Universidad Nacional de Río Cuarto

Facultad de Ciencias Exactas Físico Químicas y Naturales

Departamento de Computación

Taller de Diseño de Software

Proyecto
Primera entrega

Campregher Bruno
Machuca Franco
Natali Valentino

Analizador Sintáctico y Léxico

1. Introducción

El presente documento describe el desarrollo del analizador léxico y sintáctico correspondiente a la etapa actual del proyecto.

El objetivo principal de esta entrega es implementar las herramientas necesarias para reconocer los elementos básicos del lenguaje (tokens) y verificar la corrección sintáctica de los programas escritos en dicho lenguaje, detectando y reportando errores de forma clara y precisa.

Repositorio del proyecto

El **código fuente completo** del proyecto está disponible en nuestro **repositorio de GitHub**.

La rama `Entrega-15/9` contiene la versión correspondiente a la **entrega actual**, incluyendo el analizador léxico y sintáctico implementado, los archivos de tests y todos los scripts necesarios para ejecutar y verificar el funcionamiento de las distintas etapas.

2. Analizador Léxico

El **parser** recibe los tokens generados por el lexer y se encarga de **verificar que el programa cumpla con la gramática del lenguaje**.

Tokens enviados al parser:

- Palabras reservadas: `program, extern, bool, integer, void, if, then, else, while, return, true, false`
- Operadores aritméticos: `+, -, *, /, %`
- Operadores lógicos: `&&, ||, !`
- Operadores de comparación: `==, <, >`
- Delimitadores: `{, }, (, ), ;, ,`
- Operador de asignación: `=`
- Identificadores: siguen el patrón letra seguida de letras, dígitos o guiones bajos
- Literales enteros: solo números positivos

Símbolos no válidos:

Cualquier carácter que no coincida con una regla del lexer genera un error léxico.

Otras funciones del lexer:

- Ignora comentarios (// o /* */) y espacios en blanco.
- Mantiene el seguimiento de línea con yylineno y columna con yycolumn.
- Imprime en la salida cada token reconocido para seguimiento y depuración.

Asunciones realizadas

Solo se soportan literales enteros positivos

3. Analizador Sintáctico

El parser recibe los tokens generados por el lexer y se encarga de verificar que el programa cumpla con la gramática del lenguaje.

Permite estructurar programas compuestos por declaraciones de variables y métodos, así como bloques de instrucciones que incluyen asignaciones, llamadas a métodos, estructuras de control (if, while) y retornos. Además, gestiona expresiones que combinan identificadores, literales, operadores aritméticos, lógicos y de comparación, respetando la precedencia definida para cada tipo de operador.

Un programa reconocido por el parser comienza con la palabra reservada program y su cuerpo puede contener declaraciones de variables, métodos, ambas o ninguna.

Manejo de errores

Cuando el parser detecta un token que no concuerda con las reglas de la gramática, invoca la función yyerror.

Dicha función se encarga de **reportar de manera precisa la ubicación del error**, indicando tanto la línea como la columna en la que se produjo, lo que facilita la identificación y corrección de inconsistencias en el código fuente.

Asunciones realizadas

- En program: Las declaraciones de variables (sí están presentes) deben preceder a las declaraciones de métodos (si están presentes).
- En blocks: Las declaraciones de variables (sí están presentes) deben preceder a los statements (si están presentes).
- Ambas estructuras permiten la ausencia total o parcial de sus componentes.

4. Pruebas y verificación

Se realizaron pruebas para las etapas de **scanner** y **parser** utilizando archivos de prueba organizados en carpetas:

- `test/scan/valid` y `test/scan/invalid` para la etapa del lexer.
- `test/parser/valid` y `test/parser/invalid` para la etapa del parser.

Cada prueba se encuentra en archivos denominados `test1.ctds`, `test2.ctds`, etc.

- Los **tests válidos** verifican que los analizadores reconozcan correctamente los **tokens** y la **estructura sintáctica** permitida.
- Los **tests inválidos** buscan provocar errores **léxicos** o **sintácticos**, asegurando que el sistema maneje correctamente entradas incorrectas o inesperadas.

A continuación se detallan los tests inválidos junto con el motivo de fallo esperado

Etapa	**Test**	**Motivo**
Scan	test1.ctds	Caracter no reconocido (#).
Parser	test1.ctds	Declaración de variable (z) después de la definición de método (inc).
	test2.ctds	El bloque de un método (inc) contiene una declaración de método (is_true)
	test3.ctds	Declaración de variable (z) después de statements en bloque de método (inc)
	test4.ctds	Declaración de variable(x) sin asignación de valor

Ejecución automatizada de pruebas

Para facilitar la verificación de los casos de prueba, se desarrolló un **script de automatización** (run_tests.sh) que recorre sistemáticamente las carpetas valid e invalid correspondientes a las etapas de **scanner** y **parser**.

El script ejecuta el programa principal (c-tds) con la opción -target scan o -target parse para cada archivo de prueba. Durante la ejecución, se muestra en pantalla el nombre del archivo que se está procesando, permitiendo una verificación manual de los resultados.

Cada prueba genera los archivos de salida correspondientes:

- En la etapa de **scanner**, se produce un archivo .lex que contiene los tokens reconocidos por el lexer.

- En la etapa de **parser**, se generan **ambos archivos**: .lex con los tokens y .sint con el resultado del análisis sintáctico o el mensaje de éxito/error.

Esta automatización asegura que **todos los casos de prueba se ejecuten de manera consistente** y facilita la revisión de los resultados.

Ejecución del script de pruebas

Para ejecutar de manera automática todos los tests, se debe otorgar permisos de ejecución al script y posteriormente ejecutarlo desde la terminal:

```
chmod +x run_tests.sh
```

```
./run_tests.sh
```

5. Ejecución del compilador

El archivo principal del proyecto (main.c) se encarga de coordinar la ejecución del analizador léxico y sintáctico en función de los parámetros recibidos por consola.

Uso del programa

```
programa [-target etapa] archivo.ctds
```

Donde:

- -target scan ejecuta únicamente el analizador léxico y genera como salida un archivo con extensión .lex.

- -target parse ejecuta el analizador sintáctico (por defecto si no se especifica -target) y genera un archivo con extensión .sint.

- archivo.ctds corresponde al programa fuente a compilar. Debe finalizar con la extensión .ctds.

Manejo de archivos

- El compilador valida que el archivo fuente sea correcto (no debe comenzar con `-` y debe terminar en `.ctds`).
- En caso de error al abrir el archivo fuente o crear el archivo de salida, se informa al usuario mediante un mensaje descriptivo.
- El nombre del archivo de salida se deriva automáticamente del archivo de entrada, reemplazando la extensión `.ctds` por `.lex` o `.sint` según corresponda.

Asunciones realizadas

- El parámetro `-target` solo admite los valores `scan` o `parse`.
- Si no se especifica `-target`, la etapa ejecutada por defecto es parse.
- Se asume que el archivo de entrada existe y es accesible para lectura.

6. Proceso de compilación

El proyecto utiliza un archivo `Makefile` para automatizar el proceso de compilación del compilador.

El Makefile define

- El nombre del ejecutable: `c-tds`.
- Los archivos fuente principales: `scanner.l`, `parser.y` y `main.c`.
- Las herramientas utilizadas: `flex` para el analizador léxico, `bison` para el analizador sintáctico y `gcc` como compilador de C.
- Reglas para generar automáticamente los archivos intermedios `lex.yy.c`, `parser.tab.c` y `parser.tab.h`.
- Una regla clean que elimina los binarios y archivos generados (`*.lex`, `*.sint`, etc.).

Esto permite compilar el proyecto de manera sencilla ejecutando: `make` y limpiar los archivos intermedios con: `make clean`.