Universidad Católica del Norte

Escuela de Ingeniería en Computación e Informática

Informe Teórico

Creación de un Compilador para un Lenguaje Propio (CEWE)

Autor: Nicolás Gonzalo Cordero Varas
Fecha: 17 Junio 2025
RUT: 20.543.155-1
Docente: Jose Luis Veas
Fecha: 17 Junio 2025

Índice

1. Introducción

En el ámbito de las ciencias de la computación, la construcción de compiladores representa una de las tareas más complejas y formativas que un programador puede enfrentar. La creación de un compilador desde cero no solo implica dominar aspectos técnicos como el análisis léxico, sintáctico y semántico, sino también comprender a profundidad el funcionamiento interno de los lenguajes de programación, sus estructuras y la forma en que son interpretadas por una máquina.

Este informe describe el proceso de diseño e implementación de un compilador para un lenguaje propio denominado **Cewe**. Este lenguaje, además de cumplir funciones básicas como entrada, salida y operaciones aritméticas, está orientado a propósitos didácticos, incluyendo una sintaxis lúdica y amigable, con nombres como `printiwi`, `forowo` o `whilewe`. Su principal objetivo es servir como puente entre la teoría de lenguajes formales y su aplicación práctica en sistemas de traducción de código.

El proyecto Cewe fue desarrollado utilizando herramientas clásicas de compilación en sistemas GNU/Linux, como lo son Flex y Bison. Flex permite definir el comportamiento del analizador léxico mediante expresiones regulares, mientras que Bison facilita la construcción del parser, utilizando gramáticas libres de contexto. Complementariamente, se desarrolló una arquitectura de árbol sintáctico abstracto (AST) en lenguaje C, que permite representar la estructura jerárquica del código fuente y realizar su evaluación directa.

Este trabajo se enmarca en la asignatura de Fundamentos de la Computación, impartida en la Escuela de Ingeniería en Computación e Informática de la Universidad Católica del Norte, y tiene como propósito no solo desarrollar competencias prácticas, sino también cimentar una comprensión profunda del proceso de compilación. De este modo, se exploran los fundamentos teóricos, las decisiones de diseño, las estructuras internas del compilador y se demuestra, mediante un ejemplo funcional (una calculadora de potencias y operaciones básicas), que el lenguaje Cewe es ejecutable y útil como herramienta de aprendizaje.

Cabe destacar que el proceso de implementación del compilador implicó múltiples etapas iterativas, desde el diseño de tokens, la depuración de reglas gramaticales, la validación de árboles sintácticos, hasta la construcción de una interfaz mínima de ejecución. Esto permitió integrar conocimientos transversales como estructuras de datos, lógica de programación, uso de punteros en C, control del flujo de ejecución, y uso avanzado de herramientas de consola y scripts de automatización.

Finalmente, el presente documento busca no solo dejar constancia de los logros obtenidos, sino también servir como guía y referencia para futuros desarrollos de compiladores propios. En cada sección se desarrollan los principios teóricos subyacentes, su aplicación concreta al lenguaje Cewe, y los resultados observables tras la ejecución del código fuente. Así, se espera aportar tanto al aprendizaje personal como al acervo académico de la comunidad universitaria.

2. Fundamentos Teóricos

2.1. ¿Qué es un Lenguaje de Programación?

Un lenguaje de programación es un sistema formal compuesto por un conjunto finito de símbolos y reglas sintácticas y semánticas que permiten a los seres humanos comunicar instrucciones a una computadora. Estos lenguajes son una abstracción que permite ocultar los detalles del hardware subyacente, facilitando la implementación de algoritmos, estructuras de datos y sistemas complejos.

Desde una perspectiva teórica, un lenguaje de programación puede definirse formalmente como un conjunto de cadenas de caracteres sobre un alfabeto dado que son aceptadas por una gramática formal. Estas gramáticas, clasificadas según la jerarquía de Chomsky, permiten definir la sintaxis del lenguaje de forma precisa. En la práctica, los lenguajes de programación se sitúan mayoritariamente en el nivel de las gramáticas libres de contexto (tipo 2), lo que los hace aptos para ser analizados mediante autómatas de pila.

El lenguaje CEWE, desarrollado para este proyecto, es un lenguaje imperativo, minimalista y con fines educativos. Su sintaxis y nomenclatura fueron diseñadas para facilitar el aprendizaje y la relación entre los conceptos teóricos y su implementación práctica. Palabras clave como `printuwu`, `ifiwi` o `whiliwi` actúan como sustitutos lúdicos de estructuras tradicionales como `print`, `if` y `while`, incentivando una relación más amena y creativa con la lógica de control.

2.2. ¿Qué es un Compilador?

Un compilador es un programa informático que traduce un código fuente escrito en un lenguaje de alto nivel a un código objeto o ejecutable que puede ser comprendido y ejecutado por una máquina. A diferencia de los intérpretes, que traducen línea por línea en tiempo de ejecución, los compiladores realizan la traducción completa en una fase previa a la ejecución. Esto permite optimizaciones más profundas y una ejecución más eficiente del programa final.

El proceso de compilación tradicionalmente se divide en múltiples fases:

1. **Análisis léxico**: El código fuente se divide en componentes léxicos denominados *tokens*. Estos tokens son secuencias de caracteres que representan las unidades básicas del lenguaje, como palabras clave, identificadores, operadores, literales, etc. Esta etapa es comúnmente implementada mediante autómatas finitos y expresiones regulares.

2. **Análisis sintáctico (Parsing)**: En esta etapa se verifica que los tokens generados formen construcciones válidas de acuerdo con la gramática del lenguaje. Se construye un árbol de derivación (o AST) que representa jerárquicamente la estructura del programa. Este proceso suele implementarse utilizando algoritmos de análisis descendente

(LL) o ascendente (LR).

3. **Análisis semántico**: Aquí se comprueba que las construcciones sean coherentes a nivel semántico. Por ejemplo, se valida que las variables estén declaradas antes de ser usadas, que las operaciones se realicen entre tipos compatibles, y que las funciones reciban el número y tipo correctos de argumentos.

4. **Generación de código intermedio**: Se transforma el AST en una representación intermedia más cercana al lenguaje de máquina, pero aún independiente del hardware específico. Esta representación suele facilitar optimizaciones posteriores.

5. **Optimización**: Opcionalmente, se realiza un análisis profundo del código intermedio para mejorar su eficiencia, reducir el uso de recursos o acortar el tiempo de ejecución.

6. **Generación de código objeto**: Finalmente, el compilador emite instrucciones en lenguaje máquina o ensamblador para la arquitectura destino.

En el compilador Cewe se implementan principalmente las tres primeras fases: análisis léxico con Flex, análisis sintáctico con Bison, y evaluación semántica mediante un AST en C. La generación de código final es sustituida por la evaluación directa del árbol de sintaxis.

2.3. Lenguajes Formales y Gramáticas

El diseño de un lenguaje requiere el conocimiento de los lenguajes formales, los cuales son estudiados en el marco de la teoría de autómatas. Una gramática formal es una tupla $G = (N, \Sigma, P, S)$, donde:

- N es un conjunto finito de símbolos no terminales.
- Σ es un conjunto finito de símbolos terminales (alfabeto).
- P es un conjunto finito de producciones o reglas de reescritura.
- S es el símbolo inicial.

Las gramáticas más utilizadas en la práctica para lenguajes de programación son las libres de contexto (CFG), ya que permiten definir estructuras recursivas como expresiones anidadas, bloques de código, listas de argumentos, etc.

2.4. Tokens y Expresiones Regulares

Los **tokens** son las unidades básicas del lenguaje reconocidas durante el análisis léxico. Cada token se define mediante una expresión regular que describe un patrón de caracteres aceptables. Por ejemplo:

- `[0-9]+` ⇒ Número entero.
- `[a-zA-Z_][a-zA-Z0-9_]*` ⇒ Identificador.
- `"\n"` ⇒ Nueva línea.

El analizador léxico creado con Flex escanea el código fuente y transforma la secuencia de caracteres en una secuencia de tokens. Esta transformación es indispensable para que el parser posterior pueda trabajar sobre unidades significativas, en lugar de caracteres individuales.

2.5. Parsers y Árboles de Derivación

Un **parser** es un componente del compilador encargado de analizar la estructura sintáctica de los tokens generados. El parser construye un árbol de derivación o AST que refleja la jerarquía de las expresiones. Los parsers más comunes en compiladores modernos son de tipo LL (descendentes) y LR (ascendentes), siendo Bison un generador de parsers LALR(1), una variante eficiente de los LR.

El AST (Árbol de Sintaxis Abstracta) es una versión simplificada del árbol de derivación que elimina elementos redundantes y agrupa los nodos según su significado semántico. Es en este árbol donde se implementan las evaluaciones semánticas y lógicas del código fuente.

2.6. Tokens y Expresiones Regulares

Los **tokens** son las unidades básicas del lenguaje reconocidas durante el análisis léxico. Cada token se define mediante una expresión regular que describe un patrón de caracteres aceptables. Por ejemplo:

- `[0-9]+` ⇒ Número entero.
- `[a-zA-Z_][a-zA-Z0-9_]*` ⇒ Identificador.
- `"\n"` ⇒ Nueva línea.

El analizador léxico creado con Flex escanea el código fuente y transforma la secuencia de caracteres en una secuencia de tokens. Esta transformación es indispensable para que el parser posterior pueda trabajar sobre unidades significativas, en lugar de caracteres individuales.

2.7. Parsers y Árboles de Derivación

Un **parser** es un componente del compilador encargado de analizar la estructura sintáctica de los tokens generados. El parser construye un árbol de derivación o AST que refleja la jerarquía de las expresiones. Los parsers más comunes en compiladores modernos son de tipo LL (descendentes) y LR (ascendentes), siendo Bison un generador de parsers LALR(1), una variante eficiente de los LR.

El AST (Árbol de Sintaxis Abstracta) es una versión simplificada del árbol de derivación que elimina elementos redundantes y agrupa los nodos según su significado semántico. Es en este árbol donde se implementan las evaluaciones semánticas y lógicas del código fuente.

3. Análisis Léxico con Flex

El análisis léxico es la primera etapa del proceso de compilación, y su función principal es transformar la cadena de entrada (código fuente) en una secuencia de tokens. Un token representa una unidad léxica válida, como un identificador, palabra clave, número o símbolo de operador.

Desde la teoría de autómatas, el análisis léxico puede modelarse mediante autómatas finitos deterministas (DFA), dado que los lenguajes que describen los tokens suelen pertenecer a la clase de lenguajes regulares. Esto permite expresar cada patrón de token como una expresión regular. Flex (Fast Lexical Analyzer) es una herramienta que genera automáticamente un DFA eficiente a partir de estas expresiones.

3.1. Funcionamiento de Flex

El archivo `scanner.l` contiene tres secciones: definiciones, reglas y código adicional. En la sección de reglas se definen las expresiones regulares asociadas a cada token. Flex transforma estas reglas en un analizador eficiente implementado en C (`lex.yy.c`), que se encarga de leer la entrada carácter por carácter y devolver el tipo de token correspondiente.

Ejemplos de definiciones léxicas en Cewe:

- `printuwu` ⇒ `PRINT`
- `intiwi` ⇒ `INTIWI`
- `[0-9]+` ⇒ `NUM`
- `[a-zA-Z_][a-zA-Z0-9_]*` ⇒ `ID`

Cada token reconocido por Flex es devuelto al parser (Bison), el cual interpreta su función sintáctica dentro del programa.

3.2. Expresiones Regulares y Autómatas

Una expresión regular (ER) es una fórmula que describe un lenguaje regular. Las ER son cerradas bajo las operaciones de concatenación, unión y estrella de Kleene, lo que las convierte en herramientas ideales para representar la estructura de tokens.

Las expresiones regulares pueden traducirse a autómatas finitos no deterministas (NFA) mediante el algoritmo de Thompson, y luego a DFA utilizando el algoritmo de subconjuntos. Flex realiza esta conversión internamente, optimizando el analizador para su uso en compilación real.

3.3. Ventajas del Análisis Léxico Separado

Separar el análisis léxico del análisis sintáctico ofrece ventajas claras:

- Mejora la modularidad del compilador.
- Facilita la detección temprana de errores léxicos.
- Aumenta la eficiencia del parser al trabajar sobre tokens y no sobre caracteres individuales.

El uso de Flex permite mantener esta separación de forma automática y eficiente, respetando la estructura modular del compilador Cewe.

4. Análisis Sintáctico con Bison

Una vez que los tokens han sido identificados por el analizador léxico, el parser toma el control del flujo de entrada. Bison es una herramienta de análisis sintáctico que genera un parser ascendente (de tipo LALR(1)) a partir de una gramática libre de contexto.

4.1. Gramática del Lenguaje CEWE

La gramática define cómo se estructuran los programas válidos del lenguaje. En el archivo `parser.y`, la gramática CEWE establece reglas para:

- Declaraciones de funciones.

- Sentencias condicionales (`ifiwi`, `elsewe`).
- Bucles de repetición (`whiliwi`).
- Expresiones aritméticas y lógicas.
- Llamadas y retornos de funciones.

Bison construye un árbol de derivación aplicando las reglas desde los tokens hacia el símbolo inicial (`programa`). A diferencia de Flex, Bison opera sobre secuencias de tokens y utiliza una pila para reducir cadenas de tokens a símbolos no terminales, siguiendo el paradigma bottom-up.

4.2. Conflictos en Parsing y Resolución

Durante la definición de gramáticas, pueden surgir conflictos como *shift/reduce* o *reduce/reduce*. Bison utiliza técnicas LALR(1) que permiten resolver muchos de estos conflictos mediante un único token lookahead. Sin embargo, una mala definición gramatical puede hacer que el parser generado sea ambiguo o erróneo, lo cual obliga a reescribir o factorear la gramática.

4.3. Integración con el AST

En CEWE, cada regla de producción en `parser.y` no solo valida la sintaxis, sino que además construye nodos del árbol de sintaxis abstracta (AST). Este árbol representa la estructura semántica del programa y sirve como entrada directa para su evaluación.

Cada nodo del AST es una estructura en C que contiene:

- El tipo de nodo (suma, multiplicación, llamada, retorno, condición, etc.).
- Subárboles hijos (izquierdo, derecho o lista de argumentos).
- Información contextual (identificador, valor, condición).

4.4. Evaluación Directa del AST

Una vez construido el AST, no se realiza una traducción adicional a código objeto. En lugar de eso, se recorre recursivamente el árbol con funciones en C que evalúan cada nodo de acuerdo con su tipo. Por ejemplo, los nodos de operaciones binarias suman o multiplican los valores de sus hijos; los nodos `if` evalúan la condición y escogen el subárbol correspondiente.

Esto permite ejecutar el programa directamente desde su estructura lógica, lo cual es común en lenguajes interpretados o en compiladores educativos como CEWE.

4.5. Ventajas del Enfoque AST + Evaluación

- Permite mantener el código modular, legible y extensible.
- Facilita el debug del flujo lógico del programa.
- Evita la necesidad de generar código de máquina o bytecode.
- Permite implementar funciones recursivas y estructuras complejas sin mayor esfuerzo.

Este enfoque resulta especialmente útil en etapas tempranas de desarrollo de un lenguaje, o en entornos educativos donde lo primordial es comprender el funcionamiento interno de un compilador.

5. Diseño del Lenguaje Cewe

CEWE es un lenguaje imperativo, con una gramática inspirada en lenguajes como C o JavaScript. Utiliza sufijos amigables como `printiwi`, `forowo`, `whilewe` que reemplazan términos tradicionales.

Funciones se definen con `fun`, las variables con `intiwi` y la salida con `printiwi`. Los operadores aritméticos (+, -, *, /) mantienen su notación estándar.

6. Árbol de Sintaxis Abstracta (AST)

El árbol AST representa la estructura semántica del programa. Los nodos son de diferentes tipos:

- `NODE_NUM`: nodos con valores numéricos
- `NODE_OPERACION_BINARIA`: operaciones como suma o multiplicación
- `NODE_LLAMADA_FUNCION`, `NODE_DECLARACION_FUNCION`
- `NODE_RETURN`, `NODE_IF`, `NODE_WHILE`

Cada nodo contiene sus hijos y puede evaluarse recursivamente.

7. Evaluación del AST

El archivo `ast.c` contiene funciones para evaluar el AST. No se genera un ejecutable, sino que el programa se interpreta directamente desde el árbol.

Ejemplo de evaluación:

```
if (nodo->tipo == NODE_OPERACION_BINARIA) {
    int izq = evaluar(nodo->izq);
    int der = evaluar(nodo->der);
    return izq + der; // seg n operador
}
```

Esto permite simular el comportamiento de un programa interpretado, sin compilar.

8. Programa Cewe: Calculadora

El archivo `programa.cewe` implementa una calculadora que:

- Suma, resta, multiplica y divide dos números
- Calcula potencias usando un ciclo `whiliwi` con multiplicaciones sucesivas
- Utiliza funciones como `sumar`, `restar`, `potencia`
- Muestra resultados en consola con `printuwu`

Ejemplo:

```
fun potencia(base, exp) {
  intiwi res = 1;
  intiwi i = 0;
  whiliwi (i < exp) {
    res = res * base;
    i = i + 1;
  }
  returnuwu res;
}
```

9. Herramientas Utilizadas

- **Flex**: definición de patrones léxicos
- **Bison**: generación del parser
- **GCC**: compilación del proyecto
- **Bash**: automatización con `build.sh`
- **GNU/Linux**: entorno de trabajo para compilación y pruebas

10. Conclusiones

Este proyecto integró exitosamente los principales elementos teóricos de los lenguajes de programación, autómatas y gramáticas. La experiencia permitió entender el flujo completo de compilación:

- Desde una entrada textual hasta la interpretación real del código.
- Implementación concreta de árboles AST, recursividad, tablas de símbolos.
- Integración de herramientas reales como Flex y Bison, ampliamente utilizadas en compiladores reales.

El lenguaje Cewe sirve como plataforma pedagógica eficaz para explorar los principios fundamentales de la computación.