HULK — **H**avana **U**niversity **L**anguage **K**ompiler

Diego A. Martínez Jiménez

Introdución

HULK es un lenguaje de programación imperativo, funcional, estática y fuertemente tipado. Casi todas las instrucciones en HULK son expresiones. En particular, el subconjunto de HULK implementado se compone solamente de expresiones que pueden escribirse en una línea.

Un Intéprete

A la hora de interpretar un lenguaje el código fuente pasa por varias fases. A continuación una breve descripición de cada una.

+ Análisis léxico

El análisis léxico constituye la primera fase, aquí se lee el programa fuente de izquierda a derecha y se agrupa en componentes léxicos (tokens), que son secuencias de caracteres que tienen un significado. Además, todos los espacios en blanco, líneas en blanco, y demás información innecesaria se elimina del programa fuente. También se comprueba que los símbolos del lenguaje (palabras clave, operadores, etc.) se han escrito correctamente.

+ Análisis sintáctico/Parser

En esta fase los caracteres o componentes léxicos se agrupan jerárquicamente en frases gramaticales que el compilador utiliza para sintetizar la salida. Se comprueba si lo obtenido de la fase anterior es sintácticamente correcto (obedece a la gramática del lenguaje). Por lo general, las frases gramaticales del programa fuente se representan mediante un árbol de análisis sintáctico.

+ Análisis semántico

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

Un componente importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si cada operador tiene operandos permitidos por la especificación del lenguaje fuente, entre otros.

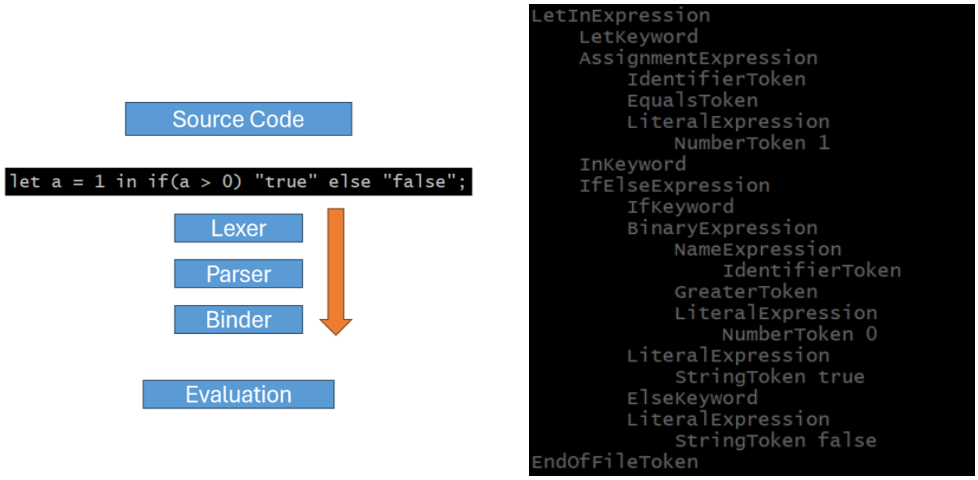
+ Evaluación

En la fase de evaluación, el intérprete ya tiene todo la información necesaria para saber de qué forma evaluar cada expresión. Por lo que en esta fase ya ejecutará el código fuente de la forma esperada o, en caso de haber algun error en el código fuente, los reportará.

Intérprete de HULK

+ Análisis del código

El codigó fuente se analiza como se explicó anteriormente. Para esto existen tres clases principales `Lexer`, `Parser` y `Binder`. Estas clases se encargan del analisis lexico, sintactico y semantico respectivamente. A continuación una breve descripición de su funcioanmiento.



Estructura del intérprete y AST

+ Clase Lexer

Esta clase se encarga de tokenizar el código. Primero almacena el codigo en la variable estatica luego con la funcion `Lex()` retorna el proximo `SyntaxToken`, siendo `SyntaxToken` la clase padre que representa todos los tokens. En esta clase se usan algunos metodos auxiliares de la clase `SyntaxFacts`; por ejemplo `SyntaxFacts.GetText(SyntaxKind kind)` y `SyntaxFacts.GetKeywordKind(string text)`, mas adelante se explicara esta clase. Haciendo uso de la clase `DianosticsBag` se reportan los errores lexicos que se encuentren a la hora de analizar la cadena.

+ Clase Parser

La classe Parser se encarga de crear el AST. El constructor de esta clase recible el codigo, luego hace uso de la clase Lexer para almacenar todos los tokens ignorando los espacios en blanco y los tokens desconocidos; en el constructor tambien son almacenados los errores encontrados en el analisis lexico para luego unirlos a los que errores que se encuentren en esta parte del analisis. Una vez construida una instancia se hace uso del metodo publico `ParseCompilationUnit()` que se encarga de inciar el proceso de analisis sintactico. En este procesos primero se analiza si se la expresion es la declaracion de una funcion, en caso de que no se analiza si es una expresion binaria; es decir que sea una expresion tipo *left + right* por ejemplo, para lo cual se analiza la parte *left* la cual sera una expresion primaria (expresion primaria se refiere a las expresiones que deben parsearse antes de una expresion binaria o una expresion unaria) o una expresion unaria, en el caso de una expresion unaria se analiza la precedencia del operador de la expresion unaria y a partir de la precedencia se parseara antes o despues de la expresion binaria (en principio todos los operadores unarios implementados en el proyecto tienen la misma precendencia y tambien precedencia mayor que todos los operadores binarios). Luego de analizar *left*, se analiza el operador binario en cuanto a su precedencia, luego analiza *right* de similar forma a *left* y asi se obtiene la expresion binaria y en caso de no serlo *left* sera el tipo de expresion que luego habra que se tendra que luego analizar semanticamente.

+ Clase Binder

Esta clase se encarga del analisis semantico del AST. El metodo `BindExpression()` de esta clase se encarga de crear una `BoudExpression` la cual sera la expresion con toda la información necesaria para evaluar el código correctamente y haciendo la menor cantidad de verificaciones. Este metodo recibira el root del AST y se ira analizando recursivamente. Las clases `BoundUnaryOperator` y `BoundBinaryOperator` son esenciales para el asegurar que los operadores de una expresion unaria o binaria esten siendo correctamente usados; con el metodo `Bind()` cada una de las clases asegura la existencia de el operador con los tipos especificados. De forma similar se verifican las funciones predefinidas mediante la clase `BoundPredifinedFunction`.Las expresiones de asignacion infieren el tipo a partir de lo que se encuentra a la de derecha del `=`.

+ Clase Evaluator

Luego del analisis semantico se evalua el codigo mediante la clase `Evaluator`. Similar a la clase `Binder` esta clase analiza recursivamente el AST pero ahora en vez de analizar la semantica evalua las expresiones, hasta tener un literal que sera el resultado de la expresion evaluada.

*

Para este proyecto se uso como guía una **playlist de YouTube** acerca de como crear un compilador.