuario interactuar con el programa.

Esto permite sentencias como `leer(x, y);` para capturar datos ingresados por el usuario o `escribir(x + y);` para mostrar resultados.

Análisis Semántico

Se verifica que las operaciones entre variables sean compatibles con sus tipos. Algunos ejemplos:

- Asignar un número a una variable `int` es válido, pero intentar asignar una cadena genera un error.
- Operaciones matemáticas solo se permiten entre variables de tipo `int`, mientras que las concatenaciones aplican únicamente a cadenas.
- Se asegura que las variables sean declaradas antes de su uso. Si se detecta el uso de una variable no declarada, se genera un error semántico.
- Se evita la redeclaración de una variable o constante dentro del mismo ámbito.

Tabla de Símbolos

La tabla de símbolos es una estructura clave en el intérprete. Su propósito es mantener un registro de todas las variables y constantes declaradas en el programa, incluyendo su tipo y valor actual. Este mecanismo es fundamental durante el análisis semántico y la ejecución.

Funcionalidades de la Tabla de Símbolos:

- Registro de Declaraciones: Cuando una variable o constante es declarada, se agrega un nuevo registro a la tabla con información como su identificador, tipo y valor inicial.
- Consulta de Información: Permite verificar si una variable ha sido declarada previamente o buscar su tipo para validar asignaciones y operaciones.
- Actualización de Valores: En el caso de variables, la tabla permite modificar su valor cuando se realiza una asignación válida.

Evaluación de Expresiones

El sistema procesa expresiones aritméticas y de texto para garantizar que sean evaluadas correctamente durante la ejecución. Esto incluye:

Expresiones Aritméticas: Se resuelven operaciones como suma, resta y agrupaciones utilizando un árbol de derivación generado por las reglas de BISON.

Expresiones de Texto: Se valida que las concatenaciones de cadenas sigan las reglas del lenguaje y se procesen adecuadamente.

Finalmente, los resultados de las expresiones son utilizados en asignaciones, sentencias de salida o como parte de otras operaciones, asegurando un manejo correcto de tipos y valores.

Interfaz de Entrada y Salida

El intérprete soporta dos modos de entrada:

- **Entrada estándar:** el usuario puede escribir sentencias directamente en la consola.
- **Entrada desde archivo:** el intérprete lee un archivo con sentencias MICRO.

La salida del programa incluye los resultados de las sentencias `escribir` y mensajes de error en caso de inconsistencias léxicas, sintácticas o semánticas.

Gestión de Errores

Clasifica y reporta errores en:

- **Léxico:** Tokens inválidos.
- **Sintáctico:** Secuencias mal formadas.
- **Semántico:** Problemas lógicos como incompatibilidad de tipos.

Casos de Validación

Se desarrollaron varios casos para probar las características del intérprete, tanto de manera manual como automatizada, para garantizar el correcto funcionamiento en diversos escenarios.

Validación automatizada con `ejemplos/run_test.sh`

El archivo `run_test.sh` es un script de Bash que automatiza la ejecución de pruebas para programas escritos en MICRO. Su objetivo principal es procesar múltiples casos de prueba almacenados en la carpeta./`ejemplos/`, capturar salidas estándar y errores, y generar resúmenes claros sobre el desempeño del intérprete.

Funcionamiento del script

- **Búsqueda de pruebas:**
  El script busca todos los archivos `.micro` en el directorio de ejemplos, excluyendo aquellos que comienzan con `leer_` o `no-test_`.
- **Ejecución de cada prueba:**
  Utiliza el ejecutable del intérprete (`./bin/solucion`) para procesar los archivos de prueba, capturando tanto la salida estándar como los errores en archivos temporales.
- **Criterio de validación:**
  - Las pruebas se consideran exitosas si el intérprete termina correctamente, salvo para aquellos archivos que el nombre del archivo comiencen con `error_`, se considera correcta la prueba si emiten error.
  - Los errores capturados en casos no esperados se notifican tanto en la consola como en el archivo `error.log`.
- **Manejo de logs:**
  Las salidas estándar de todas las pruebas se consolidan en `output.log`, mientras que los errores se almacenan en `error.log`.

  Comando "`make test`"

El `Makefile` del proyecto incluye un objetivo `test`, que simplifica la ejecución de todas las pruebas al invocar automáticamente el script `run_test.sh`. Su comportamiento esperado es:

```
===== Ejecutando prueba: error_variable-no-inicializada.micro =====
Prueba error_variable-no-inicializada.micro completada exitosamente.
Errores capturados en error_variable-no-inicializada.micro:
ERROR: la variable 'chao' no está inicializada.
===== Fin de prueba: error_variable-no-inicializada.micro =====


===== Ejecutando prueba: imprimir-expresion.micro =====
Prueba imprimir-expresion.micro completada exitosamente.
Sin errores en imprimir-expresion.micro.
===== Fin de prueba: imprimir-expresion.micro =====
```

- Compila el intérprete si es necesario.
- Ejecuta todas las pruebas disponibles en la carpeta `./ejemplos/`.
- Genera y muestra un resumen de los resultados directamente en la consola.

Flujo de Ejecución

1. Entrada del programa fuente.
2. Análisis léxico para generar tokens.
3. Validación sintáctica basada en las reglas gramaticales.
4. Verificación semántica, generando la tabla de símbolos.

5. Reporte de errores o confirmación de validez.

El sistema no corrige errores, pero proporciona mensajes detallados para identificar y resolver problemas. Es una herramienta educativa para entender los fundamentos del análisis de lenguajes.

Conclusión: (parte práctica)

En conclusión, el proyecto de desarrollo de un intérprete para el lenguaje MICRO utilizando FLEX, BISON y C permitió abordar y consolidar los principios fundamentales del diseño e implementación de intérpretes. MICRO, como lenguaje didáctico simplificado, sirvió para explorar conceptos clave como la definición de diferentes gramáticas, el análisis léxico y sintáctico, y la verificación semántica mediante una tabla de símbolos.

El intérprete asegura el cumplimiento de reglas claras y estrictas: declaración explícita de variables y constantes, operaciones coherentes con los tipos de datos definidos, y validación de la estructura general del programa. Además, incluye un manejo exhaustivo de errores léxicos, sintácticos y semánticos, lo que brinda retroalimentación detallada al usuario.

Se implementaron dos modos de interacción, por consola y mediante archivos, facilitando la evaluación de sentencias y expresiones. Asimismo, el uso de herramientas como un script de prueba automatizado y un sistema de logs permitió verificar el funcionamiento del intérprete en diversos escenarios y garantizar la calidad del desarrollo.

Por ende, su diseño modular y educativo lo convierte en una base sólida para profundizar en el diseño de lenguajes de programación y herramientas de interpretación. Este proyecto destaca no solo por cumplir con los objetivos planteados, sino también por su aporte formativo en el campo de la construcción de lenguajes.

DESARROLLO PARTE TEÓRICA

Fases de compilación de un programa C

1. **Etapa de Preprocesamiento**: el preprocesador se encarga de procesar directivas como `#include` y `#define`. El resultado es una versión intermedia del código fuente en la que se reemplaza cada instancia de una macro con su contenido correspondiente y se han incluido los archivos de cabecera.
2. **Analisis Lexico:** Encargado de convertir el código fuente en una secuencia de tokens.
3. **Análisis Sintáctico:** Asegura que la secuencia de tokens generada por el analizador léxico siga las reglas de sintaxis del lenguaje
4. **Análisis Semántico:** Se comprueba que las operaciones realizadas en las expresiones sean válidas.
5. **Generación de código Intermedio:** Se genera un código intermedio que se sitúa entre el código fuente y el código máquina. Este código intermedio es más fácil de optimizar y traducir a diferentes arquitecturas de hardware.
6. **Optimización:** Se aplican técnicas de optimización para mejorar el rendimiento del código intermedio, eliminando redundancias y mejorando la eficiencia del código resultante.
7. **Generación de código Objeto:** el compilador traduce el código intermedio optimizado a código objeto, que es específico para la arquitectura del sistema en el que se ejecutará.
8. **Enlazador:** Se encarga de combinar los archivos objeto, obteniendo un programa ejecutable. También resuelve referencias externas, vinculando funciones y variables que pueden estar definidas en diferentes módulos.

Las fases que interactúan con la tabla de Símbolos son:

- **Análisis Léxico**, se verifica cuáles lexemas pueden ser reconocidos por los tokens y se comienza a construir una lista de identificadores que luego se almacenará en la tabla de símbolos.
- **Análisis Sintáctico,** se utiliza la tabla de símbolos para verificar que los identificadores (variables y funciones) que aparecen en las expresiones y declaraciones han sido declarados previamente.
- **Análisis Semántico,** la tabla de símbolos se utiliza para:
    - **Verificación de tipos**: Asegurarse de que las operaciones en las expresiones sean semánticamente válidas (por ejemplo, no intentar sumar un entero con una cadena).
    - **Alcance**: Determinar si un identificador es accesible desde el contexto actual, lo que implica verificar si una variable local está fuera de su ámbito.
    - **Verificación de argumentos**: Corroborar que la cantidad y tipos de argumentos utilizados en una llamada a función coincidan con su declaración.

Espacio de Nombres en C

Un **“espacio de nombres” o namespace** en C organiza los identificadores (como nombres de funciones,variables y tipos) para evitar conflictos. Aunque C no tiene namespaces explícitos (a diferencia de C++), maneja namespaces implícitos lo que permite controlar el alcance y evitar conflictos de nombres. Esta estructura organiza los identificadores en función del contexto, ayudando a manejar el uso de variables, funciones y tipos en distintos lugares del programa.

Ejemplos: espacio de nombres de etiquetas (nombres de struct, union y enum), espacio de nombres global (para variables y funciones definidas a nivel global), espacio de nombres local (para variables y parámetros dentro de una función).

Ejemplo en C:

```c
#include <stdio.h>

struct Animal {
    char nombre[20];
    int edad;
};

int edad = 10; // Variable global en el espacio de nombres global

void mostrarEdad() {
    int edad = 5; // Variable local a esta función
    printf("Edad local: %d\n", edad);
}

int main() {
    struct Animal perro;
    perro.edad = 3;
    printf("Edad del animal: %d\n", perro.edad); // Espacio de nombres de etiquetas
    mostrarEdad();
    printf("Edad global: %d\n", edad); // Espacio de nombres global
    return 0;
}
```

En este ejemplo “Animal” está dentro de nombres de etiqueta, y edad en struct Animal, edad en main, y edad en `mostrarEdad` son tres variables distintas en diferentes espacios de nombres: de etiquetas, local y global.

La **tabla de símbolos** organiza los identificadores según su espacio de nombres y su alcance. Esto permite resolver ambigüedades y asegurarse de que se acceda a los elementos correctos en cada contexto y para cada conjunto de nombres del programa. En este sentido la tabla ayuda al compilador a:

- Gestionar el alcance: permite que diferentes partes del programa puedan usar identificadores con el mismo nombre sin causar conflictos.
- Resolver referencias: cuando el compilador encuentra una referencia a un identificador, consulta la tabla de símbolos para verificar su tipo, espacio de nombres y dirección en memoria.

- Verificar tipos y contextos: al verificar el tipo de una variable, función o tipo, la tabla de símbolos permite comprobar que las operaciones y el uso de los identificadores sean consistentes con su declaración

Makefile en C

Un makefile es un archivo utilizado por el programa make, en el cual se escriben reglas de compilación que el programa make seguirá al momento de compilar un programa. El formato de las reglas el:

```
objetivo: dependencias
→ comandos
```

donde:

- **`objetivo:`** se especifica el módulo o programa que queremos crear
- **`dependencias:`** también llamados prerrequisitos, acá se definen qué otros módulos o programas son necesarios para conseguir el **`objetivo`**
- **`comandos:`** se indican los comandos necesarios para llevar esto a cabo. Pueden ser una o varias líneas y todas deben estar indentadas con un **caracter de tabulación**.

ejemplo:

```
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
	cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o

main.o : main.c defs.h
	cc -c main.c
kbd.o : kbd.c defs.h command.h
	cc -c kbd.c
command.o : command.c defs.h command.h
	cc -c command.c
display.o : display.c defs.h buffer.h
	cc -c display.c
insert.o : insert.c defs.h buffer.h
	cc -c insert.c
search.o : search.c defs.h buffer.h
	cc -c search.c
files.o : files.c defs.h buffer.h command.h
	cc -c files.c
utils.o : utils.c defs.h
	cc -c utils.c
clean :
	rm edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

Una vez definido el Makefile, para compilar un programa basta con ejecutar el programa make con el comando `**make**` y make va a ejecutar todos los **`objetivo`** en orden según las dependencias hasta haberlos ejecutado a todos. Casos particulares donde los **`objetivos`**

no tiene dependencias y no son dependencias de ningún otro objetivo, no son ejecutados automáticamente, estos son llamados *PHONY Targets* y son vistos como simples nombres de acción, que para ser ejecutadas deben ser explicitadas al invocar al make,

ej: `make clean`

distintas acciones (targets) pueden ser explicitadas y serán ejecutadas en orden:

`make targetA targetB … targetN`

Pasos del make en C

Los pasos que realiza el make son:

1.Lectura del makefile: Make inicia y lee el archivo Makefile que contiene las reglas de compilación. Luego identifica el objetivo por defecto o el objetivo especificado en la línea de comandos.

2.Evaluación de Objetivos y Dependencias: En este paso Make analiza las dependencias de cada objetivo, osea de cada archivo que se desea construir (este archivo puede ser un archivo ejecutable o un archivo objeto). Cada objetivo tiene una lista de dependencias, que son otros archivos necesarios para construir ese objetivo.

3.Verificación de Tiempos de modificación: Make verifica las fechas de modificación de los archivos de dependencias en comparación con el objetivo. Si el objetivo no existe o alguna de sus dependencias ha sido modificada antes que el objetivo, se determina que el objetivo está desactualizado y necesita ser reconstruido.

4.Ejecución de comandos Asociados: Si el objetivo está desactualizado, make ejecuta los comandos asociados especificados en la regla para ese objetivo. Estos comandos son instrucciones del compilador para convertir archivos fuente en programas objeto o para enlazar archivos objeto en un ejecutable.

Si el objetivo está actualizado no es necesario la ejecución de comandos asociados.

5.Recursividad: El make aplica recursivamente el mismo proceso a cada dependencia que también es un objetivo con sus propias dependencias, así se asegura que todas las dependencias están actualizadas antes de construir el objetivo principal.

6.Construcción del Objetivo Principal: Una vez actualizadas todas las dependencias, Make ejecuta el comando final para construir el objetivo principal, generalmente esto implica enlazar varios programas objeto juntos. El linker es el que se encarga de enlazar los programas objetos generados por el compilador y resolver todas las referencias entre ellos para producir el ejecutable final.

7.Finalización: Make termina después de que todos los objetivos especificados han sido procesados y construidos si es necesario.