Compilador de Hulk

Carlos Manuel González Peña C411
Jorge Alberto Aspiolea C412
Alex Sánchez Saez C412

8 de abril de 2024

1. Uso del compilador

1.1. Compilar un archivo .hulk

Puede ejecutar el comando **python . –file ./my-code.hulk** para ejecutar el codigo en el fichero **./my-code.hulk**.

2. Estructura del compilador

2.1. Lexer

Para el Lexer se usó un generador de expresiones regulares básicas compuesta por los operadores **| ( ) * symbol** ϵ, y se usó una gramática **LL(1)** para parsear el lenguaje de dichas expresiones regulares. También se tomaron todos los tokens que conforman el lenguaje hulk, se crearon sus expresiones regulares y mediante las operaciones con autómatas se unieron todas las expresiones en un único autómata que reconoce todos los tokens del lenguaje. La definición de la gramática del generador de expresiones regulares se encuentra en el archivo *cmp/tools/regex.py*.

2.2. Gramática del Hulk

La gramática del Hulk es una gramática libre del contexto no ambigua que cumple las reglas de un parser **LR(1)**. La definición de la gramatica se encuentra dentro del código en el archivo */cmp/core/grammar.py*

2.3. Parser

Se usó un parser **LR(1)** el cual recibe una lista de tokens obtenidos mediante la operación de lexer y este devuelve un árbol de derivación derecha y un árbol de sintaxis abstracta gracias a la atributación de la gramática.

2.4. Chequeo semántico

En la recolección de tipos se hace un primer recorrido por el AST, donde se definen los tipos iniciales o por defecto que tiene un programa de Hulk. Agregamos además nuestro tipo global el cual encapsulará las variables y funciones definidas en el programa con sus respectivos tipos de dato. Para inferir los tipos agregamos un método `type_of` a cada nodo del AST el cual devolverá el tipo de cada nodo, y en cada clase concreta definimos cómo se calcula ese tipo. Por ejemplo, el `type_of` de una función será el tipo que retorne la expresión más a la derecha de ella, en el caso de la declaración de variables se le asigna a la variable el tipo de dato que devuelve la expresión que está después del =. Todos los tipos de datos tienen como padre común el tipo `Object` por lo que todos conforman con `Object`.

Chequeo de la semántica en el chequeo semántico se hace un recorrido por el AST en el nodo inicial `'ProgramNode'` se inicializa el scope y se revisa cada instrucción definida en el programa. En este Scope vienen definidas las funciones built-in del programa tales como `tan`, `sin`, `cos`, `print` y las constantes como `pi`. En el scope se guarda la información de qué variables están definidas en el scope, qué métodos y con qué cantidad de argumentos, y también un scope puede tener y conocer sus scopes hijos, así como su único scope padre. Así utilizando el patrón visitor al igual que en la sección anterior visitamos cada nodo del AST (simulando así un recorrido de árbol) y chequeamos que las variables solo se usen después de estar definidas, así como las funciones y sus variables internas. Uno de los problemas con los que chocamos al crear esta parte del compilador fue los parámetros de las funciones, los cuales deben ser definidos en el scope hijo de la función antes de visitarla, o cuando se usen se obtendrá un error semántico.

Chequeo de tipos En este fragmento del compilador chequeamos que los tipos utilizados en las diferentes partes del programa fueran consistentes. Una vez recolectados e inferidos los tipos solo resta chequear que en las operaciones se estén utilizando tipos que acomoden entre sí y que en las llamadas a métodos y atributos de las variables sean consistentes con las que existen en su respectivo tipo, todo esto utilizando el contexto que se creó en el primer recorrido durante la recolección de tipos. Durante todo el proceso (estas tres fases) se almacenan los errores generados por los checkers para luego mostrárselos a los usuarios. Una vez pasadas estas fases, sin errores, entonces el AST está listo para ser traducido a otro lenguaje o interpretarse directamente.

2.5. Generación de código