Proyecto

Taller de Diseño de Software

Conti, Bruno
Gonzalez, Juan Cruz
Vollenweider, Erich

Universidad Nacional de Río Cuarto

Índice

1. Entrega 1: Análizador Sintáctico y Léxico

En esta primera entrega se entregan el analizador léxico (*lexico.l*) y el analizador sintáctico (*parser.y*) para el lenguaje **TDS25**. El equipo trabajó de forma colaborativa: primero se implementó el lexer, luego el parser y finalmente los tests. Algunos tests pasan correctamente, mientras que otros no lo hacen de forma intencional, ya que están diseñados para probar casos fuera del alcance de esta primera entrega (por ejemplo, errores semánticos o características que se implementarán más adelante). En consecuencia, los resultados observados son los esperados para el estado actual del proyecto.

1.1. Repositorio y archivos entregados

- Repositorio del proyecto: `https://github.com/brunocontii/proyecto-compiladores`
- Rama con la primera entrega: `lexico-sintactico`. En esa rama se encuentra todo lo relativo a esta etapa.
- En la rama `lexico-sintactico` hay un `README.md` que explica cómo compilar y ejecutar el **lexer**, el **parser** y los respectivos **tests**.
- Los dos archivos más importantes para esta entrega están dentro de la carpeta `lexico_sintactico/`:
  - `lexer.l` (scanner — Flex)
  - `parsar.y` (parser — Bison)

1.2. Proceso de trabajo

Durante el desarrollo de esta primera entrega seguimos un flujo de trabajo colaborativo y secuencial. Primero nos concentramos en construir el analizador léxico, luego avanzamos con el analizador sintáctico y finalmente preparamos y ejecutamos un conjunto de pruebas para verificar el funcionamiento de ambos componentes. A continuación se describen brevemente cada una de estas fases.

1.3. Análisis léxico (Lexer)

El analizador léxico fue implementado en el archivo `lexer.l` utilizando **Flex**. Entre los aspectos más destacados de su diseño se encuentran:

- Reconocimiento de **palabras reservadas**: `program`, `integer`, `bool`, `void`, `extern`, `return`, `if`, `else`, `then`, `while`, `true`, `false`.
- Soporte para **operadores lógicos y aritméticos**: `&&`, `||`, `!`, `+`, `-`, `*`, `/`, `%`, así como los operadores de comparación `=`, `==`, `<`, `>`.
- Reconocimiento de **delimitadores y símbolos de agrupación**: `( ) { } ; ,`.
- Manejo de **identificadores** (letras, números y guión bajo, iniciando con letra) y **constantes enteras positivas**.
- Ignora espacios en blanco, tabulaciones, saltos de línea y comentarios de línea (`//`) o de bloque (`/* ... */`).

- Estrategia de **depuración**: cada token reconocido se imprime junto a su lexema, lo cual facilita verificar la correcta ejecución.
- **Manejo de errores léxicos**: si se detecta un caracter no válido, se informa en pantalla y el programa finaliza.

El lexer constituye el primer paso del compilador, transformando el código fuente en un flujo de tokens que alimenta al parser en la etapa siguiente.

1.4. Análisis sintáctico (Parser)

El analizador sintáctico fue implementado en el archivo `parser.y` utilizando **Bison**. Sus características principales son:

- Definición de todos los **tokens** producidos por el lexer, garantizando la integración entre ambas fases.
- Especificación de **precedencia y asociatividad** de los operadores mediante directivas de Bison, respetando el orden definido en la especificación del lenguaje (por ejemplo: unario `-` y `!` tienen mayor precedencia que multiplicación/división, etc.).
- La regla inicial `prog` establece la estructura global de un programa: debe comenzar con la palabra reservada `program` y contener entre llaves las declaraciones de variables y/o métodos.
- Soporte para **declaraciones de variables** con asignación y **declaraciones de métodos**, incluyendo parámetros, tipo de retorno y la opción de declarar funciones externas con la palabra clave `extern`.
- Definición de **sentencias** básicas: asignaciones, llamadas a métodos, estructuras de control (`if/then/else`, `while`), sentencias `return`, bloques anidados y sentencias vacías.
- Manejo completo de **expresiones**: literales enteros y booleanos, identificadores, llamadas a funciones, operaciones aritméticas, lógicas, de comparación, así como operadores unarios (menos y negación lógica).
- Inclusión de **bloques** (`{ ... }`) con variables locales y/o secuencias de sentencias.
- **Manejo de errores sintácticos**: se implementó la función `yyerror()`, que informa la línea donde se produjo el error, facilitando la depuración:

  ```
  -> ERROR Sintactico en la linea: N
  ```

En conjunto, el parser valida que la secuencia de tokens producida por el lexer cumpla con la gramática del lenguaje TDS25 y prepara la base para las etapas posteriores.

1.5. Pruebas y verificación

Para esta entrega se preparó un conjunto de casos de prueba en la carpeta `tests/`. Estos casos se dividen en **tests positivos** (que deben pasar correctamente) y **tests negativos** (diseñados para fallar de forma intencional, validando la detección de errores sintácticos).

Tests positivos

- `test1.ctds`
- `test2.ctds`
- `test3.ctds`

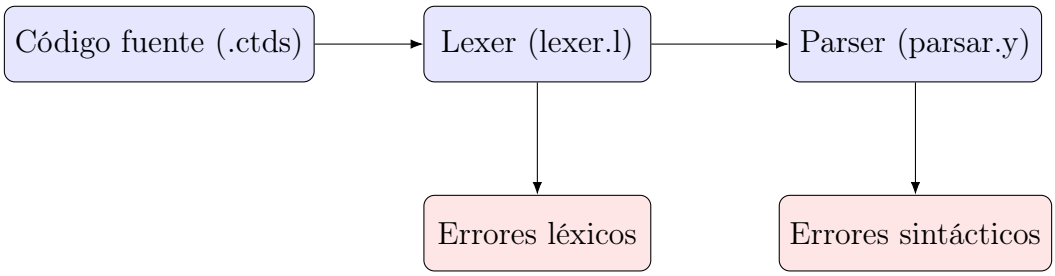
Estos archivos contienen programas válidos en TDS25. El lexer y el parser procesan estos casos sin errores y generan la salida esperada.

Tests negativos (fallan intencionalmente)

- `testneg1.ctds`: contiene una declaración de variable después de un método. Según la gramática, primero deben declararse todas las variables y luego los métodos.
- `testneg2.ctds`: no incluye la definición obligatoria de `program`.
- `testneg3.ctds`: define un método que no posee ni un bloque ni la palabra clave `extern`.
- `testneg4.ctds`: dentro de un bloque aparecen sentencias antes de las declaraciones, lo cual viola la regla de que deben declararse primero las variables.

1.6. Diagrama del flujo de compilación (parcial)

El siguiente diagrama ilustra el flujo de trabajo de las primeras etapas del compilador implementadas en esta entrega:



— Fin de la documentación de la Entrega 1 —