Tecnológico de Monterrey

Proyecto 4: Generador de Código

Miguel Ángel Cabrera Victoria A017282982

Francisco Martínez Gallardo Lascurain A01782250

29 de mayo del 2025

Desarrollo de aplicaciones avanzadas de ciencias computacionales

Dr. Victor Manuel de la Cueva Hernandez

El compilador desarrollado en este proyecto genera código ensamblador **MIPS**, esta arquitectura fue seleccionada como objetivo de generación de código, ya que tiene una simplicidad y claridad al tener instrucciones reducidas, también los desarrolladores instalaron MISP en clase y ya tenía algo de experiencia en el.

Manual de Usuario

Requisitos

- Python 3
- Simulador MISP o MARS

Archivos

- main.py - Archivo principal para ejecutar el compilador
- parser.py - Analizador Semantico
- symtab.py - Analizador Sintactico / Tabla de simbolos
- gloabalTypes.py
- cgen.py - Generador de Código
- test files - Folder con archivos de prueba

1. **Ejecución Compilador**

   El usuario debe posicionarse en la dirección raíz del proyecto **Compiladores**

   El usuario debe abrir un terminal, dentro de la dirección raíz.

   Ejecutar main.py

   

2. **Ejecución MARS**

   El usuario debe abrir la aplicación MISP o MARS

   

   El usuario debe copiar el contenido del archivo output.asm

   El usuario debe pegar el contenido copiado en MISP o MARS

   El usuario debe ensamblar el código con la siguiente imagen

   

   El usuario debe ejecutar el código con la siguiente imagen

   

Apéndices

A continuación se incluyen los documentos y diagramas complementarios al proyecto de análisis léxico y semántico.

Apéndice A: Diagrama de Estados Finito Determinista (DFA)

Figura A.1 muestra el DFA que reconoce identificadores, números y operadores, con transiciones específicas para cada token.

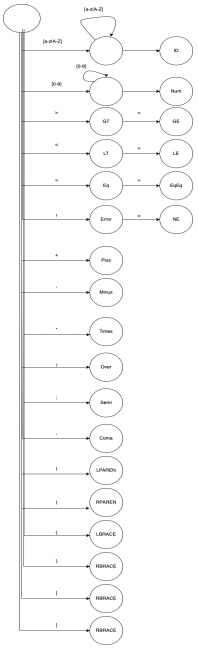

Figura A.1 DFA.

Apéndice B: Documento del Analizador Léxico

Expresiones Regulares para tokens

- `^\$` → `ENDFILE`
- `^[0-9]+$` → `NUM`
- `^[a-zA-Z]+$` → `ID`
- `^(if|else|int|return|void|while)$` → `RESERVED_WORD`
- `\+$` → `PLUS` (`+`)

- `\-$` → `MINUS` (`-`)
- `\*$` → `TIMES` (`*`)
- `/$` → `OVER` (`/`)
- `==$` → `EQEQ` (`==`)
- `!=$` → `NE` (`!=`)
- `=$` → `EQ` (`=`)
- `<$` → `LT` (`<`)
- `<=$` → `LE` (`<=`)
- `>$` → `GT` (`>`)
- `>=$` → `GE` (`>=`)
- `;$` → `SEMI` (`;`)
- `,$` → `COMMA` (`,`)
- `\($` → `LPAREN` (`(`)
- `\)$` → `RPAREN` (`)`)
- `\{$` → `LBRACE` (`{`)
- `\}$` → `RBRACE` (`}`)
- `\[$` → `LBRACKET` (`[`)
- `\]$` → `RBRACKET` (`]`)

Documentación del Analizador Léxico

El lexer está compuesto por tres archivos en Python:

- **main.py**: Lee el archivo fuente (`sample2.txt`), inicializa el lexer y extrae tokens con `getToken`.
- **lexer.py**: Contiene la lógica del analizador léxico y la función para extraer el siguiente token.
- **types.py**: Define los tipos de tokens, estados y palabras reservadas como *enums*.

Este analizador léxico reconoce:

- Palabras reservadas: `if`, `else`, `int`, `return`, `void`, `while`.

- Símbolos especiales: `+`, `-`, `*`, `/`, `=`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `;`, `,`, `(`, `)`, `{`, `}`, `[`, `]`.
- Identificadores: cadenas de letras (`a-z`, `A-Z`), no pueden empezar con número.
- Números: secuencias de dígitos enteros.
- Comentarios: de línea `//` o de bloque `/**/`.

Apéndice C: Reglas Lógicas y Tabla de Símbolos

Reglas lógicas utilizadas

1. **Declaración de variables**

   a. Una variable o parámetro no puede ser declarado con tipo `void`.

   b. No se permite declarar una variable con el mismo nombre dos veces dentro del mismo *scope*.
2. **Declaración de funciones**

   a. No se permite declarar múltiples veces una misma función en el mismo *scope*.

   b. El tipo de retorno debe coincidir con el tipo con el que se creó la función.
3. **Llamadas a variables**

   a. Si una variable es referenciada y no ha sido declarada en el *scope*, se lanza un error.
4. **Llamadas a funciones**

   a. Se verifica que una función no regrese `void` cuando se usa dentro de expresiones.

   b. Se comprueba que el número de argumentos y sus tipos coincidan con la declaración.

   c. No se permite asignar un valor a una lista sin especificar índice.
5. **Condicionales**

   a. La condición debe ser una expresión que no contenga `void`.
6. **Return**

   a. El valor de retorno debe coincidir con el tipo de la función.

Tabla de símbolos

1. La tabla de símbolos está dada por un *bucket list* que contiene las siguientes llaves:
   - `location`: línea de posición.

- `linenos`: líneas donde se hace referencia.
- `type`: tipo declarado.
- `Is_array`: indica si es un arreglo.
- `metadata`: información adicional (usada principalmente para parámetros).

2. **Scopes**
   - Se implementan como una pila de *bucket lists* junto con una lista de nombres de *scopes*.
   - Cada nombre de *scope* es clave para acceder y operar en ese contexto.
   - Solo existen dos tipos de *scopes*: el global y uno por cada función.
3. **Funciones de operación sobre la tabla**
   - `st_insert`: inserta un identificador.
   - `st_lookup`: busca un identificador en todos los *scopes*.
   - `st_lookup_current_scope`: busca solo en el *scope* actual.
   - `st_get_type`, `st_get_metadata`: recuperan tipo o metadatos.
   - `st_get_is_array`: comprueba si la variable es un arreglo.

Apéndice D: Gramática en Notación EBNF

Se presenta la gramática formal del lenguaje en notación EBNF, especificando reglas para declaraciones, expresiones, declaraciones de funciones y control de flujo:

```
Unset
1. program -> declaration_list
2. declaration_list -> { declaration }
3. declaration -> var-declaration | fun-declaration ;
4. var-declaration -> type-specifier ID [ [ NUM ] ] ;
5. type-specifier -> int | void ;
```

Tecnológico de Monterrey

```
6. fun-declaration -> type-specifier ID ( params ) compound-stmt
;
7. params -> param-list | void
8. param-list -> param { , param }
9. param -> type-specifier ID [ [] ] ;
10. compound-stmt -> { local-declarations statement-list } ;
11. local-declarations -> { var-declaration } ;
12. statement-list -> { statement } ;
13. statement -> expression-stmt | compound-stmt | selection-stmt
| iteration-stmt | return-stmt ;
14. expression-stmt -> expression ;
15. selection-stmt -> if ( expression ) statement [ else
statement ] ;
16. iteration-stmt -> while ( expression ) statement ;
17. return-stmt -> return [ expression ] ;
18. expression -> var = expression | simple-expression ;
19. var -> ID [ expression ] ;
20. simple-expression -> additive-expression [ relop
additive-expression ] ;
21. relop -> "<=" | "<" | ">" | ">=" | "==" | "!=" ;
22. additive-expression -> term { addop term } ;
23. addop -> "+" | "-" ;
24. term -> factor { mulop factor } ;
25. mulop -> "*" | "/" ;
26. factor -> ( expression ) | var | call | NUM ;
27. call -> ID ( args ) ;
28. args -> expression { , expression } ;
```

La gramática completa está disponible en el archivo EBNF adjunto.