Informe Compilador ForceCode

Autores: Pablo Villarroel y Sebastián Pleticosic
Asignatura: Fundamentos de la Computación
Profesor: José Veas
Fecha de Entrega: 17-06-2025

Índice

Introducción

Este informe detalla el proceso de diseño e implementación del compilador **ForceCode**, un lenguaje temático inspirado en Star Wars. El proyecto abarca todas las etapas de un compilador moderno: análisis léxico, sintáctico, semántico, construcción de un Árbol de Sintaxis Abstracta (AST), generación de código y documentación. La versión actual del compilador introduce una característica fundamental: el **soporte completo para funciones**, permitiendo la declaración con parámetros, llamadas en expresiones y el retorno de valores, lo que aumenta significativamente la modularidad y expresividad del lenguaje.

Arquitectura General del Compilador

El compilador ForceCode se compone de los siguientes módulos principales:

- **scanner.l**: Analizador léxico (Flex) que reconoce tokens, palabras clave, identificadores, literales y reporta errores léxicos.
- **parser.y**: Analizador sintáctico (Bison) que verifica la estructura del programa, incluyendo definiciones y llamadas a funciones, y construye el AST.
- **ast.hpp** y **ast.cpp**: Definen la estructura y los métodos del AST. Esto incluye nodos para declaraciones, expresiones, sentencias de control y, de manera crucial, para la **definición de funciones (NodoFuncion)**, **llamadas a funciones (NodoLlamadaFuncion)** y **sentencias de retorno (NodoReturn)**.
- **semantic.cpp** y **semantic.hpp**: Implementan el análisis semántico. Verifican la consistencia de tipos, gestionan la tabla de símbolos (incluyendo ámbitos para funciones y sus parámetros) y detectan errores como el uso de variables no declaradas o la redeclaración.
- **main.cpp**: Punto de entrada que gestiona los archivos de entrada, invoca el parser, inicia el análisis semántico y la posterior generación de código C.
- **makefile**: Automatiza el proceso de compilación del propio compilador, enlazando todos los módulos para generar el ejecutable mycompiler.
- **Documentación**: README.md, MANUAL_STARWARS.md y MANUAL_EJECUCION.md proveen toda la información necesaria para compilar, ejecutar y programar en ForceCode.

Diseño del Lenguaje y Gramática

ForceCode es un lenguaje imperativo, estructurado y de tipado estático, con una sintaxis inspirada en C/C++ pero ambientada en el universo de Star Wars. La adición más significativa es el soporte para funciones, permitiendo una programación más modular.

Tipos de datos:

- r2d2: Enteros.
- c3p0: Flotantes.
- deathStar: Cadenas.
- booleano: Booleanos.

Gramática simplificada:

programa ::= lista_instrucciones

lista_instrucciones ::= (instruccion)*

instruccion ::= declaracion_variable | declaracion_funcion | asignacion | if | while | imprimir | leer | return_stmt | llamada_funcion_stmt | ';'
|---|---|---|---|---|---|---|---|

declaracion_variable ::= tipo identificador (';' | teOtorgamos expresion ';')

declaracion_funcion ::= teInvoco tipo identificador '(' lista_parametros_opcional ')' bloque

return_stmt ::= 'return' expresion ';'

asignacion ::= identificador teOtorgamos expresion ';'

if ::= jedi '(' expresion ')' bloque ('sith' bloque)?

while ::= cloneWar '(' expresion ')' bloque

imprimir ::= palpatine '(' lista_expresiones ')' ';'

leer ::= leia '(' identificador ')' ';'

expresion ::= llamada_funcion | operacion_binaria | literal | identificador

llamada_funcion ::= identificador '(' lista_expresiones_opcional ')'

Tokens y Palabras Clave

- **Funciones**:
  - teInvoco: Palabra clave para la declaración de una función.
  - return: Palabra clave para devolver un valor desde una función.
- **Tipos de Datos**: r2d2, c3p0, deathStar, booleano.
- **Control de Flujo**: jedi (if), sith (else), cloneWar (while).
- **Asignación**: teOtorgamos.
- **Entrada/Salida**: palpatine (imprimir), leia (leer).
- **Operadores Aritméticos**: luz (+), oscuridad (-), unlimitedPower (*), lightsable (/).
- **Operadores de Comparación**: equilibrio (==), rebelde (!=), highground (<), youUnderestimateMyPower (>), padawan (<=), maestro (>=).
- **Valores Booleanos**: obiwan (verdadero), anakin (falso).

Funcionamiento de Cada Módulo

- **scanner.l**: Define patrones para cada token. No sufrió cambios mayores para la implementación de funciones, ya que las nuevas palabras clave (teInvoco, return) se manejan como tokens simples.

- **parser.y (Analizador Sintáctico)**: La gramática fue extendida significativamente para reconocer la sintaxis de teInvoco para las definiciones de funciones y return para las sentencias de retorno. Al reconocer estas estructuras, crea los nodos del AST correspondientes (NodoFuncion, NodoReturn, NodoLlamadaFuncion), enlazándolos correctamente en el árbol.

- **ast.hpp** y **ast.cpp (AST)**: Se añadieron nuevas clases de nodos para manejar funciones:
    - **NodoFuncion**: Representa la definición completa de una función. Almacena el tipo de retorno, el nombre, una lista de parámetros (que son NodoDeclaracionVariable) y el cuerpo de la función (un NodoListaInstrucciones).

    - **NodoLlamadaFuncion**: Representa una llamada a una función. Contiene el nombre de la función y una lista de nodos de expresión para los argumentos.

    - **NodoReturn**: Representa la sentencia return. Guarda la expresión cuyo valor se va a devolver.

- **semantic.cpp (Análisis Semántico)**: El analizador semántico ahora crea un nuevo ámbito en la tabla de símbolos cuando entra a una función, registrando sus parámetros. Verifica que el tipo de valor devuelto en una sentencia return coincida con el tipo de retorno declarado para la función. También valida las llamadas a funciones, comprobando que la función haya sido declarada y que los tipos de los argumentos sean compatibles.

Detalle del AST en ForceCode

El Árbol de Sintaxis Abstracta (AST, por sus siglas en inglés) es el corazón de nuestro compilador. Una vez que el analizador sintáctico (`parser.y`) verifica que el código fuente sigue las reglas gramaticales de ForceCode, no se limita a decir "está bien", sino que construye una estructura de datos en forma de árbol que representa la lógica y el significado del programa. Este árbol elimina los detalles "no esenciales" de la sintaxis (como los puntos y comas o los paréntesis) y se enfoca en la estructura real: qué operaciones se realizan, en qué orden, y sobre qué datos.

¿Para qué sirve el AST? El AST es una estructura intermedia crucial que facilita las fases posteriores del compilador:

1. **Análisis Semántico:** Se recorre el árbol para verificar que el programa tenga sentido. Por ejemplo, que no se usen variables sin declararlas, que los tipos en una operación sean compatibles (no se puede sumar un texto con un número), o que una función se llame con la cantidad correcta de argumentos.

2. **Generación de Código:** Se vuelve a recorrer el árbol, y esta vez, cada nodo se "traduce" a su instrucción equivalente en el lenguaje de destino (en nuestro caso, C). Un nodo de suma se convierte en un “`+`” en C, un nodo `jedi` se convierte en un `if`, y así sucesivamente.

Estructura y Tipos de Nodos Principales: El AST de ForceCode está compuesto por diferentes tipos de nodos, cada uno representando una construcción específica del lenguaje. Todos heredan de una clase base `Nodo`. A continuación, se detallan los más importantes:

- **`NodoPrograma`**: Es la raíz de todo el árbol. Su único hijo es un `NodoListaInstrucciones` que contiene todas las sentencias del programa. Representa el programa completo.

- **`NodoListaInstrucciones`**: Funciona como un contenedor. Almacena una lista (un vector) de todos los nodos de instrucción que se encuentran en un bloque de código (sea el cuerpo principal del programa, el interior de un `jedi`, un `cloneWar` o una función).

- **Nodos de Sentencias:**
  - **`NodoDeclaracionVariable`**: Se crea para una declaración (`r2d2 vidas;`). Almacena el tipo de dato (`r2d2`) y el nombre (`vidas`). Si hay una inicialización (`teOtorgamos 10`), también almacena el nodo de la expresión correspondiente.

  - **`NodoAsignacion`**: Representa una asignación (`vidas teOtorgamos 5;`). Contiene dos hijos: el identificador de la variable a la izquierda y el nodo de la expresión a la derecha.

  - **`NodoSi`**: Para la estructura `jedi-sith`. Tiene hasta tres hijos: la expresión de la condición, un `NodoListaInstrucciones` para el bloque `jedi` (entonces), y un `NodoListaInstrucciones` opcional para el bloque `sith` (sino).

  - **`NodoMientras`**: Para el bucle `cloneWar`. Tiene dos hijos: la expresión de la condición y un `NodoListaInstrucciones` para el cuerpo del bucle.

  - **`NodoImprimir`**: Para `palpatine(...)`. Contiene una lista de nodos de expresión, uno por cada argumento a imprimir.

  - **`NodoLeer`**: Para `leia(...)`. Contiene el `NodoIdentificador` de la variable donde se guardará el dato leído.

Nodos de Expresión:

- **`NodoOperacionBinaria`**: Representa operaciones como `luz` o `highground`. Almacena el operador (`+`, `<`) y tiene dos hijos: las expresiones de la izquierda y la derecha.

- **`NodoIdentificador`**: Representa el uso de una variable en una expresión. Simplemente guarda el nombre de la variable.

- **`NodoLiteral...`** (`Entero`, `Flotante`, `Cadena`, `Booleano`): Son las hojas del árbol. Representan valores constantes y directos en el código, como `10`, `3.14`, `"Hola"` u `obiwan`.

- **Nodos de Funciones (La Novedad Principal):**
  - **`NodoFuncion`**: Se crea para una declaración `teInvoco`. Es un nodo complejo que almacena el tipo de retorno, el nombre de la función, una lista de nodos `NodoDeclaracionVariable` para sus parámetros, y un `NodoListaInstrucciones` para su cuerpo.

  - **`NodoLlamadaFuncion`**: Representa una llamada a una función en una expresión (`suma(5, 3)`). Contiene el nombre de la función que se llama y una lista de nodos de expresión para los argumentos.

  - **`NodoReturn`**: Se crea para la sentencia `return`. Su único hijo es el nodo de la expresión cuyo valor se va a devolver.

Visualización de un Ejemplo: Para el código `r2d2 x teOtorgamos 10;`, el parser construiría este simple AST:

```
NodoPrograma
└── NodoListaInstrucciones
    └── NodoDeclaracionVariable (tipo: r2d2, nombre: x)
        └── Inicialización: NodoLiteralEntero (valor: 10)
```

Esta estructura jerárquica es la que permite al compilador entender la semántica del código más allá del texto plano y actuar en consecuencia para validarlo y traducirlo.

Proceso de generación de código

La generación de código es la fase final del compilador, donde el Árbol de Sintaxis Abstracta (AST), ya validado semánticamente, se traduce a un lenguaje de más bajo nivel, en este caso, C. Este proceso se realiza mediante un recorrido recursivo del AST, donde cada nodo es responsable de generar su propio fragmento de código.

El Recorrido Recursivo del AST: El proceso se inicia llamando a un método `generarCodigo()` en el nodo raíz del AST (`NodoPrograma`). Este método, a su vez, llama a los métodos `generarCodigo()` de sus nodos hijos, y así sucesivamente hasta llegar a las hojas del árbol (los literales e identificadores). La combinación de los fragmentos de código generados en cada llamada recursiva produce el programa completo en C.

Traducción de Nodos Específicos:

- **`NodoPrograma`**: Genera el código de cabecera, como los `#include` necesarios (`stdio.h`, `windows.h` para compatibilidad), y luego invoca la generación de código para la lista principal de instrucciones. Si no se define una función `main` en el código ForceCode, este nodo genera una función `main()` por defecto que envuelve las instrucciones globales.
- **Tipos y Variables (`NodoDeclaracionVariable`)**: Se mapean los tipos de ForceCode a sus equivalentes en C:
    - `r2d2` se convierte en `int`.
    - `c3p0` se convierte en `float`.

- `deathStar` se convierte en `char*` (puntero a caracter).

- `booleano` se convierte en `int` (usando `1` para `obiwan` y `0` para `anakin`)

- Una declaración como `r2d2 vidas teOtorgamos 3;` se traduce a `int vidas = 3;`.

- **Expresiones (`NodoOperacionBinaria`, `NodoLiteral`, `NodoIdentificador`)**:

  - Un `NodoLiteral` simplemente imprime su valor (ej. `10`).

  - Un `NodoIdentificador` imprime el nombre de la variable (ej. `vidas`).

  - Un `NodoOperacionBinaria` genera el código de su operando izquierdo, luego el operador C correspondiente (`luz` -> `+`), y finalmente el código de su operando derecho, todo ello envuelto en paréntesis para preservar la precedencia. Ej: `(vidas + 10)`.

- **Estructuras de Control (`NodoSi`, `NodoMientras`)**: Se traducen a sus contrapartes directas en C.
-

  - `jedi (condicion) { ... } sith { ... }` se convierte en `if (condicion_c) { ... } else { ... }`.

  - `cloneWar (condicion) { ... }` se convierte en `while (condicion_c) { ... }`. El generador de código invoca recursivamente la generación para las expresiones de condición y los bloques de instrucciones.

- **Funciones (`NodoFuncion`, `NodoLlamadaFuncion`, `NodoReturn`)**:
    - `NodoFuncion`: Genera la signatura completa de la función en C, traduciendo el tipo de retorno, el nombre y la lista de parámetros. Luego, invoca la generación de código para el cuerpo de la función.

    - `NodoLlamadaFuncion`: Genera el nombre de la función seguido de paréntesis. Dentro de los paréntesis, invoca recursivamente la generación de código para cada una de las expresiones que actúan como argumentos, separadas por comas.

    - `NodoReturn`: Se traduce a la palabra clave `return` de C, seguida de la generación de código para la expresión que se devuelve.

- **Entrada y Salida (`NodoImprimir`, `NodoLeer`)**:
    - `palpatine(...)` (`NodoImprimir`) se traduce a una llamada a `printf()`. El primer argumento de `printf` es una cadena de formato que se construye dinámicamente según los tipos de las expresiones a imprimir (`%d` para enteros y booleanos, `%f` para flotantes, `%s` para cadenas).

    - `leia(var)` (`NodoLeer`) se traduce a una llamada a `scanf()`, usando el especificador de formato adecuado (`%d`, `%f`, etc.) y pasando la dirección de la variable (`&var`).

Este enfoque modular y recursivo asegura que cada parte del lenguaje ForceCode se traduzca de manera consistente y correcta al lenguaje C, produciendo un programa ejecutable que replica la lógica del código fuente original.

Ejemplos de programas y código generado

Ejemplo en ForceCode (basado en `calculadora.txt`):

```
teInvoco r2d2 suma(r2d2 a, r2d2 b) {
    return a luz b;
}

teInvoco r2d2 main() {
    r2d2 x teOtorgamos 5;
    r2d2 y teOtorgamos 7;
    r2d2 z teOtorgamos suma(x, y);
    palpatine("Suma:", z);
    return 0;
}
```

Código C generado (`output.c`):

```
#include <stdio.h>
#ifdef _WIN32
#include <windows.h>
#endif

int suma(int a, int b) {
    return (a + b);
}

int main() {
    #ifdef _WIN32
    SetConsoleOutputCP(CP_UTF8);
    #endif
    int x = 5;
    int y = 7;
    int z = suma(x, y);
    printf("Suma: %d\n", z);
    return 0;
}
```

Manual de usuario de ForceCode

1. Conceptos Básicos:

Todo programa en ForceCode se construye a partir de instrucciones. Cada instrucción debe terminar con un punto y coma (;), como un golpe preciso de un sable de luz.

Comentarios: Para dejar notas que el compilador ignorará, se expresa de la siguiente manera:

- `// Comentario de una sola línea`

- `/* Un bloque de comentario que puede ocupar varias líneas */`

2. Variables:

Las variables son contenedores para almacenar datos. Primero se deben declarar, especificando su tipo.

Tipos de Datos:

- `r2d2`: Para números enteros (ej. `10`, `-5`, `138`).

- `c3p0`: Para números de punto flotante o decimales (ej. `3.14`, `-0.5`).

- `deathStar`: Para cadenas de texto (ej. `"Hola, galaxia"`).

- `booleano`: Para valores de verdad, que solo pueden ser `obiwan` (verdadero) o `anakin` (falso).

Declaración:

```
// Declarando variables sin valor inicial

r2d2 naves;

c3p0 energia;

deathStar mensaje;

booleano esUnJedi;
```

Asignación:

`teOtorgamos` Para dar un valor a una variable, usa la palabra clave `teOtorgamos`. Puedes hacerlo en la misma línea de la declaración o después.

```
// Declaración con asignación inmediata

r2d2 vidas teOtorgamos 3;
deathStar planeta teOtorgamos "Tatooine";

// Asignación posterior

c3p0 precision;

precision teOtorgamos 99.9;
```

3. Operaciones:

Puedes realizar operaciones aritméticas y de comparación.

Operadores Aritméticos:

- `luz`: Suma (`+`)
- `oscuridad`: Resta (`-`)
- `unlimitedPower`: Multiplicación (`*`)
- `lightsable`: División (`/`)

```
r2d2 totalDroides;

totalDroides teOtorgamos 5 luz 3; // totalDroides ahora es 8
```

Operadores de Comparación: Estos operadores siempre devuelven un valor `booleano`.

- `equilibrio`: Igual que (`==`)
- `rebelde`: Diferente de (`!=`)
- `highground`: Menor que (`<`)
- `youUnderestimateMyPower`: Mayor que (`>`)
- `padawan`: Menor o igual que (`<=`)
- `maestro`: Mayor o igual que (`>=`)

```
booleano esPoderoso;

esPoderoso teOtorgamos 100 youUnderestimateMyPower 50;

 // esPoderoso es 'obiwan' (verdadero)
```

4. Control de Flujo: Puedes tomar decisiones y repetir acciones con las estructuras de control.

Condicionales: `jedi` y `sith` Equivalente a `if-else`. El bloque `jedi` se ejecuta si la condición es verdadera (`obiwan`). Si no, se ejecuta el bloque `sith` (si existe).

```
r2d2 midiclorianos teOtorgamos 15000;

jedi (midiclorianos youUnderestimateMyPower 10000) {

    palpatine("Eres sensible a la Fuerza.");
} sith {

    palpatine("La Fuerza no es intensa en ti.");
}
```

Bucles: `cloneWar` Equivalente a `while`. El bloque de código se repite mientras la condición sea verdadera.

```
r2d2 clones teOtorgamos 1;

cloneWar (clones padawan 5) {

   palpatine("Clon número:", clones);

   clones teOtorgamos clones luz 1; // ¡Cuidado con los bucles infinitos!
}
```

5. Funciones:

Las funciones te permiten agrupar código en bloques reutilizables.

Declaración: `teInvoco` Para crear una función, usa `teInvoco`, seguido del tipo de valor que devolverá, su nombre, sus parámetros entre paréntesis y su cuerpo entre llaves.

```
// Esta función recibe dos números y devuelve su suma

teInvoco r2d2 calcularSuma(r2d2 num1, r2d2 num2) {

   r2d2 resultado;

   resultado teOtorgamos num1 luz num2;

   return resultado; // 'return' devuelve el valor
}
```

Llamada a Funciones: Para usar una función, simplemente escribe su nombre seguido de los argumentos entre paréntesis.

```
r2d2 total;

// Llamamos a la función y guardamos el resultado

total teOtorgamos calcularSuma(10, 20); // total ahora es 30

palpatine("El resultado es:", total);
```

Función `main`: Si tu programa contiene una función llamada `main`, esa será la primera en ejecutarse. Es el punto de entrada de tu aplicación.

```
telnvoco r2d2 main() {

   palpatine("¡Que la Fuerza te acompañe!");

   return 0; // Por convención, devuelve 0 si todo salió bien
}
```

6. Entrada y Salida:

- **palpatine(...)**: Para mostrar mensajes o valores en la consola. Puede aceptar múltiples argumentos separados por comas.

- `leia(...)`: Para leer un valor introducido por el usuario y guardarlo en una variable.

  ```
  r2d2 edad;

  palpatine("Saludos, joven Padawan. ¿Cuál es tu edad?");

  leia(edad);

  palpatine("Entiendo. Tienes", edad, "años.");
  ```

Errores y mensajes del compilador

- **Error léxico**: Palabra desconocida o carácter inválido.
  - Ej: `Error léxico: Carácter inesperado '@'`
- **Error sintáctico**: Estructura incorrecta, falta de punto y coma o llave.
  - Ej: `Error de sintaxis cerca de 'sith'.`
- **Error semántico**: Variable no declarada, incompatibilidad de tipos, redeclaración de variables.
  - Ej: `Variable no declarada: 'midiclorianos'`
  - Ej: `Asignación de tipo incompatible a variable: 'fuerza'`
  - Ej: `Variable redeclarada: 'vidas'`

Conclusión

La implementación de funciones en ForceCode representa un avance crucial en su desarrollo, transformándolo de un lenguaje puramente script a uno con capacidades de programación estructurada. El diseño modular del compilador, basado en Flex, Bison y C++, ha facilitado la extensión de la gramática, el AST y el análisis semántico para incorporar esta nueva funcionalidad. El resultado es un compilador más robusto y un lenguaje mucho más potente y versátil, manteniendo siempre su divertida temática de Star Wars. Los siguientes pasos podrían incluir la implementación de arreglos, estructuras de datos más complejas o un sistema de módulos.