Informe del Proyecto de Compilador Chileno

Autores: Abraham Sepúlveda y Matías Villalobos
Universidad Católica del Norte
Carrera: Ingeniería Civil en Computación e Informática
Año: 2025

1. Introducción

Este proyecto consistió en la creación de un compilador que interpreta un lenguaje de programación diseñado con expresiones chilenas. Se utilizaron herramientas clásicas de compilación como Flex y Bison para construir un analizador léxico y sintáctico respectivamente. El objetivo fue adaptar estructuras tradicionales como condicionales y ciclos a una sintaxis local y representarla mediante un Árbol de Sintaxis Abstracta (AST).

2. Objetivos

- Crear un compilador funcional usando Flex y Bison.

- Adaptar la sintaxis del lenguaje a expresiones chilenas.

- Generar un árbol de sintaxis abstracta (AST).

- Permitir la interpretación estructural del código fuente chileno.

- Incorporar funciones con parámetros, retorno y recursividad.

- Ejecutar e interpretar directamente desde el AST.

- Generar código C++ desde el AST.

3. Herramientas Utilizadas

- Lenguaje de programación: C++

- Generadores: Bison (parser), Flex (scanner)

- Entorno: MSYS2 en Windows

- Editor: Visual Studio Code

4. Diseño del Lenguaje Chileno

Función clásica	Versión Chilena
if	si_po
else	si_no_po
while	mientras_la_wa
for	pa_cada
int	numerito
string	palabrita
print	suelta_la_wa
read/input	lee_la_wa
function	hace_la_pega
return	devuelve_la_wa
==	igualito
!=	igualitont
<=	igualitito
>=	igualitote

5. Implementación Técnica

5.1 Escáner (.l)

- Reconoce palabras clave chilenas, identificadores, números, flotantes y cadenas.

- Utiliza expresiones regulares avanzadas para diferenciar tipos.

5.2 Analizador Sintáctico (.y)

- Define estructuras para declaraciones, funciones, llamadas, expresiones, condiciones, ciclos.

- Usa `%union` y múltiples tipos de tokens.

5.3 Árbol de Sintaxis Abstracta (AST)

- Construcción del AST

    Define nodos para distintos elementos como literales (int, float, string), identificadores, asignaciones, operadores, control de flujo (if, while, for) y funciones.

- Cada tipo de nodo se construye con su propia función make_*.

Evaluación del AST

- eval_ast(AST* tree) es el motor de ejecución, procesando expresiones, asignaciones y estructuras de control.

Variables (NODE_DECL) y su manejo con std::map<std::string, VarInfo> para almacenar su tipo y valor.

Funciones (NODE_FUNC_DEF y NODE_FUNC_CALL) permiten reutilización de código, con parámetros y ejecución.

- Generación de Código en C++

- generar_programa(AST* tree) convierte el AST en un programa C++ válido.

- generate_code_funcs(AST* tree) genera las funciones antes del main().

- generate_code_main(AST* tree) traduce las operaciones AST a código C++.

Características:

Control de flujo completo (if, while, for)

Gestión de funciones y parámetros

Conversión de expresiones en código C++

Entrada/salida con cin y cout

Validación de tipos y errores semánticos

5.4 Evaluador (interpretación directa)

- `eval_ast()` ejecuta el AST como si fuera un programa.

- Soporta variables globales, entorno local de funciones, retorno de valores.

- Se implementa la recursividad, ejemplo: `factorial(5)`.

5.5 Generador de Código

- `generar_programa()` transforma el AST en código C + +.

- Se generan funciones y el cuerpo del `main`.

- Compatible con entradas y salidas por consola.

6. Resultados

- El compilador reconoce y ejecuta correctamente programas escritos en lenguaje chileno.

- Se pueden definir y llamar funciones, incluyendo llamadas recursivas.

- Se genera código C++ equivalente, guardado en `cpp_chileno.cpp`.

- El AST es impreso visualmente para análisis del árbol.

Ejemplo de ejecución:

Código:

```
hace_la_pega factorial(n) {
    si_po (n igualito 0) {
        devuelve_la_wa 1;
    }
    si_no_po {
        devuelve_la_wa n * factorial(n - 1);
    }
}

numerito resultado;
numerito valor = 5;
resultado = factorial(valor);
suelta_la_wa resultado;

```

AST que imprime:

```
--- Arbol de sintaxis generado ---
SEQ
  SEQ
    SEQ
      SEQ
        FUNC_DEF: factorial
          PARAMS
            n
          IF
            BINOP (==)
              ID: n
              INT: 0
            RETURN
              INT: 1
            RETURN
              BINOP (*)
                ID: n
                FUNC_CALL: factorial
                  ARGS
                    BINOP (-)
                      ID: n
                      INT: 1
        DECLARACION: resultado Tipo: int
      SEQ
        DECLARACION: valor Tipo: int
        ASSIGN
          ID: valor
          INT: 5
    ASSIGN
      ID: resultado
      FUNC_CALL: factorial
        ARGS
          ID: valor
  PRINT
    ID: resultado
```

Lo que imprime el programa:

```
--- Ejecucion del programa ---
120

--- Generando codigo C++ ---
Archivo generado: cpp_chileno.cpp
```

Código en c++ generado:

cpp_chileno.cpp > factorial(auto)

```cpp
#include <iostream>
#include <string>
using namespace std;

auto factorial(auto n) {
if ((n == 0)) {
return 1;
}
else {
return (n * factorial((n - 1)));
}
}

int main() {
int resultado;
int valor;
valor = 5;
resultado = factorial(valor);
;
cout << resultado;
return 0;
}
```

7. Manual de usuario

Una vez obtenido el repositorio.

1- Ejecuta el generador de análisis sintáctico:

`bison -d chileno.y

2-Renombra el archivo “chileno.tab.c” a “chileno.tab.cpp” para que sea código C + +.

`mv chileno.tab.c chileno.tab.cpp`

3- Le pasamos chileno.l a flex (el generador de analisis lexico)

`flex chileno.l

4- Compila el parser, el scanner y el archivo que define y maneja el AST.

g++ chileno.tab.cpp lex.yy.c ast.cpp -o chileno_compilador

5-Ejecutar

Ejecuta el comando requerido para el test de prueba.

-Test ciclos

./chileno_compilador test/ciclos.txt

-**Test condicionales**

./chileno_compilador test/condicionales.txt

- Test funciones

./chileno_compilador test/funciones.txt

-Test ciclos

./chileno_compilador test/input.txt

-Test completo

./chileno_compilador test/completo.txt

-Ejercicio Profesor

./chileno_compilador ejercicio_profesor/ejercicio.txt

¿Qué muestra por pantalla?

Primero imprime el AST.

Luego imprime lo que ejecuta el programa.

Finalmente imprime que se ha generado el código en C + +.

Compilar archivo generado C++

g++ cpp_chileno.cpp -o cpp_chileno.exe

Ejecutar archivo generado C++

./cpp_chileno

8. Conclusiones

El desarrollo de este compilador demuestra que es posible adaptar la tecnología a un contexto cultural específico sin comprometer su funcionalidad. A través de herramientas como Flex y Bison, se ha logrado construir un lenguaje desde cero, lo que resalta la versatilidad y el potencial de estos frameworks en la creación de lenguajes personalizados.

Además, la implementación de ejecución y generación de código no solo amplía las capacidades del compilador, sino que también refuerza su utilidad en entornos educativos, permitiendo una enseñanza más práctica y accesible sobre el funcionamiento de los lenguajes de programación. En definitiva, este proyecto no sólo valida la viabilidad de una adaptación cultural en el desarrollo de software, sino que también abre nuevas posibilidades para su aplicación en la enseñanza y exploración de la informática.