Instituto Tecnológico de Costa Rica

Unidad de Computación

Compiladores E Intérpretes

Verano I 2024

Segundo Proyecto

Profesor

Allan Rodriguez Davila

Estudiantes

Deivid Matute Guerrero

Joshua Sancho Burgos

Centro Académico de Limón

06 de Enero del 2024

Manual de Usuario

Creación del Lexer:

Para programar el lexer es necesario instalar las librerías JFlex y CUP. Una vez instaladas e implementadas en el proyecto se procede a crear un archivo en formato ".jflex" ,".flex", o ".lex" que son los formatos compatibles con la librería JFlex. Este archivo se divide en 3 secciones, el código de usuario (donde se realizan configuraciones, se declara el paquete y se realizan los imports), opciones y declaraciones (donde se declaran los macros y estados) y las reglas léxicas (donde se especifican los tokens que regresará el lexer). Es importante que la configuración "%cup" esté incluida para ser compatible con el parser.

Creación del Parser:

Para programar el parser es necesario instalar e implementar la librería CUP. Una vez terminada la programación del lexer se procede a programar el parser. Primero se realizan los ajustes necesarios para la compatibilidad con JFlex (imports y scanner), luego se escribe código para el parser (manejo de errores) y luego se declaran los símbolos terminales y no terminales. Los terminales deben ser los mismos que se declararon en el lexer, si no, se generará un error, al no existir tal símbolo en dado caso. Cuando se tiene certeza de haber agregado todos los símbolos no terminales se procede a programar los símbolos no terminales donde se debe ajustar la gramática para el análisis sintáctico y asegurarse que entren todas las posibles formas de generar un programa. Para generar el parser es necesario que exista un símbolo no terminal que de entrada al programa. El parser tiene la opción de elevar los resultados que encuentra con la función RESULT, lo cual resulta muy útil para almacenar y controlar la información encontrada y la tabla de símbolos en esta segunda parte.

Generación de Parser y Lexer:

Para generar el lexer y el parser se tienen 2 opciones.

La primera es ejecutar la función "GenerarLexerParser()" en el archivo "App.java'. Esta función elimina los archivos "lexer.java, "parser.java" y "sym.java" si son archivos existentes en el sistema, en caso contrario, los genera si se encuentran los archivos "lexer.flex" y "parser.cup" en las carpetas correspondientes y los mueve de su localización hacia la carpeta "ParserLexer"; en caso de que estos archivos no existan

(en la última actualización del proyecto, estos archivos sí existen), se deberá localizar el archivo MainJFlexCup.java y comentar la línea 4, que contiene lo siguiente: “import ParserLexer.lexer;”, además de comentar la función “LexerTest” completa, ya que estas partes del código requieren de la existencia del archivo “lexer.java”.

La otra forma es generarlos manualmente por medio de los comandos del manejador de paquetes Maven, con “mvn jflex:generate” para generar a “lexer.java” y “mvn cup:generate” para generar a “parser.java” y “sym.java”, sin embargo, no se recomienda usar esta opción, ya que localiza los archivos en target y no los mueve a la carpeta “ParserLexer”, por lo que se pueden ocasionar algunas incompatibilidades.

Tabla de Símbolos:

Para crear la tabla de símbolos se realizaron tres clases:

- TabSymbol: Unidad mínima de la tabla de símbolos, donde se crean los símbolos de arreglos, variables y parámetros. Se almacena su nombre, tipo (int, float, bool, char, ...), clase (arr, var, param) y en el caso de los arreglos, su tamaño. Cuando estos símbolos son declarados el parser este las añade a la tabla de símbolos correspondiente.
- SymbolTable: Cada vez que el parser encuentra una declaración de función se genera una instancia de la tabla de símbolos. En esta se almacenan características de la función como nombre, tipo de retorno y sus símbolos. La clase posee funciones para agregar, obtener e imprimir símbolos.
- SymbolTableManager: Esta clase es un administrador de tablas de símbolos. Permite llevar control de las tablas, obtenerlas e imprimir sus valores.

Pruebas de funcionalidad

Para probar el parser con sus no terminales y nuevas funciones y la funcionalidad de la tabla de símbolos se realizó el siguiente script:

```
function int main() {
   local int ab|
}
/_ Comentario Multilinea _/
@ Comentario simple
function int func_a(int b[alfa(a, b, 1)], bool y, float x) {
  local int i <= alfa() |
  local char o|
  local bool bip <= alfa(a, b, c[0])|
  local int carr[] <= [1, 12, 300]|
  local char hiss[7*r+1.0+7] <= ['a', 'a', 'd']|
  a <= 4|
  b[4] <= 8|
  if(a == 9) {
      a <= 0|
  }
  elif (a =< 8) {
      a <= 10|
  }
  elif (a =< 11) {
      if(b =< 10) {
         b <= 11|
      }
  } else {
    a <= 10|
  }
  read(a)|
  print(a+b)|
  local string var_str <= "Hello"|
  return a|
  do {
     ++a |
  } until ( a => 8) |
    for (i <= 0 | ++i | i =< 10) {
         i <= i + 1|
    }
    local int xyz <= (x + y + z) ** (alfa(a, b, c) - ---5 / ++y)|
}
```

El script anterior considera la mayoría de casos comunes y atípicos que se pueden observar en la programación de un lenguaje imperativo básico. En este se encuentran declaraciones de funciones, parámetros, arrays y variables (estos dos últimos inicializados o no), bloques de código, ciclos, expresiones y sentencias.

Luego, mediante la función “ParserTest” se probó el parser en general con todas sus funcionalidades.

Anexo 1: Test de Análisis Sintáctico

```
ANÁLISIS SINTÁCTICO
variable::int::ab
function::int::main
parameter::int::b::alfa(id=a, id=b, int=1)
parameter::bool::y
parameter::float::x
variable::int::i::alfa()
variable::char::o
variable::bool::bip::alfa(id=a, id=b, id=c[int=0])
variable::string::var_str::string=Hello
variable::int::xyz::(id=x+id=y+id=z)**(alfa(id=a, id=b, id=c)---int=-5/++id=y)
function::int::func_a
```

En el anexo 1 se puede observar cómo el parser obtiene los valores de las funciones, variables, parámetros y arreglos. Como estos son los valores que serán insertados en la tabla de símbolos se imprimen en consola para verificar posteriormente sus valores.

Anexo 2: Test de funcionalidad de tabla de símbolos

```
Symbol Tables:
*-----------------------------------------------------*
Function: func_a
Return Type: int
Symbols:
  carr: Type: int, Kind: array, Size: int=3
  hiss: Type: char, Kind: array, Size: int=7*id=r+float=1.0+int=7
  var_str: Type: string, Kind: variable
  b: Type: int, Kind: parameter
  x: Type: float, Kind: parameter
  bip: Type: bool, Kind: variable
  xyz: Type: int, Kind: variable
  y: Type: bool, Kind: parameter
  i: Type: int, Kind: variable
  o: Type: char, Kind: variable
*-----------------------------------------------------*
Function: main
Return Type: int
Symbols:
  ab: Type: int, Kind: variable
```

En el anexo 2 se puede observar cómo cada función es una tabla de símbolos y cómo estas muestran sus características (nombre, tipo, clase y tamaño en caso de ser arreglo).

Anexo 3: Test de funcionalidad de manejo de errores irrecuperables

```
function int 12func_a(int b[alfa(a, b, 1)], bool y, float x) {
```

```
ANÁLISIS SINTÁCTICO
variable::int::ab
function::int::main
syntax_error: Error sintáctico '12' en la línea: 7, columna: 13
unrecovered_syntax_error: Error sintáctico inrecuperable '12' en la línea: 7, columna: 13
```

En el anexo 3 se puede observar que la línea tiene un 12 antes del identificador, haciendo que el parser se detenga al no poder continuar con la operación luego de reportar el error como irrecuperable.

Anexo 4: Test de funcionalidad de manejo de errores recuperables

```
ANÁLISIS SINTÁCTICO
syntax_error: Error sintáctico 'int' en la línea: 1, columna: 10
function::int::main
parameter::int::b::alfa(id=a, id=b, int=1)
parameter::bool::y
parameter::float::x
variable::int::i::alfa()
variable::char::o
variable::bool::bip::alfa(id=a, id=b, id=c[int=0])
variable::string::var_str::string=Hello
variable::int::xyz::(id=x+id=y+id=z)**(alfa(id=a, id=b, id=c)---int=-5/++id=y)
function::int::func_a

Symbol Tables:
*-----------------------------------------------------*
Function: func_a
Return Type: int
Symbols:
  carr: Type: int, Kind: array, Size: int=3
  hiss: Type: char, Kind: array, Size: int=7*id=r+float=1.0+int=7
  var_str: Type: string, Kind: variable
  b: Type: int, Kind: parameter
  x: Type: float, Kind: parameter
  bip: Type: bool, Kind: variable
  xyz: Type: int, Kind: variable
  y: Type: bool, Kind: parameter
  i: Type: int, Kind: variable
  o: Type: char, Kind: variable
*-----------------------------------------------------*
Function: main
Return Type: int
Symbols:

Process finished with exit code 0
```

En el anexo 4 se puede observar que el error fue recuperable, reportado, y que el parsing continuó en las demás funciones sin embargo la declaración errónea no fue incluida en la tabla de símbolos.

Descripción del Problema para la Fase de Parser:

Contexto:

En el ámbito del desarrollo de compiladores, la fase de análisis sintáctico es esencial para comprender la estructura y la gramática del código fuente. En este escenario, se emplea CUP para la generación del parser (analizador sintáctico) que permitirá la interpretación correcta de la secuencia de tokens previamente generada por el lexer desarrollado con JFlex.

Problema:

El desafío radica en implementar un parser eficiente utilizando CUP que sea capaz de analizar la secuencia de tokens generada por el lexer y construir el árbol de análisis sintáctico correspondiente al lenguaje de programación objetivo. Este proceso implica la identificación y validación de la estructura gramatical, garantizando la correcta organización de las instrucciones y la interpretación coherente del código fuente.

Desafíos Específicos:

- Definición de Reglas Gramaticales: Enumera y define las reglas gramaticales que definen la sintaxis del lenguaje. Especifica la jerarquía de operadores, la precedencia y la asociatividad para evitar ambigüedades.
- Manejo de Tokens: Integra las reglas de producción de CUP con los tokens generados por el lexer. Asegúrate de que el parser reconozca y utilice correctamente cada tipo de token en la construcción del árbol sintáctico.
- Construcción del Árbol Sintáctico: Implementa las acciones semánticas necesarias para construir el árbol sintáctico de manera coherente. Cada regla gramatical debe contribuir a la estructura general del árbol, reflejando la organización del código fuente.
- Manejo de Ambigüedades y Conflictos: Resuelve posibles ambigüedades y conflictos en la gramática mediante la especificación adecuada de reglas y la utilización de precedencia y asociatividad. Asegúrate de que el parser tome decisiones coherentes en situaciones conflictivas.
- Gestión de Errores Sintácticos: Implementa mecanismos para reportar errores sintácticos de manera clara. Indica la posición en la que se detectó el error y describe la naturaleza del mismo para facilitar la depuración.

- Objetivo Final: La solución al problema debe proporcionar un parser robusto y eficiente que interprete correctamente la estructura sintáctica del código fuente del lenguaje objetivo. Al trabajar conjuntamente con el lexer, esta herramienta permitirá la construcción de un compilador completo capaz de procesar y comprender el código fuente de manera precisa y coherente.

Diseño del programa

Especificaciones Iniciales:

- Identificación de las reglas gramaticales, acciones semánticas y estructura del árbol sintáctico del lenguaje de programación objetivo.
- Definición de reglas en CUP para reconocer y procesar la secuencia de tokens generada por el lexer.

Implementación del Parser con CUP:

- Utilización del algoritmo LALR (Look-Ahead Left-to-Right) implementado por CUP para el análisis sintáctico.
- Creación de un archivo ".cup" con las reglas gramaticales y especificaciones del árbol sintáctico.
- Desarrollo de acciones semánticas para construir el árbol sintáctico y validar la estructura del código fuente.
- Integración del lexer de JFlex con el analizador sintáctico de CUP para la interpretación coherente del código.

Generación de la función ParserTest:

- Aceptación como argumento de la ruta del archivo que contiene el código fuente a analizar.
- Validación de que el parser interprete correctamente la secuencia de tokens, construyendo el árbol sintáctico de manera precisa.
- Generación de mensajes de error descriptivos al detectar un error sintáctico.
- Lectura del archivo para mostrar la estructura de la tabla de símbolos en consola.

Manejo de Errores:

- Generación de mensajes de error claros y descriptivos utilizando técnicas de análisis de contexto y reglas gramaticales.
- Inclusión de información sobre la posición en la que se detectó el error y una descripción de su naturaleza.

Pruebas y Depuración:

- Creación de conjuntos de pruebas exhaustivos que aborden diferentes aspectos del lenguaje y su sintaxis.

- Ejecución de pruebas para verificar la correcta interpretación y construcción del árbol sintáctico.
- Depuración de errores y ajuste de las reglas gramaticales y acciones semánticas según sea necesario.

Documentación:

- Elaboración de documentación detallada que explique el funcionamiento del parser y las reglas gramaticales.
- Inclusión de ejemplos de uso y casos de prueba en la documentación.

Integración en un Compilador:

- Integración del lexer y del parser en un compilador completo capaz de analizar y transformar el código fuente en una representación ejecutable.
- Garantía de la coherencia entre el análisis léxico y sintáctico para asegurar la comprensión y ejecución correcta del código fuente.

Librerías usadas

- JFlex: Es un generador de analizadores léxicos (conocidos también como escáner) para Java, escrito en Java. Este toma como entrada un conjunto de expresiones regulares y acciones correspondientes. Estos se basan en autómatas finitos deterministas (DFAs), los cuales son muy veloces y no requieren de un backtracking pesado. Está diseñado para trabajar con el generador de parsers LALR CUP.
- CUP: Es un generador de parsers LALR. En este se pueden especificar los símbolos de una gramática propia, así como producciones y acciones. Este funciona especialmente bien con JFlex.

Análisis de resultados

Tarea / Requerimiento	***Estado***	***Observaciones***
Parser	Completado	
Acepta símbolo inicial	Completado	
Acepta definiciones de funciones	Completado	
Acepta declaraciones de variables y arreglos	Completado	
Acepta expresiones booleanas	Completado	
Acepta expresiones aritméticas	Completado	
Acepta sentencias de retorno	Completado	
Acepta sentencias de lectura y escritura	Completado	
Acepta sentencias condicionales (if-elif-else)	Completado	
Acepta bucles (for, do-until)	Completado	
Gramática	Completado	
Define reglas para estructuras de control	Completado	
Define reglas para expresiones booleanas	Completado	
Define reglas para expresiones aritméticas	Completado	
Define reglas para sentencias de retorno	Completado	
Define reglas para sentencias de lectura y escritura	Completado	
Manejo de Errores	Completado	
Genera mensajes de error claros y descriptivos	Completado	

Indica la posición y naturaleza de los errores	Completado	
Maneja errores sintácticos de manera precisa	Completado	
Tabla de Símbolos	Completado	
Almacena información sobre funciones, variables, parámetros y arreglos	Completado	
Gestiona el ámbito de las variables	Completado	
Proporciona acceso rápido y eficiente a la información de los símbolos	Completado	
Actualiza la tabla de símbolos durante el análisis sintáctico	Completado	
Administrador de Tablas de Símbolos	Completado	
Implementa operaciones para agregar símbolos	Completado	
Implementa operaciones para buscar símbolos	Completado	
Implementa operaciones para actualizar información de símbolos	Completado	
Gestiona eficientemente el ámbito de las variables	Completado	

Bitácora

Repositorio de Github:

https://github.com/JoshSan14/Xmas-Compiler