Proyecto de Compilación
Compilador de HULK a CIL

Integrantes:
Raimel Daniel Romaguera Puig C312
Manuel Alejandro Gamboa Hernández C311
Yoel Enríquez Sena C311
David Sánchez Iglesias C311

Fecha:
5 de julio de 2024

1 Introducción

En este documento se presenta el informe del proyecto de compilación, el cual consiste en la implementación de un compilador de HULK a CIL. Un compilador es un programa, cuya entrada y salida resultan ser también programas. La entrada son líneas de código de un lenguaje de alto nivel (en este caso HULK), mientras que la salida son líneas de un lenguaje de bajo nivel (en este caso CIL, Common Intermadiated Language). Este proyecto fue implementado usando el lenguaje de programaci'on Python.

2 Arquitectura del compilador

Esta propuesta de compilador consta de 3 fases: Análisis Léxico, Análisis Sintáctico y Análisis Semántico. A continuación se describen las características de cada una de estas fases.

2.1 Análisis Léxico o Lexer

En esta fase se realiza la lectura del código fuente y se identifican los tokens que lo conforman. Para ello se creó un archivo llamado *lexer.py* donde se realiza un análisis léxico donde se evalúan las reglas del código fuente y crea una lista de tokens, donde cada token es una expresión regular. Durante este proceso se construye un árbol que representa la jerarquía de los tokens, y que al final crea un autómata finito determinista (DFA) que reconoce las diferentes entidades del lenguaje HULK (identificadores, números, operadores, etc.). Al final, devuelve un flujo de tokens que serán usados en la siguiente fase del compilador.

2.1.1 Gramática

La gramática es la responsable de definir la estructura y las reglas de sintaxis del lenguaje de programación que el compilador va a traducir. La gramática es, sencillamente, el conjunto de reglas que definen cómo se pueden combinar las palabras y símbolos de un lenguaje para formar sentencias válidas. Esta estructura es esencial para el análisis sintáctico, una de las fases críticas en el proceso de compilación.

Durante el análisis sintáctico, el compilador verifica si el código fuente proporcionado sigue las reglas gramaticales establecidas para el lenguaje de programación. Esto incluye verificar la correcta aparición de palabras clave, la estructura de las declaraciones, la coincidencia de paréntesis y corchetes, y otras reglas específicas del lenguaje. Si el código fuente no cumple con estas reglas, el compilador generará un error, indicando que el código no es sintácticamente correcto

La gramática también permite la generación de un árbol sintáctico, que es una representación jerárquica del programa fuente. Este árbol es fundamental para las capas posteriores que posee el compilador, como

la optimización de código y la generación de código objetivo. El árbol sintáctico refleja la estructura gramatical del programa, permitiendo al compilador entender y manipular el código fuente de manera eficiente.

La gramática ayuda a definir el alcance de variables, controlar el flujo de ejecución del programa a través de estructuras de control, como bucles y condicionales, y manejar la declaración y uso de funciones. Todas estas características del lenguaje de programación deben ser capturadas y validadas por el compilador a través de su gramática

La gramática reconocedora de expresiones regulares implementada por nuestro equipo, se puede ver en el archivo *algo.py*

2.2 Análisis Sintáctico o Parser

Para esta fase se crearon dos archivos fundamentales: *parsers.py* y *grammar.py*. En el primero se implementaron una serie de parsers (todos dados en clase), no obstante en el proyecto solo se usan dos: el parser LL1 durante el lexer como parser izquierdo, y el LR1 para el parser del lenguaje HULK. En el segundo archivo se definió la gramática de HULK, la cual se presenta a continuación.

La grmática de HULK propuesta por nosotros se puede encontrar en el archivo *grammar.py*.

Esta implementación usa también la clase *Grammar*, y crea los nodos de la gramática de HULK. El parser LR1 se encarga se construir el AST (Abstract Syntax Tree) del programa que se pasó como entrada.

2.3 Análisis Semántico

El chequeo semántico es una etapa crítica en el proceso de compilación o interpretación de programas, enfocada en garantizar la corrección y coherencia de los tipos de datos utilizados dentro del código fuente. Este proceso asegura que todas las operaciones se realicen entre tipos de datos compatibles, que las variables y funciones se utilicen correctamente según sus definiciones, y que los tipos de datos especificados sean apropiados para cada operación o estructura de datos. El chequeo semántico se realiza mediante el análisis del Árbol de Sintaxis Abstracta (AST) generado durante la fase de análisis sintáctico. A medida que se recorre el AST, se realizan comprobaciones específicas para cada tipo de nodo, incluyendo pero no limitado a:

- *Operadores Binarios*: Verifica que los operandos sean compatibles con el operador aplicado.
- *Bloques de Código*: Asegura que todas las variables y funciones declaradas dentro del bloque estén definidas antes de su uso.
- *Accesos a Miembros de Clases*: Confirma que los objetos accedidos posean los atributos solicitados.
- *Declaraciones Condicionales*: Revisa que las condiciones y los cuerpos de las declaraciones condicionales sean compatibles en términos de tipos de datos.
- *Constantes*: Asegura que los valores asignados a las constantes sean del tipo esperado.
- *Asignaciones Destructivas*: Verifica que los tipos de datos de las variables asignadas sean compatibles con los valores recibidos.
- *Declaraciones de Funciones*: Comprueba que los tipos de parámetros y el valor de retorno sean consistentes con la definición de la función.
- *Invocaciones de Funciones*: Asegura que los argumentos pasados a las funciones sean compatibles con los tipos de parámetros esperados.
- *Declaraciones* `let`: Gestiona el alcance de las variables declaradas y verifica su uso adecuado.
- *Valores Nuevos*: Asegura que los tipos de datos de los nuevos valores sean compatibles con el contexto en el que se utilizan.
- *Parámetros*: Verifica que los tipos de datos de los parámetros sean compatibles con los tipos esperados por las funciones o métodos.

- *Declaraciones de Protocolos*: Asegura que las implementaciones de protocolos cumplan con las especificaciones definidas.
- *Declaraciones de Tipos*: Verifica que los tipos de datos declarados sean utilizados correctamente en el resto del código.
- *Operadores Unarios*: Comprueba que los operandos sean compatibles con el operador unario aplicado.
- *Valores de Variables*: Asegura que los valores asignados a las variables sean compatibles con sus tipos declarados.