Instituto Tecnológico de Costa Rica
Campus Tecnológico Central de Cartago
Escuela de Ingeniería en Computación

[IC-5701] Compiladores e Intérpretes
Prof. Ing. Erika Marín Schumann
Grupo 01

Proyecto 2

Parser

Sebastián Granados Artavia c.2022117459

Joselyn Vanessa Montero Rodríguez c.20221366356

Jose Pablo Barquero Díaz c.2022119938

II Semestre

Tabla de Contenidos

Introducción

Este documento presenta el desarrollo y análisis de un parser implementado en el lenguaje de programación Java, utilizando CUP como herramienta para la generación del analizador sintáctico. Este parser usa como base el scanner que se realizó en el proyecto anterior para el curso de Compiladores e Intérpretes. Su principal función es procesar y analizar archivos escritos en el lenguaje C, permitiendo la interpretación y validación de su estructura sintáctica. El documento detalla tanto el desarrollo del parser como sus componentes principales, explicando las decisiones de diseño tomadas durante el proceso. Además, se incluyen los resultados obtenidos al aplicar el parser a varios archivos fuente en C, acompañados de un análisis crítico de su desempeño y las áreas de mejora para las etapas futuras del proyecto. Este enfoque proporciona una visión integral del parser y su integración en el proyecto general, destacando su rol fundamental en la creación de un compilador funcional.

Estrategia de Solución

A continuación, ciertos aspectos relacionados a la estrategia utilizada por el equipo para completar el trabajo solicitado:

1. La implementación del parser se construyó sobre la base del código del scanner previamente desarrollado, lo que permitió mantener la elección de las herramientas Java y Maven, que ya estaban configuradas para el proyecto.
2. Se eligió trabajar con CUP debido a su capacidad para trabajar eficazmente con los tokens generados por el scanner, y también porque facilita la integración con el proceso general de compilación en Java.
3. El parser recibe los tokens generados por el scanner (que utiliza JFlex), y luego evalúa si las secuencias de tokens forman estructuras válidas según la gramática definida. Esto permite que el parser se base directamente en el archivo de salida del análisis léxico para continuar el proceso de compilación.
4. Se definen los terminales y no terminales, que corresponden a las diversas reglas gramaticales que el parser debe reconocer y procesar en el lenguaje C.

5. Para asegurar que las expresiones se evalúan en el orden correcto, se define la precedencia y la asociatividad de los operadores, tales como AND, OR, y los operadores aritméticos y relacionales.
6. Cada regla describe cómo se componen los diferentes elementos del lenguaje. La regla programa ::= globales define la estructura básica de un programa, y las reglas más complejas describen cómo se manejan las instrucciones, expresiones y estructuras de control.
7. El parser está diseñado para detectar y reportar errores sintácticos de forma detallada. Si el código fuente no sigue la gramática esperada, se generarán mensajes de error que indican el tipo de problema y su ubicación en el código fuente. Esto se logra mediante el uso de métodos como syntax_error y unrecovered_syntax_error.
8. El parser genera objetos de tipo Symbol, que contienen información sobre el token actual, como su tipo y valor, así como su posición en el código fuente. Esta información es crucial para las fases posteriores del compilador.
9. El archivo de prueba input.c se creó para cubrir una amplia variedad de casos, incluyendo variables globales o constantes y funciones que pueden contener expresiones o estructuras de control, entre otros, esto con el fin de validar el funcionamiento del parser.
10. Se incluyeron comentarios detallados en el código y en las reglas sintácticas para documentar decisiones.

Análisis de Resultados

1. Detección de errores sintácticos básicos
    a. El parser identifica y reporta la mayoría de los errores sintácticos, incluyendo constantes no inicializadas, variables sin tipo y declaraciones de variables void. Sin embargo, la precisión de las ubicaciones de los errores no es perfecta, especialmente en casos como la falta de punto y coma, lo que puede generar confusión en el usuario.
    b. El estado se considera del **90%**

2. Manejo de declaraciones globales y constantes

a. Se logró la detección de constantes no inicializadas y variables declaradas incorrectamente. No obstante, en ciertos casos, la identificación de errores en la inicialización de constantes puede resultar menos precisa en cuanto a la línea exacta reportada.
b. El estado se considera del **95%**

3. Detección de errores en funciones
    a. El parser identifica errores en las funciones, como declaraciones de variables sin punto y coma, asignaciones incompletas, y parámetros sin identificador. Sin embargo, en algunos casos de estructuras, los mensajes de error pueden indicar información que no es suficientemente específica.
    b. El estado se considera del **80%**

4. Validación de parámetros de función
    a. Aunque se detectan errores en las funciones que tienen parámetros sin tipo o identificador, los mensajes generados no son siempre claros en cuanto a la causa del error. La precisión de los reportes de línea y columna también presenta limitaciones.
    b. El estado se considera del **90%**

5. Estructuras de control (if, while, for, switch)
    a. El parser detecta errores relacionados con múltiples cláusulas else, faltas de paréntesis en expresiones de control, bucles while con expresiones faltantes y switch con uno o varios cases así como con defaults. Sin embargo, hay dificultades en la detección precisa de ciertos errores y en la claridad de los mensajes generados, como la indicación de un error por falta de paréntesis en las declaraciones de función. Falla detectando los errores de cases en la estructura del switch, tampoco verifica que el switch sólo tenga un default.
    b. El estado se considera del **90%**

6. Precedencia y asociatividad de operadores

a. La precedencia y la asociatividad de los operadores se definieron correctamente en el parser, garantizando un análisis preciso de las expresiones complejas y evitando ambigüedades.
b. El estado se considera del **100%**

7. Manejo de errores de gramática
    a. Se implementaron métodos para detectar errores gramaticales y reportar mensajes con información de tipo de error y ubicación. Sin embargo, cuando varios errores se encuentran en líneas consecutivas, solo se reporta el primero, lo cual puede limitar la utilidad del análisis.
    b. El estado se considera del **85%**

8. Integración de scanner y parser
    a. La integración del scanner (JFlex) y el parser (CUP) se realizó sin inconvenientes, asegurando una comunicación fluida entre las fases del análisis.
    b. El estado se considera del **100%**

9. Pruebas con archivo .c
    a. Se creó un archivo de prueba para validar el correcto funcionamiento del *Parser* y en este se cubrieron todos los casos posibles.
    b. El estado se considera del **100%**

10. Manejo de producción y recuperación de errores
    a. Se implementaron métodos para manejar errores con syntax_error y unrecovered_syntax_error, lo que permite la recuperación parcial de algunos errores. Además se creó la función report_error que ayuda a especificar el mensaje de error para cada caso.
    b. El estado se considera del **95%**

11. Documentación
    a. Se creó este documento y el código se encuentra con varios comentarios que ayudan al entendimiento del mismo.
    b. El estado se considera del **100%**

Lecciones aprendidas

1. Aprendimos a configurar e integrar CUP con JFlex, lo que nos permitió construir un analizador léxico y sintáctico de forma más estructurada en el entorno de Java.
2. La investigación sobre las reglas del lenguaje C nos permitió aplicar conceptos teóricos a problemas reales y adaptarlos al diseño del parser.
3. Nos familiarizamos con la estructura de gramáticas en CUP y con la definición de reglas sintácticas, logrando entender mejor cómo se construyen y procesan árboles de análisis.
4. Profundizamos en las reglas de precedencia y asociaciones en CUP, lo que fue clave para gestionar correctamente operaciones complejas y evitar problemas de ambigüedad en el análisis.
5. Aprendimos la importancia de ser detallistas y revisar cuidadosamente el código, ya que un pequeño error en una regla gramatical puede afectar el rendimiento completo del parser.
6. Trabajar en equipo nos permitió mejorar la comunicación y colaboración, asegurando que cada miembro del grupo contribuyera de manera efectiva al desarrollo del proyecto, optimizando el proceso de planificación y ejecución.
7. Finalmente, experimentamos de primera mano el proceso de construcción de un compilador básico, desde la identificación de tokens en JFlex hasta el procesamiento de la gramática en CUP, dándonos una comprensión completa de cómo funcionan los compiladores.

Casos de pruebas

Globales o constantes

input.c:

```
const int a;      // error: debe inicializar la constante
int g;
y;                // error: no especifica tipo
int z, t, y, y;
```

```
void x;          //deberia de dar error pues no se pueden declarar variables void
int h            // error: falta punto y coma
const char b = lqs;
```

Resultados Esperados

-Las variables y las constantes que están bien definidas debería ignorarlas.

- Las constantes que no se inicializan debe marcarlas como error.

- Las variables y constantes que no tiene punto y coma al final debe marcarlas como error.

- Las variables a las que no se les especifica un tipo o se les asigna tipo void debe marcar como error.

Resultados Obtenidos

```
Error sintáctico en línea 1, columna 12: ;      Error: falta inicializar la constante.
Error sintáctico en línea 3, columna 1: y       Error: debe especificar el tipo de variable.
Error sintáctico en línea 5, columna 7: ;       Error: no puede declarar variables void.
Error sintáctico en línea 7, columna 1: const
```

Conclusión

El parser identifica los errores esperados, pero no señala con precisión la línea y columna exactas cuando falta un punto y coma al final de una declaración. Esto se debe a que el parser espera el siguiente token para decidir, lo que hace que el error se indique en la línea siguiente. Además, el mensaje de error resultante no aclara que el problema es un punto y coma faltante, lo cual es confuso para el usuario. Esta imprecisión es particularmente problemática cuando hay otro error en la línea siguiente, ya que solo se detecta uno de ellos, dificultando el análisis y la corrección del código.

Funciones

input.c:

```
int funcion1(){
        int k;
        int g           //error por no tener ;
        a = a * c       //error por no tener ;
```

```
        a = a * funcion(8);
        if (y>=0) {
                x= a+b;
        }
        else {
                x= a+b;
        }
        else {              //deberia dar error por doble uso de else
        x= a+b;
        }

        if x==3 {        //deberia dar error por expresion fuera de parentesis
                x= a+b;
        }
        else {
                x--;
        }
}

int funcion4(int, int) {            //no se permite solo los tipos
        if ( (x<7) && (x>2) ){
                x= g+6;
        }
        while ( ){        //debe dar error porque falta expresion
                x= a+b;
                break;
        }

        for (x=0; ) {    //error pues hay solamente una expresion
                x= a+b;
                break;
```

```
        }
}
```

Resultados Esperados

- Las funciones que contengan expresiones, o estructuras de control, instrucciones como read,
write, break, continue bien definidas debería ignorarlas.
- Si los parámetros de la función no especifican su tipo o su identificador debe dar error.
- Las declaraciones de variables o constantes son incorrectas debe marcarlas como error.
- El if que tenga más de un else debe marcarse como error.
- Si la expresión del if no está entre paréntesis debe dar error.
- El while que no tenga definida su expresión de condición debe dar error.
- El for debe tener una asignación y dos expresiones, si no debe marcarse como error.

Resultados Obtenidos

Error sintáctico en línea 13, columna 1: a
Error sintáctico en línea 19, columna 2: else
Error sintáctico en línea 22, columna 2: else Error: falta paréntesis en la declaración de función.
Error sintáctico en línea 27, columna 5: x Error: falta paréntesis en la expresión 'if'.
Error: falta paréntesis en la declaración de función.
Error sintáctico en línea 36, columna 17: ,
Error sintáctico en línea 42, columna 10:) Error: falta la expresión del 'while'.
Error sintáctico en línea 47, columna 12:) Error: error en la estructura del bucle 'for'.
Error: falta paréntesis en la declaración de función.

Conclusión

El parser tiene limitaciones en la precisión de los mensajes de error y en la identificación de ciertos errores esperados, como la falta de punto y coma o la definición incompleta de parámetros de función. Aunque reporta algunos errores sintácticos, estos mensajes carecen de detalles suficientes para ayudar al usuario a comprender y corregir los problemas. Además el parser hace una confusión con algunas producciones donde erróneamente les suma un error que que señala que falta paréntesis en la declaración de función.

Manual de usuario

Para poder utilizar el programa ingrese el código escrito en C en el archivo ***input.c***, a continuación ejecute el archivo ***Main.java.*** Esto le desplegará todos los errores sintácticos encontrados, al final debería ver el mensaje “Parsing completed successfully.”.

Bitácora

Fecha	Descripción	Participantes
14/10/2024	Reunión inicial para la organización del proyecto	Todos
19/10/2024	Reunión de seguimiento, se empezó a trabajar en Miro	Todos
24/10/2024	Reunión de seguimiento, se afinó la definición de gramática en Miro	Todos
28/10/2024	Reunión para atender problema con los símbolos	Todos
01/11/2024	Reunión para pruebas con no terminales	Todos
04/11/2024	Reunión para ver mensajes de errores del archivo .cup	Todos
06/11/2024	Reunión de cierre del proyecto	Todos

Bibliografía

CUP User’s Manual. (s. f.).

https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html#intro

CUP. (s. f.). http://www2.cs.tum.edu/projects/cup/docs.php