Instituto Tecnológico y de Estudios Superiores de Monterrey

**Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 501)**

**A blue and black logo

Description automatically generated**

**Proyecto de compiladores**

Leonardo Mojica Amézquita - A00571960

07/05/2024

Este proyecto consiste en generar un lenguaje de programación básica.

Se basa en el siguiente diagrama de “Little\_Duck”.

A diagram of a computer program

Description automatically generated**Lista de tokens:**

Símbolos:

* (): paréntesis para encerrar expresiones, y para el uso de funciones.
* {}: llaves que se usan para dar inicio y fin de un “body”.
* []: Corchetes que se usan para encerrar la lógica de una función.
* ;: punto y coma que sirve como un símbolo terminador, el cual se pone cada que termines una asignación de id, llames a una función, declares una función, termines un ciclo, hagas una condición, asignación, etc.
* +: símbolo de suma, se usa para sumar variables.
* -: simbol de resta, se usa para restar variables.
* \*: asterisco que se usa como símbolo de multiplicación.
* /: símbolo que se usa para división.
* =: símbolo de asignación.
* ,: coma, se usa como separador, por ejemplo, para declarar varias variables, en una sola línea, o para poner los parámetros de una función.
* <: menor que, símbolo para comparar dos expresiones.
* >: mayor que, símbolo para comparar dos expresiones.
* !=: símbolo diferente de para comparar dos expresiones
* :: dos puntos que van entre id y su tipado, se usa para especificar el tipado de una variable.

Palabras reservadas:

* program: indica el inicio del programa
* main: indica el inicio del bloque de código principal de tu programa
* end: indica el final del main
* var: indica el inicio de una declaración de variable
* int: indica que tu variable es un entero
* float: indica que tu variable es flotante
* print: función reservada para mostrar información en consola
* if: indica el inicio del condicional if
* else: indica el inisio de lo que se hace en caso de que no se cumpla el if
* void: indica el inicio de una función
* while: indica el inicio de un ciclo while, donde se pondrá la condicional
* do: indica el inicio del cuerpo del while donde se indicará lo que se hará mientras el while se cumpla

Tokens con regex:

En estos tokens uso el separador | como or para especificar las posibles opciones de rangos, pero esto no se suele usar, tengo pensado usar ANTLR4 para desarrollar el compilador, y este puede generar conflictos a la hora de crear las expresiones regulares, así que se deben quitar los |.

* id
  + El id tiene que empezar con una letra (mayúscula o minuscula) y puede seguir con 0 o n letras más (mayúscula o minúscula), o 0 o n números o 0 o n guiones bajo, en cualquier orden.
  + [A-Z|a-z] [A-Z|a-z|0-9|\_]\*
* cte.string
  + una cadena de caracteres (cualquier carácter menos salto de linea), la cual esta encerrada por dos comillas dobles (“”), o dos comillas sencillas (‘’).
  + ("[^\n|"]\*") | ('[^\n|']\*'), el símbolo ^ representa todos los caracteres menos la lista de caracteres a continuación.
  + Esta expresión tiene que ser reescrita para que funcione en antlr4, la cual queda de la siguiente forma: ’ (~[\n]\*) ’ | ” (~[\n]\*) ”, ambos lados del or | son iguales solo cambiando las comillas, para que acepte tanto comillas simples como sobles, adentro cambiamos a ~[\n], el símbolo ~ es un negador, por lo que está diciendo que no acepte lo que esta dentro de los corchetes, que es un salto de línea.
* cte.int
  + Toma valor de números enteros decimales.
  + [0-9]+
* cte.float
  + Únicamente acepta valores numéricos seguidos de un único punto (.) seguido de 1 o más caracteres numéricos
  + [0-9]+ (.) [0-9]+

**Gramática libre de contexto**

<Program> 🡪 program id ; <a\_vars> <a\_funcs> main <BODY> end

<a\_vars> 🡪 ε | <VARS>

<a\_funcs> 🡪 ε | <FUNCS> <a\_funcs>

<VARS> 🡪 var <list\_vars>

<list\_vars> 🡪 id <list\_id> : <TYPE>; <more\_vars>

<more\_vars> 🡪 ε | id <list\_id> : <TYPE>; <more\_vars>

<list\_id> 🡪 ε | , id

<TYPE> 🡪 int | float

<Body> 🡪 {<List\_statements>}

<List\_statements> 🡪 ε | <STATEMENT> <List\_statements>

<STATEMENT> 🡪 <ASSIGN> | <CONDITION> | <CYCLE> | <F\_Call> | <Print>

<Print> 🡪 print (<List\_Expresion>);

<List\_Expresion> 🡪 <exp\_o\_string> |<exp\_o\_string> , <List\_Expresion>

<exp\_o\_string> 🡪 <EXPRESION> | cte.string

<ASSIGN> 🡪 id = <EXPRESION> ;

<CYCLE> 🡪 while <Body> do ( <EXPRESION> );

<CONDITION> 🡪 if (<EXPRESION>) <Body> <ELSE>;

<ELSE> 🡪 else <Body> | ε

<EXPRESION> 🡪 <EXP> < comparar\_exp>

<comparar\_exp> 🡪 < <EXP> | > <EXP> | != <EXP> | ε

<EXP> 🡪 <List\_Terminos>

<List\_Terminos> 🡪< TERMINO > <next\_termino>

<next\_termino> 🡪 <sum\_rest> < TERMINO ><next\_termino> | ε

<sum\_rest> 🡪 + | -

<TERMINO> 🡪 <List\_Factor>

<List\_Factor> 🡪<FACTOR> <next\_Factor>

<next\_Factor> 🡪 <mult\_divt> <FACTOR><next\_factor> | ε

<mult\_div> 🡪 \* | /

<FACTOR> 🡪 (<EXPRESION>) | <sum\_rest><id\_cte> |<id\_cte>

<id\_cte> 🡪 id | <CTE>

<CTE> 🡪 cte\_int | cte\_float

<FUNCS> 🡪 void id (<list\_params>) [<var\_no\_var> <BODY>];

<list\_params> 🡪 id : <TYPE> | <more\_params>

<more\_params> 🡪 ε | , id : <TYPE> <more\_params>

<var\_no\_var> 🡪 ε | <VARS>

<F\_Call> 🡪 id (f\_list\_expresion);

<f\_list\_expresion> 🡪 ε | <EXPRESION> <f\_more\_expresion>

<f\_more\_expresion> 🡪 ε | , <EXPRESION> <f\_more\_expresion>

**Opciones de herramientas de generación automática de compiladores**

Estas herramientas fueron escogidas para poder hacer la implementación en Python.

Encontré el siguiente git, el cual muestra que herramientas de Python puedo usar para todas las etapas de un compilador. <https://gist.github.com/dimitrio-m/cf6cde052787ed97164fe6422a5e4cb0>

3 de las tecnologías que encontré para el análisis léxico y sintáctico es PLY (Python Lex-Yacc), ANTLR y Lark.

PLY:

Es una implementación fiel a lex and yacc. El principal objetivo de PLY es ser lo

más fiel posible la forma en la que herramientas lex/yacc funcionan, como soportar LALR (look-ahead, left to right, rightmost derivation parser) parser y proveer validación de input, reporte de errores y diagnósticos.

PLY cuenta con dos módulos, lex.py y yacc.py dentro del paquete de ply. Lex.ply se usa para el análisis léxico seccionando el input del texto en colecciones de tokens, y el yacc.py se usa para reconocer el lenguaje sintáctico que se debió haber especificado cono una gramática libre de contexto. En general ambos módulos están diseñados para trabajar en conjunto, por ejemplo lex.py tiene una interface en forma de una función token() la cual regresa el próximo token valido del input. Yacc.py normalmente regresa un AST (abstract syntax tree), pero el desarrollador puede cambiarlo para ser implementado como un “simple one-pass compiler”.

ANTLR:

Es un generador para procesar texto estructurado o archivos binarios. Es usado para en base a una gramática, generar un parser que pueda crear y caminar un árbol de parse. ANTLR está dividido en dos partes, el ANTLR tool y la ANTLR runtime libraary. La primera corre en JAVA y genera un parser en base a una gramática. El segundo es la parte que nos ayuda a generar el parser en Python.

LARK:

LARK es una librería de parseo para Python, el cual puede parsear basado en una gramática libre de contexto. Acepta gramáticas basadas en EBNF (Extended Backus-Naur Form), cuenta con algoritmos de parseo de árbol (Earley, LALR and CYK), crea de forma automática un árbol inferido de la gramática, y cuenta con un analizar léxico que soporte expresiones reculares.

**Herramienta ANTLR4 (para python)**

Decidí usar ANTLR4, ya que al haberlo probado un poco y al leer su documentación me di cuenta de que es sencillo de usar y me ofrece muchas herramientas para validar mis gramáticas, por ejemplo, puedo pedirle que me muestre de forma gráfica un árbol de mi input de la gramática.

Para el uso de la herramienta se uso la documentación del git oficial de antlr: <https://github.com/antlr/antlr4/blob/master/doc/getting-started.md>

Para el uso de antlr se necesita tener instalado Python y pip.

En la línea de comandos se debe ejecutar

* pip install antlr4-tools

En Windows puede haber problemas de no poder ejecutar los comandos de antlr4 y es porque la dirección de la herramienta no suele agregarse al path de las variables de sistema, por lo que se debe agregar manualmente. Al intentar ejecutar antlr4 debería salir un error diciendo que agregues la dirección al path, pero si no llega a salir el path que se debe agregar es c:\Users...\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.10\_qbz5n2kfra8p8\LocalCache\local-packages\Python310\Scripts

Después de agregar el path debes reiniciar la computadora.

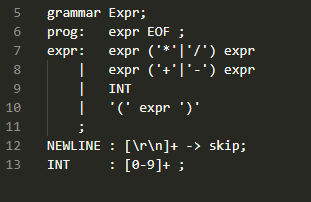
Cuando reinicies ejecuta en terminal.

* Antlr4

Si no tienes java instalado te preguntara si quieres instalarlo, debes dar que si ya que antlr necesita java para correr.

Una ves instale todo lo necesario para usar antlr, intente algunos de los ejemplos de la documentación que me sirvieron para decidir usar esta herramienta.

Lo primero que hice fue crear un documento Expr.g4 el cual contiene el ejemplo de la gramática a analizar. El contenido del documento es el siguiente:



Como se puede observer, se debe inicial con el nombre de tu gramática, luego pones las variables con las reglas de cada una de estas expresiones, al inicio se pone la expresión inicial que en el ejemplo es prog la cual consiste de un “expr” seguido del EOF (end of file). La expresión expr muestra nos deja agregar sumas, restas, multiplicaciones o divisiones entre más expr, además expr, puede ser un entero (INT) o estar encerrado entre paréntesis, ANTLR permite poner las expresiones con ors | para dar las diferentes posbiles expresiones. Al final se ponen los tokesn en formato de expresión regular, como podemos ver en el ejemplo se tiene INT el cual acepta símbolos de 0 a 9 una o más veces, además tenemos el token NEWLINE que lo único que hace es tener la lista de terminadores para ignorarlos, o sea que si encuentra \r \n (tab o newline) haga un “skip”.

Con este documento creado, pude ejecutar el comando:

* antlr4-parse Expr.g4 prog

el cual me analiza la gramática contenida en Expr.g4, además, al final del comando, agregue “-gui”, “-tree”, “-tokens -trace”, dependiendo de que comando use puedo conseguir diferentes outputs, por ejemplo, si uso -gui me muestra gráficamente el árbol que se genera.

Una vez corro el comando, me deja poner el input que quiero analizar y después debo dar ctrl+z (“end of input”) para indicar que termine de poner el input, y a continuación me regresara el resultado.

El ejemplo que corri fue 10\*2+20, con la gramática de prueba presentada anteriormente, usando -gui obtuve como resultado:

A screenshot of a computer

Description automatically generated

Como pueden ver se genera el árbol de la expresión que se analizo y muestra el nombre de los tokens. Este mismo ejemplo con -tree regresa:

(prog:1 (expr:2 (expr:1 (expr:3 10) \* (expr:3 2)) + (expr:3 20)) <EOF>)

Que no es más que una forma escrita de mostrar el árbol.

Hay varias opciones que puedo agregar al comando para que me meustre diferente información sobre el análisis sintáctico.

Algo importante a destacar es que el ejemplo que mostré tiene el problema de tener recursión a la izquierda en expr: expr (‘\*’|’/’) expr | expr: expr (‘+’|’-’) expr, ya que la regla gramatical se llama a si misma en su primer término, sin embargo, no hubo problemas con antlr4, lo que significa que este es capaz de funcionar con algunas ambigüedades y recursividad izquierda, por lo que puedo darle una gramática relativamente “sucia” y puede que funcione sin problemas.

Además, antlr4 me permite generar un parser en el lenguaje que yo quiera, que en mi caso es Python. Para hacer esto debo ejecutar el comando:

Para esto use la documentación de la siguiente página: <https://yetanotherprogrammingblog.medium.com/antlr-with-python-974c756bdb1b>

* antlr4 -Dlanguage=Python3 Grammar.g4

Que lo que hace es crear varios archivos de Python y texto como un proyecto el cual es mi parser diseñado con la gramática contenida en el archivo Grammar.g4

Con esto ya tengo un entorno donde puedo desarrollar mi parser en un archivo main.py (para eso segui la guía que se encuentra en la página mencionada anteriormente).

**Scanner desarrollado**

Se diseño un scanner usando antlr4 en Python basado en la gramática creada anteriormente en este documento, todo el código esta como una carpeta .zip adjunta a este documento, recuerda que para correrlo es recomendable tener instalado todo lo necesario (mencionado en la sección anterior). Si no se tiene el .zip, en una carpeta nueva y crear un archivo Grammar\_duck.g4, y ejecutar los comandos para crear el proyecto en Python, después copiar el código explicado más adelante en los archivos necesario.

**Diseñar un Test-Plan y comprobar que funciona adecuadamente.**

**Genera, como parte de la entrega, una documentación que describa brevemente los principales hallazgos del análisis de las diferentes herramientas. Agrega, a la documentación previa, cómo (en qué formato) diste de alta las reglas de construcción de LittleDuck que contenga la definición de las expresiones regulares y reglas gramaticales desarrolladas. Considera que este documento irá creciendo conforme trabajes en las siguientes entregas.**

Fuentes:

<https://gist.github.com/dimitrio-m/cf6cde052787ed97164fe6422a5e4cb0>

<https://github.com/antlr/antlr4/blob/master/doc/getting-started.md>

<https://yetanotherprogrammingblog.medium.com/antlr-with-python-974c756bdb1b>