Compilador usando Flex y Bison

Mauricio Díaz

Nicolas Cortes

Profesor Jose Luis Veas

Fundamentos de la Computación

Introducción

En este proyecto se desarrolla un compilador sencillo utilizando las herramientas Flex y Bison, ampliamente empleadas en la construcción de analizadores léxicos y sintácticos. El objetivo es implementar un lenguaje propio que permite declarar variables, realizar operaciones aritméticas, utilizar estructuras de control como if, while, y funciones de entrada/salida como leer desde el teclado e imprimir por pantalla. Este trabajo busca familiarizarse con los fundamentos de los lenguajes formales, el análisis sintáctico y la generación de árboles de sintaxis abstracta (AST), fundamentales en la creación de compiladores.

Gramática del Lenguaje KeyeScript

La gramática ha sido implementada utilizando Flex (scanner.l) y Bison (parser.y), incluyendo tanto el análisis léxico como sintáctico. Esta gramática fue diseñada para soportar variables de tipo entero, flotante y cadena, incluyendo operaciones aritméticas básicas, estructuras de control condicional (if, else) y repetitiva, lectura y escritura de datos, y funciones aritméticas y matemáticas como calcular potencias.

1. Análisis Léxico (scanner.l)

El analizador léxico se encarga de reconocer los componentes básicos del lenguaje (tokens) y categorizarlos para su posterior procesamiento sintáctico. Las siguientes reglas definen los tokens del lenguaje:

1.1 Palabras clave:

KeyeScript define las siguientes palabras reservadas, que son esenciales para la estructura del lenguaje:

-Tipos de datos: entero, flotante, cadena

-Estructuras de control: si, sino, mientras

-Entrada/salida: imprimirKeye, leer

-Funciones matemáticas: potencia

1.2 Operadores y Símbolos

Los operadores reconocidos incluyen:

-Aritméticos: +, -, *, /

-Asignación: =

-Potenciación: ^

-Separadores: ,, ;, (,), {, }

1.3 Literales

-Enteros: Secuencias de dígitos ([0-9]+) → ENTERO_LIT

-Flotantes: Números con punto decimal ([0-9]+\.[0-9]*) → FLOTANTE_LIT

-Cadenas: Secuencias de letras y dígitos ((letra(letra|digito)*)) → CADENA_LIT

1.4 Elementos Ignorados

Espacios en blanco, tabulaciones y saltos de línea no afectan la estructura del programa.

1.5 Manejo de Errores Léxicos

Cualquier símbolo no reconocido genera un error léxico, lo que permite una detección temprana de caracteres inválidos.

2. Análisis Sintáctico (parser.y)

El analizador sintáctico define la estructura gramatical del lenguaje mediante reglas de producción. KeyeScript sigue una gramática recursiva por la izquierda para manejar expresiones y declaraciones de manera eficiente.

2.1 Estructura General del Programa

Un programa en KeyeScript consiste en una secuencia de declaraciones:

```
programa → lista_declaraciones

lista_declaraciones → lista_declaraciones declaracion |
declaracion
```

2.2. Declaraciones

Las declaraciones pueden ser de distintos tipos:

- **Declaración de variables:**

```
declaracion → TIPO IDENTIFICADOR ';'
            | TIPO IDENTIFICADOR '=' expresion ';'
```

- **Asignación:**

```
declaracion → IDENTIFICADOR '=' expresion ';'
```

- **Estructuras de control:**

  Condicional (si-sino):

```
declaracion → SI '(' expresion ')' bloque
            | SI '(' expresion ')' bloque SINO bloque
```

  Bucle (mientras):

```
declaracion → MIENTRAS '(' expresion ')' bloque
```

- **Entrada/salida:**

  Impresión: imprimirKeye(expresion);
  Lectura: leer(IDENTIFICADOR);

- **Bloques:** Secuencias de declaraciones entre llaves { }

2.3. Expresiones

Las expresiones siguen una jerarquía de operaciones para evitar ambigüedades:

```
expresion → expresion '+' termino
| expresion '-' termino
| termino

termino → termino '*' factor
| termino '/' factor
| factor

factor → ENTERO_LIT
| FLOTANTE_LIT
| CADENA_LIT
| IDENTIFICADOR
| '(' expresion ')'
| potencia_expr
```

2.4. Funciones Matemáticas

La función potencia permite cálculos exponenciales:

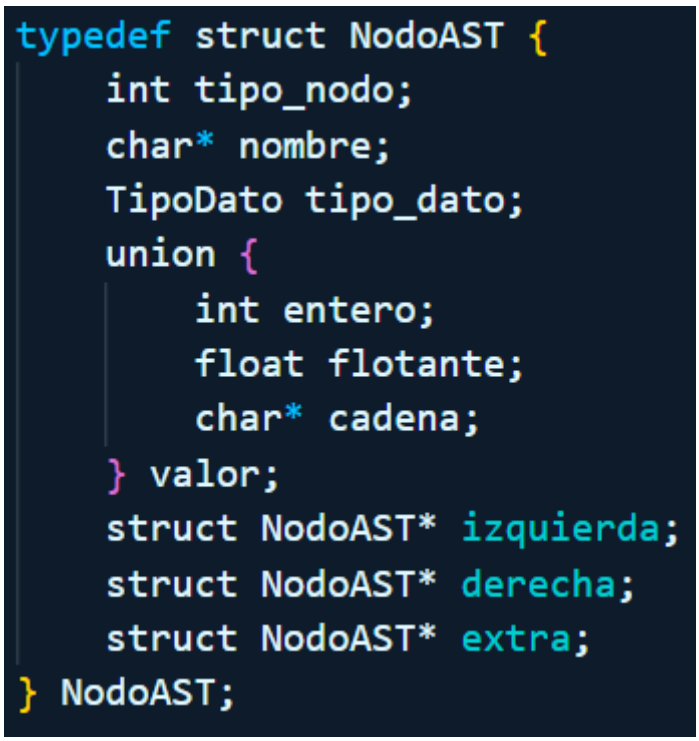
```
potencia_expr → POTENCIA '(' expresion ',' expresion ')'
```

Diseño y Estructura del Árbol de Sintaxis Abstracta (AST)

El Árbol de Sintaxis Abstracta (AST) es una representación jerárquica del código fuente que captura su estructura lógica, eliminando detalles innecesarios (como paréntesis o puntos y coma). En este compilador, el AST se construye durante el análisis sintáctico y se usa para generar código.

1. **Estructura del Nodo AST**

Cada nodo del AST se define en ast.h con la siguiente estructura:



Tipos de Nodos

Cada nodo del AST tiene:

- **Tipo:** Indica la categoría (ej. Declaración, Expresión, If-Else).
- **Atributos:** Información relevante (ej. valor de un número, nombre de una variable).
- **Hijos:** Subnodos que representan componentes más pequeños (ej. condición y cuerpo de un if).

Proceso de Generación de Código y Ejemplos

La generación de código en el compilador de KeyeScript se realiza a partir del Árbol de Sintaxis Abstracta (AST) que se construye en tiempo de análisis sintáctico. A diferencia de un compilador tradicional que genera código en lenguaje ensamblador o intermedio, este compilador ejecuta directamente el AST mediante recorridos recursivos.

El archivo principal encargado de esta ejecución es ast.c, que implementa la función `ejecutar_ast`. Esta función recibe un nodo del AST y, dependiendo de su tipo, realiza diferentes acciones:

- **Declaraciones ('D')**: Se agrega la variable a la tabla de símbolos junto a su tipo.
- **Asignaciones ('=')**: Evalúa la expresión del lado derecho y actualiza el valor de la variable en la tabla de símbolos.
- **Impresión (IMPRIMIR)**: Evalúa la expresión y muestra el resultado en pantalla según su tipo.
- **Lectura (LEER)**: Solicita al usuario un valor por teclado y lo guarda en la variable correspondiente, validando su tipo.
- **Condicionales (SI, SINO)**: Evalúa la condición; si se cumple, ejecuta el bloque asociado, de lo contrario, ejecuta el bloque alternativo si existe.
- **Bucles (MIENTRAS)**: Ejecuta el bloque de instrucciones mientras la condición se mantenga verdadera.
- **Expresiones**: Se evalúan recursivamente mediante la función evaluar_expresion, que interpreta operaciones aritméticas (+, -, *, /), la función potencia, y accede a los valores de las variables.

Además, el compilador mantiene una **tabla de símbolos** para registrar variables, tipos y valores. Esta tabla se consulta y actualiza constantemente durante la ejecución del AST.

En resumen, el proceso de “generación de código” en este lenguaje no produce un código textual, sino que ejecuta directamente la lógica representada en el AST utilizando código en C, actuando como un **intérprete** para el lenguaje KeyeScript.

- **Ejemplos:**

```
entero x;
x = 5 + 3;
imprimirKeye(x);
```

Explicación de ejecución:

1. Se crea un nodo de declaración 'D' y se registra la variable x como tipo entero en la tabla de símbolos (función `agregar_simbolo`).
2. Se construye un nodo de asignación '=' con una subexpresión 5 + 3, evaluada mediante `evaluar_expresion`, que retorna 8.
3. El valor 8 se asigna a la variable x.
4. El nodo IMPRIMIR accede al valor actual de x y lo muestra usando `printf`.

Manual de Usuario del Lenguaje de Programación KeyeScript

KeyeScript es un lenguaje de programación estructurado y diseñado para ilustrar los conceptos fundamentales de compilación, análisis léxico, análisis sintáctico, evaluación de expresiones y ejecución de instrucciones mediante un Árbol de Sintaxis Abstracta (AST). Este manual explica cómo utilizar el lenguaje, sus componentes principales y la sintaxis permitida.

1. Tipos de Datos

KeyeScript permite declarar variables con los siguientes tipos:

- **entero**: valores numéricos sin decimales. Ej: 10, -4
- **flotante**: valores numéricos con decimales. Ej: 3.14, -0.5
- **cadena**: texto encerrado entre comillas dobles. Ej: "hola"

2. Declaración de Variables

Antes de usarlas, las variables deben declararse especificando su tipo:

```
entero edad;
flotante promedio;
cadena nombre;
```

3. Asignación de Valores

La asignación se realiza con el operador =:

```
x = 5;
y = 2.5;
nombre = "Juan";
```

4. Entrada y Salida de Datos

* Para solicitar datos: leer x;
* Para mostrar valores: imprimirKeye(x);

5. Operadores Aritméticos

* Suma: +
* Resta: -
* Multiplicación: *
* División: /
* Potencia: potencia(a, b) (retorna a^b)

6. Operadores de Comparación

- Igual a: ==
- Distinto de: !=
- Mayor que: >
- Menor que: <
- Mayor o igual que: >=
- Menor o igual que: <=

7. Operadores Lógicos

AND, OR, NOT

8. Estructuras de Control

- Condicional si/sino:

```
si (condición) {
    // bloque verdadero
} sino {
    // bloque falso
}
```

- Bucle mientras:

```
mientras (condición) {
    // instrucciones repetidas
}
```

9. Funciones Incorporadas

- potencia(a, b): devuelve a elevado a la b

10. Reglas del lenguaje

- Las variables deben ser declaradas antes de usarse.
- Solo una vez declarada una variable, se le puede asignar un valor.
- Las lineas deben terminar con un ;
- No se permite dividir entre cero. Hacerlo entregará un error
- Las operaciones deben respetar el tipo de datos.
- Los errores (como variable no declarada) terminan la ejecución del programa.

11. Manejo de Errores Comunes

- Error léxico: uso de caracteres inválidos.
- Variable no declarada: intentar usar una variable sin haberla definido antes.
- División por cero: intentar dividir entre cero en una expresión.

12. Ejemplo de Programa Básico

```
entero a;
entero b;
entero resultado;

a = 2;
b = 3;

resultado = potencia(a, b);
imprimirKeye("El resultado de a^b es:");
imprimirKeye(resultado);
```

13. Como Compilar el lenguaje

Requisitos Previos

Sistema Operativo: Linux (recomendado) o Windows con WSL.

Herramientas Instaladas:

- flex: Analizador léxico.
- bison: Analizador sintáctico.
- gcc: Compilador de C.

Generar Analizadores con Flex y Bison

- Ejecuta estos comandos en orden:

```
# Generar analizador léxico a partir de scanner.l
flex scanner.l

# Generar analizador sintáctico a partir de parser.y
bison -d parser.y
```

Compilar el Proyecto:

```
gcc lex.yy.c parser.tab.c ast.c -o keyescript -lm
```

Ejecutar el Compilador:

Se debe tener el código a compilar en un archivo de tipo txt, luego, ejecutar el siguiente comando:

```
./keyescript < archivo.txt
```