Instituto Politécnico Nacional
Escuela Superior de Cómputo

Compiladores

Unidad 2: Análisis Léxico y Sintáctico

Proyecto: Desarrollo de un Analizador Sintáctico Predictivo para el Lenguaje

Nombre del Alumno:

Díaz Gutiérrez Luis

García Solís Marcos Antonio

Perez Lorenzana Axel Eduardo

Roldán Pérez Ricardo

Profesor: Dr. Mario Alberto Gutiérrez Mondragón

Grupo: 5CV3

Introducción

El análisis sintáctico es una fase crítica dentro del proceso de compilación, ya que permite validar la estructura de los programas fuente conforme a las reglas gramaticales de un lenguaje. En este contexto, el presente proyecto se enfoca en diseñar e implementar un analizador sintáctico predictivo para MiniC, un lenguaje simplificado inspirado en C, utilizado con fines educativos para ilustrar conceptos fundamentales de compiladores.

La motivación principal de este trabajo es aplicar los conocimientos adquiridos sobre gramáticas libres de contexto, transformaciones gramaticales y técnicas de parsing predictivo, para construir un sistema funcional que pueda procesar entradas del lenguaje MiniC y determinar su validez estructural. Esto refuerza la comprensión del diseño de compiladores, particularmente en las fases de análisis sintáctico y diseño gramatical.

La importancia del análisis sintáctico radica en que permite identificar errores estructurales antes de la ejecución del programa, asegurando que las construcciones como declaraciones, asignaciones y estructuras de control sigan una sintaxis válida. Además, el diseño de una gramática clara y precisa es esencial para facilitar la implementación de parsers eficientes.

El alcance de este proyecto incluye la definición de una gramática representativa del lenguaje MiniC, su transformación a una forma apta para parsers LL(1), el desarrollo de un analizador en Python, la generación de árboles de derivación para cadenas válidas, y la validación de múltiples entradas. También se realiza una comparación con parsers LR, explorando cuándo es necesario emplear técnicas más potentes debido a la ambigüedad o complejidad de la gramática.

Este documento describe de manera estructurada cada una de las fases del proyecto, así como los resultados obtenidos, reflexionando sobre las limitaciones y aprendizajes clave derivados de la implementación.

Metodología

Especificación del Lenguaje MiniC

Nombre y Propósito:
MiniC es un subconjunto del lenguaje de programación C, diseñado con fines didácticos para enseñar conceptos fundamentales de compiladores, en especial en el análisis sintáctico. Su propósito es permitir la validación y análisis de programas que incluyen declaraciones, asignaciones y estructuras de control básicas como if, while, y for.

Características Principales del Lenguaje:

- Tipado básico: int, float, char
- Estructuras de control: if, else, while, for
- Operadores aritméticos: +, -, *, /

- Operadores relacionales: <, >, ==, !=
- Delimitadores: ;, {}, ()
- Identificadores y literales

Programa de ejemplo:

```
int main() {
   int a;
   a = 5;
   if (a > 3) {
      a = a + 1;
   }
}
```

Este fragmento es válido en MiniC y contiene declaraciones, asignaciones, una condición y un bloque if.

Gramática Final en BNF

Versión revisada (BNF):

<programa> ::= <bloque>

<bloque> ::= "{" <lista_sentencias> "}"

<lista_sentencias> ::= <sentencia> <lista_sentencias> | ε

<sentencia> ::= <declaracion> ";"
| <asignacion> ";"
| <condicional>
| <bucle>

<declaracion> ::= <tipo> <id>

tipo ::= "int" | "float" | "char"

<asignacion> ::= <id> "=" <expresion>

<condicional> ::= "if" "(" <expresion> ")" <bloque> ["else" <bloque>]

<bucle> ::= "while" "(" <expresion> ")" <bloque>

<expresion> ::= <expresion> <op> <expresion> | <id> | <num>

<op> ::= "+" | "-" | "*" | "/" | ">" | "<" | "==" | "!="

<id> ::= letra (letra | digito)*

<num> ::= digito+

Transformaciones realizadas

- **Eliminación de recursividad por la izquierda:**
  La producción original de <expresion> era recursiva por la izquierda. Se reformuló para ser apta para LL(1), separando operaciones y operandos.
- **Factorización:**
  Se agruparon alternativas comunes en producciones como <op> para reducir ambigüedad.
- **Limpieza:**
  Se eliminaron reglas redundantes y símbolos no productivos en la gramática.
- **Justificación de Parser LR:**
  Aunque la gramática fue adaptada a LL(1), se presentan ambigüedades naturales (como las estructuras if-else anidadas). Por tanto, se justifica el uso de un parser LR para análisis robusto.

Resultados

Árboles de derivación (Fase 3)

Se generaron derivaciones izquierda y derecha, así como árboles de derivación para las siguientes cadenas válidas usando Graphviz:

1. **Cadena 1:** int a;
2. **Cadena 2:** a = 5;
3. **Cadena 3:** if (a > 3) { a = a + 1; }
4. **Cadena 4:** int a; a = 5; while (a < 10) { a = a + 1; }
5. **Cadena 5:** if (a > 3) { a = a + 1; } else { a = 0; }

Los árboles fueron generados utilizando estructuras jerárquicas que representan cada producción aplicada de forma secuencial. Las derivaciones ilustran cómo una cadena se forma desde el símbolo inicial utilizando las reglas de la gramática, lo cual validó la consistencia de la misma.

Análisis sintáctico predictivo (Fase 4)

Se implementó un parser LL(1) en Python que:

- Calcula los conjuntos FIRST y FOLLOW
- Genera automáticamente la tabla LL(1)
- Verifica la validez de cadenas mediante la simulación del análisis descendente

Tabla LL(1) generada: Se incluyó como estructura en código y representada como matriz en pruebas.

Cadenas válidas utilizadas:

- int x;
- x = 2;
- if (x > 1) { x = x - 1; }
- while (x < 10) { x = x + 1; }

Cadenas inválidas utilizadas:

- int = 4; (Error de sintaxis: identificador esperado)
- if x > 1 { x = x - 1; } (Error de paréntesis faltante)
- x = ; (Error: expresión incompleta)

Análisis con Parser LR (Fase 5)

Se implementó un parser LR utilizando la herramienta PLY. Se definieron las producciones y acciones de análisis LR(1). Se observaron los siguientes aspectos:

- Mayor capacidad para resolver ambigüedades como el else colgante
- Análisis más robusto frente a entradas no predecibles con LL(1)

Cadenas analizadas con LR:

- Se probaron las mismas cadenas válidas e inválidas
- Todas las cadenas válidas fueron aceptadas correctamente
- Las inválidas generaron mensajes de error específicos

Conclusiones

El desarrollo de este proyecto permitió consolidar los conocimientos sobre gramáticas libres de contexto, técnicas de análisis sintáctico y construcción de parsers. A través de las distintas fases, se comprendió la complejidad que implica transformar una gramática para que sea apta para un análisis LL(1), lo cual resulta en muchos casos poco viable sin recurrir a restricciones artificiales que comprometen la expresividad del lenguaje.

Una de las principales lecciones aprendidas fue identificar las limitaciones prácticas de los parsers predictivos. Si bien son eficientes y relativamente fáciles de implementar, no son adecuados para gramáticas ambiguas o con múltiples reglas que inician con símbolos comunes. En estos casos, el uso de parsers LR se vuelve una solución más robusta, al permitir mayor flexibilidad y manejo de estructuras más complejas sin modificar drásticamente la gramática original.

Además, la implementación del parser utilizando la biblioteca PLY permitió validar con éxito una gran variedad de cadenas, tanto válidas como inválidas, y observar el proceso de reducción y desplazamiento característico de los parsers LR. Esto proporcionó una experiencia práctica sobre cómo se construye un analizador sintáctico profesional y reforzó la importancia del diseño gramatical desde el inicio del desarrollo del compilador.

Otro aprendizaje valioso fue la generación de árboles de derivación y la visualización del análisis sintáctico. Esto facilitó enormemente la comprensión de las estructuras internas del lenguaje, y fue fundamental para la validación y corrección del comportamiento del parser.

En resumen, este proyecto proporcionó una experiencia integral en el diseño y evaluación de gramáticas, así como en la implementación de analizadores sintácticos predictivos y LR, fortaleciendo las habilidades esenciales para el diseño de lenguajes y compiladores.

Referencias

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Education.
- Python Lex-Yacc (PLY) documentation: https://www.dabeaz.com/ply/
- Gutiérrez Mondragón, M. A. (2025). Apuntes y prácticas de la materia de Compiladores. ESCOM-IPN.