Módulo de Código Intermedio (IR) y Estructuras de Pila

En esta etapa del proyecto se establecieron las estructuras de datos clásicas de compiladores para gestionar las expresiones y el control de flujo, utilizando una Fila de Cuádruplos como Representación Intermedia (IR). Estas estructuras siguen el enfoque estándar de textos de compiladores como Aho, Sethi y Ullman (el "Dragon Book").

Pila	Propósito	Implementación en Patito	Referencia Conceptual
Pila de Operandos (PilaO)	Almacena los lugares (ID, constantes, temporales) que participan en las expresiones.	El rol se cumple con los valores (place, tipo) que regresan las funciones de gramática.	Pila LIFO
Pila de Tipos (PTypes)	Acompaña a los operandos con la información de tipo para la verificación semántica.	El tipo se devuelve junto con el place y se usa para la consulta en el Cubo Semántico.	Pila LIFO
Pila de Operadores (POpper)	Maneja la precedencia de operadores (+, -, *, / y relacionales)[9].	La generación de cuádruplos se hace directamente en las reglas exp/termino del parser, aprovechando la precedencia de PLY.	Pila LIFO

Fila de Cuádruplos (quads)

La lista de cuádruplos es la Representación Intermedia (IR) principal , donde cada elemento es una tupla: (operador, operando_izq, operando_der, resultado).

Tipo de Cuádruplo	Ejemplo de Generación	Propósito
Aritmético	(*, 4, 2, t1); (+, 3, t1, t2)	Ejecuta operaciones y almacena el resultado en un temporal.
Asignación	(=, t2, None, x)	Asigna el valor del operando izquierdo a la dirección de destino.
Relacional	(<, x, y, t3)	Compara operandos y almacena el resultado booleano en un temporal.
Flujo de Control	(GOTOF, t3, None, [Dirección])	Salto condicional (se usa para if y while).
Llamadas	(ERA, total_size, None, func_name)	Preparación del espacio de memoria para una función.

Análisis Semántico

El análisis semántico se enfoca en verificar la consistencia lógica y la validez de tipos en tiempo de compilación.

2.1. Cubo Semántico

- Propósito: Define las reglas de compatibilidad de tipos para cada operación (operador, tipo_izquierdo, tipo_derecho → tipo_resultante). Esto es esencial para la verificación estática de tipos.
- Estructura: Implementado como una matriz lógica de diccionarios anidados en Python, lo que facilita el acceso directo y mejora la legibilidad y mantenibilidad.
- Regla de Promoción: La regla de asignación contempla la promoción válida: un entero puede asignarse a un flotante (flotante = entero), pero no viceversa (para evitar pérdida de precisión).

Tablas de Símbolos

Las estructuras de símbolos gestionan la información de las entidades del programa (variables y funciones) por ámbito.

Estructura	Contenido	Funcionalidad Clave
VariableTable	Almacena VariableInfo (nombre, tipo, dirección virtual).	Verifica que las variables estén declaradas antes de su uso y evita la doble declaración en el mismo ámbito.
FunctionDirectory	Almacena FunctionInfo (nombre, tipo de retorno, parámetros formales, cuádruplo de inicio).	Detecta funciones duplicadas y valida la consistencia de tipos y el número de argumentos en las llamadas.
Herencia Semántica	No es una estructura, sino un patrón de búsqueda.	Permite consultar primero la tabla de variables local de la función y luego la tabla global.

Manejo de la Infraestructura y Ejecución

3.1. Manejo de Memoria Virtual

La clase MemoryManager gestiona un espacio de direcciones virtuales dividido en cuatro segmentos lógicos, cada uno con rangos de direcciones base para cada tipo (entero, flotante, bool, letrero).

Segmento	Rangos de Dirección (Ejemplo para Entero)	Propósito
Global (SEG_GLOBAL)	10000 a 10999	Variables accesibles desde cualquier parte del programa.
Local (SEG_LOCAL)	20000 a 20999	Variables y parámetros de funciones (creados/destruidos por llamada).

Temporal (SEG_TEMP)	30000 a 30999	Resultados intermedios de expresiones, gestionados por una free-list.
Constantes (SEG_CONST)	40000 a 40999	Valores literales (5, "texto") almacenados una sola vez.

Máquina Virtual Patito

La VirtualMachine es el back-end que interpreta y ejecuta la lista de cuádruplos generados por el compilador.

- Pila de Marcos (frames): Una lista de diccionarios que simula la pila de ejecución, donde cada diccionario es un Activation Record (marco) para variables locales y temporales.
- Mecanismo de Llamadas (GOSUB):
    1. ERA: Prepara el nuevo frame de memoria.
    2. PARAMETER: Acumula los argumentos de la llamada.
    3. GOSUB: Guarda el return_ip en call_stack, crea un nuevo marco en frames y salta al start_quad de la función.
    4. ENDFUNC: Restaura el ip y elimina el marco actual de la pila, liberando la memoria local y temporal.

La implementación de Patito utiliza la librería PLY, definiendo tokens con expresiones regulares y la gramática con funciones y docstrings.

Componente	Palabras Clave y Símbolos
Palabras Reservadas	programa, vars, entero, flotante, inicio, fin, escribe, si, sino, mientras, haz, nula.
Operadores	+, -, *, /, =, ==, !=, >, <.
Literales/Tokens	ID (identificadores), CTE_ENT (enteros), CTE_FLOT (flotantes), LETRERO (cadenas de texto).

Caso de Éxito de la Gramática	Caso de Error Detectado
PROGRAM_FLOW (Validar si/sino, mientras)	BAD_MISSING_SEMI (Falta ; en asignación)
PROGRAM_VARS (Declaración y aritméticas)	BAD_UNBALANCED_BRACE (Llave de cierre faltante)
PROGRAM_CALLS_EDGE (Llamadas a funciones con argumentos)	BAD_WHILE_SYNTAX (Falta palabra clave haz)

REFERENCIAS

https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.htm

https://www.geeksforgeeks.org/compiler-design/type-checking-in-compiler-design/

https://www.geeksforgeeks.org/operating-systems/virtual-machines-in-operating-system/

Andrea Galindo Yáñez
A01368483
Patito - entrega #3 (Código de Expresiones y Estatutos lineales)

Estructuras de Pilas y Fila de Cuádruplos

En esta etapa del proyecto se utilizaron estructuras de datos clásicas de compiladores: pilas para manejar expresiones y control de flujo, y una fila (lista) para almacenar los cuádruplos generados como representación intermedia del programa. Estas estructuras siguen el enfoque estándar descrito en textos de compiladores como Aho, Sethi y Ullman (el “Dragon Book”) y notas modernas de cursos de compiladores.

1.1 Pila de operandos (PilaO)

- Propósito: almacenar los valores o lugares (identificadores, constantes y temporales) que participan en las expresiones.

- Tipo de estructura: pila LIFO (Last-In, First-Out).

- Uso en el proyecto:

    - Cada vez que se reconoce un factor (un ID, una constante o una expresión entre paréntesis), se obtiene su “place” (puede ser un nombre de variable, una constante o un temporal) y su tipo.

    - Ese lugar se utiliza inmediatamente cuando se combinan con operadores para formar cuádruplos, por lo que, aunque no estamos usando una PilaO explícita en todas las reglas del parser, el patrón que seguimos es equivalente al algoritmo clásico de “push operandos / pop operandos” para generar código intermedio.

  Intuitivamente, PilaO modela el “stack de expresión” que se usa para evaluar expresiones infijas respetando la precedencia y asociatividad de los operadores, como se explica en notas de compiladores y estructuras de datos.

1.2 Pila de tipos (PTypes)

- Propósito: acompañar a la pila de operandos con la información de tipo de dato de cada operando.

- Tipo de estructura: pila LIFO.

- Uso en el proyecto:

- Cada vez que se introduce un operando (entero, flotante, etc.) se podría registrar su tipo en PTypes.

- Cuando se va a combinar dos operandos con un operador (+, *, <, ==, etc.), se consultan sus tipos y el operador en el cubo semántico para verificar si la operación es válida y obtener el tipo resultante.

- En el código actual, este rol lo estamos cumpliendo directamente a través de valores (place, tipo) que regresan las funciones de gramática, pero conceptualmente es el mismo uso que una pila de tipos.

Esto sigue la técnica de verificación de tipos durante la generación de código intermedio, donde por cada operación se decide si es permitida y qué tipo tendrá el resultado.

1.3 Pila de operadores (POper)

Nota: en tu implementación actual estamos haciendo la generación de cuádruplos directamente en las reglas exp, termino y expresion, pero el diseño de POper está preparado para la versión “con pilas” como en las diapositivas.

- Propósito: manejar la precedencia de operadores (+, -, *, /, relacionales) de forma ordenada.

- Tipo de estructura: pila LIFO.

- Idea clásica:

  - Cuando se lee un operador, se hace push en POper.

  - Cuando la gramática detecta que se puede “reducir” (por ejemplo, se terminó un TERM y en la cima hay * o /), se hace pop del operador, pop de los dos operandos en PilaO, se verifica tipos y se genera el cuádruplo correspondiente.

- En tus diapositivas de “Adding semantic actions to create Quadruples”, los pasos numerados (1,2,3,4,5) describen justamente estas acciones sobre PilaO, PTypes y POper.

Andrea Galindo Yáñez
A01368483
Patito - entrega #3 (Código de Expresiones y Estatutos lineales)
Aunque el parser que tenemos ya incorpora la precedencia por medio de las reglas de PLY y no utiliza POper en todas partes, el diseño que estamos documentando se alinea con el enfoque estándar de compiladores para construir cuádruplos usando pilas de operandos y operadores.

1.4 Pila de saltos (PJumps)

- Propósito: soportar el control de flujo en estatutos no lineales como if, if-else y while (fase que viene después).
- Tipo de estructura: pila LIFO.
- Uso típico:
    - Cuando se genera un cuádruplo GOTOF o GOTO cuyo destino aún no se conoce, se guarda su índice en PJumps.
    - Más adelante, cuando se conoce la posición de destino, se “rellena” el campo faltante con fill_quad.
- Este patrón está documentado en muchos apuntes de construcción de compiladores cuando se explica backpatching y manejo de saltos.

1.5 Fila de cuádruplos (quads)

- Propósito: servir como representación intermedia (Intermediate Representation) del programa Patito.
- Tipo de estructura: lista/cola ordenada, donde cada elemento es un cuádruplo:
    - (operador, operando_izq, operando_der, resultado)
- Ejemplos de cuádruplos generados en tu proyecto:
    - Operaciones aritméticas:
        - (*, 4, 2, t1)
        - (+, 3, t1, t2)
    - Asignaciones:

        - (=, t2, None, x)

    - Operaciones relacionales:

        - (<, x, y, t3)

    - Salidas:

        - (PRINT, "x =", None, None)

        - (PRINT, x, None, None)

- Motivación teórica:
  Representaciones tipo cuádruplos son uno de los esquemas clásicos de código intermedio en compiladores, junto con triples y código de tres direcciones, por su sencillez para aplicar optimizaciones y traducir luego a código máquina o bytecode.

En nuestro caso, esta Fila de cuádruplos es el producto principal de la tercera etapa del proyecto.

---

2. Diagramas del lenguaje Patito con puntos neurálgicos

En clase te mostraron diagramas de flujo de la gramática (por ejemplo <EXP>, <TERM>, <FACTOR>, <IF-stmt>, <WHILE-stmt>), con números (1,2,3,4,5) marcando “puntos neurálgicos” donde se ejecutan acciones semánticas sobre las pilas.

Aquí te dejo una descripción textual que puedes traducir a tus diagramas:

2.1 Diagrama de Expresiones Aritméticas (<EXP>, <TERM>, <FACTOR>)

Estructura general:

- <EXP>: combina TERMINO con operadores + y -.

- <TERM>: combina FACTOR con operadores * y /.

- <FACTOR>: produce operandos básicos (identificadores, constantes, paréntesis, signos unarios).

Andrea Galindo Yáñez
A01368483
Patito - entrega #3 (Código de Expresiones y Estatutos lineales)
Puntos neurálgicos típicos (según las diapositivas):

1. FACTOR (id / cte) – Paso 1
    - Acción: “PilaO.Push(id.name) y PTypes.Push(id.type)”
    - En tu implementación: cuando factor_cte reconoce un ID o cte, devuelve (place, tipo).
    - Rol: introducir el operando y su tipo al flujo de evaluación.
2. Operadores * y / en <TERM> – Paso 2
    - Acción: POper.Push(* or /)
    - En la versión clásica con pilas, al ver un * o / se mete en la pila de operadores y, cuando se cierra el término, se revisa si se debe generar un cuádruplo.
    - En tu parser, esto se refleja en termino y termino_factor: tomas el operador, combinas los operandos y generas un cuádruplo (op, left_place, right_place, temp).
3. Operadores + y - en <EXP> – Paso 3
    - Acción: POper.Push(+ or -)
    - De forma similar a *//, pero con menor precedencia.
    - En tu código se implementa en exp y exp_termino, utilizando las reglas de precedencia de PLY para respetar el orden correcto antes de generar el cuádruplo.
4. Reducción en <TERM> / <EXP> – Paso 4 y 5
    - Acción típica (según las diapositivas):
        - Hacer Pop de dos operandos de PilaO y sus tipos de PTypes.
        - Hacer Pop de un operador de POper.
        - Consultar result_type = SemanticCube[left_type][op][right_type].

        - Si no hay error, generar cuádruplo y hacer Push del temporal y su tipo.

    - En tu implementación esto se resume así:

        - Las reglas termino/exp reciben (place, tipo) de los hijos.
        - Se llama a result_type(...) del cubo semántico.
        - Se genera emit_quad(op, left_place, right_place, temp).
        - Se devuelve (temp, tipo_resultante).

En tus diagramas puedes marcar:

- En la salida de id / cte: “1) Generar operando y tipo”.
- En las burbujas * / /: “2) Guardar operador / combinar factores”.
- En las burbujas + / -: “3–4) Combinar términos, generar cuádruplos si procede”.

2.2 Diagrama de Expresiones Relacionales (<EXPRESION>)

Estructura:

expresion → exp expresion_exp

expresion_exp → ε

| '>' exp

| '<' exp

| '==' exp

| '!=' exp

Puntos neurálgicos:

Andrea Galindo Yáñez
A01368483
Patito - entrega #3 (Código de Expresiones y Estatutos lineales)

1. Fin de exp (lado izquierdo):
    - Ya tenemos (left_place, left_type) provenientes de la expresión aritmética.
2. Lectura del operador relacional:
    - Se decide el operador relacional (<, >, ==, !=).
3. Fin de exp (lado derecho):
    - Se obtiene (right_place, right_type).
4. Acción semántica:
    - Se consulta el cubo semántico: result_type(left_type, op, right_type), que debería regresar bool si es una comparación válida.
    - Se genera el cuádruplo (op, left_place, right_place, temp_bool).
    - Se devuelve (temp_bool, bool) como resultado de la expresión relacional.

En tu diagrama de <EXPRESION> puedes marcar un nodo con texto como:

> “4) Verificar tipos (relacional) y generar cuádruplo condicional (resultado bool)”.

2.3 Diagrama del estatuto de asignación

Regla:

asigna → ID '=' expresion ';'

Puntos neurálgicos:

1. Lectura de ID:
    - Se consulta su tipo en la tabla de variables (global por ahora).
2. Evaluación de expresion:

- Produce (expr_place, expr_type) y genera todos los cuádruplos necesarios para calcularla.

3. Acción semántica de asignación:

    - Se consulta el cubo semántico con (var_type, '=', expr_type).
    - Si es válido, se genera el cuádruplo:
        - (=, expr_place, None, var_name)
    - Si no, se lanza un SemanticError de tipo mismatch.

En el diagrama de asignación puedes marcar:

- Flecha entre ID y =: “1) Buscar tipo de la variable”.
- Flecha saliendo de expresion: “2) Obtener (place, tipo) de la expresión”.
- Nodo antes de ';': “3) Verificar tipos y generar cuádruplo de asignación”.

2.4 Diagrama del estatuto escribe

Regla:

imprime → ESCRIBE '(' imprime_exp ')' ';'

imprime_exp → expresion imprime_exp_p

| LETRERO imprime_exp_p

imprime_exp_p → ε

| ',' imprime_exp

Puntos neurálgicos:

1. Para cada LETRERO:

- Se genera un cuádruplo:

        - (PRINT, "texto literal", None, None)

2. Para cada expresion:

    - Primero se generan los cuádruplos de la expresión aritmética/relacional.

    - Luego se genera:

        - (PRINT, expr_place, None, None)

En el diagrama puedes marcar:

- Nodo a la salida de LETRERO: “1) Generar cuádruplo PRINT de cadena”.
- Nodo a la salida de expresion: “2) Generar cuádruplo PRINT del valor”.

Referencias

https://www.geeksforgeeks.org/compiler-design/three-address-code-compiler/

https://www.m-zakeri.ir/Compilers/lectures/08_Intermediate-Code-Generation/

https://www.ida.liu.se/~TDDB44/laboratories/instructions/lab6.html

https://www.sciencedirect.com/topics/computer-science/three-address-code

Andrea Galindo Yáñez
A01368483
Entrega 2 - Patito
Durante el proceso de construcción del compilador de mi lenguaje. En esta etapa del proyecto se implementaron las estructuras fundamentales que permiten llevar a cabo el análisis semántico, el cual valida que las construcciones sintácticamente correctas también tengan significado lógico.

El análisis semántico se encarga de:
-Las variables están declaradas antes de su uso.
-Las operaciones entre datos sean compatibles en tipo.
-Las funciones tengan consistencia entre su definición y su llamado.
-No existen redefiniciones de variables o funciones.

Se diseñaron 3 estructuras, cubo semántico, directorio de funciones y tablas de variables.

Cubo Semántico
Es una tabla tridimensional que define las reglas de compatibilidad de tipos y resultados esperados de operaciones. Cada entrada del cubo se accede, por su operador, tipo izquierdo y tipo derecho.
Si una combinación es válida, el cubo devuelve el tipo resultante. Se utiliza en tiempo de compilación, en la etapa de verificación de expresiones antes de generar código intermedio.
-Los diccionarios anidados ofrecen un acceso directo, facilitando consultas durante el parseo.
-Su estructura es expandible, permitiendo agregar operadores lógicos relacionales o de asignación múltiple sin rediseñar la base.
-Evita la rigidez de un arreglo tridimensional, mejorando la legibilidad y mantenibilidad.

Directorio de Funciones
Actúa como el núcleo semántico que organiza todas las funciones declaradas en el programa. Se almacena el nombre, tipo de retorno, parámetros formales y tipo, además de las variables locales asociadas.
Se administra un diccionario global, que permite acceso directo a funciones, se encapsula la información de cada función y se permite detectar funciones duplicadas, validar consistencia de tipos y argumentos en llamadas.

Tablas de Variables
Cada ámbito del programa posee su propia tabla de variables, donde se registran todas las variables declaradas dentro del contexto. Cada variable almacena el nombre, tipo y el parámetro (o variable local).
El uso del diccionario permite la búsqueda inmediata, la estructura admite herencia semántica, que es que cada función se pueda consultar primero su tabla local y luego global. Se simplifica la verificación de redefinición de variables y detección de uso antes de la declaración.
Se valida que la variable
-Doblemente declarada -> error
-Usada sin declararse -> error
-Parámetro duplicado en la misma función -> error

Integración de las Estructuras en el Parser

Andrea Galindo Yáñez
A01368483
Entrega 2 - Patito
Se añadieron puntos neurálgicos semánticos que permiten la sincronización entre el parser y las tablas semánticas. Esto para que se realice las validaciones en tiempo de parseo, no de ejecución.

La implementación del cubo semántico, directorio de funciones y tablas de variables se establece la infraestructura central del análisis semántico en el compilador.

Referencias
Liga del github :
https://github.com/AndyGalindo0411/Desarrollo-de-aplicaciones-avanzadas-de-ciencias-computacionales

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools (2nd ed.)*. Pearson Education. Recuperado de https://www.pearson.com/en-us/subject-catalog/p/compilers-principles-techniques-and-tools/P200000003724

Stanford University. (2023). *CS143: Compilers — Lecture Notes on Semantic Analysis and Type Checking*. Recuperado de https://web.stanford.edu/class/cs143/

Presentaciones de Elda Guadalupe Quiroga González por el curso TC3002B.503. Semantics, Symbol Tables, Expressions and IR, Exp Quad.

Andrea Galindo Yáñez
A01368483
Entrega 1

Investigar algunas herramientas de generación automática de compiladores (considera la tarea previa que entregaste).

Lark

Descripción general

Lark es una librería de parsing para Python moderna que permite definir una gramática y un conjunto de tokens para luego parsear textos y construir árboles de análisis (parse trees).

En su página de PyPI se describe como: “parse any context-free grammar, efficiently, with very little code”.

En el repositorio de GitHub se afirma que “Lark can parse all context-free languages. … Implements both Earley (SPPF) and LALR(1)… you can trade-off power and speed”.

Características técnicas destacadas

Soporta múltiples algoritmos: Earley (capaz de gramáticas amplias/ambiguas), LALR(1) (rápido para gramáticas no ambiguas).

Permite definir tokens mediante expresiones regulares y luego gramáticas en formato estilo EBNF (o similar). Por ejemplo en los tutoriales de Lark se muestra cómo definir la gramática para JSON.

Soporte para ignorar espacios/whitespace (%ignore u opción similar), seguimiento automático de línea/columna, unicode, importación de gramáticas, etc.

Ventajas

Muy buena adaptación a Python puro, sin generar código externo: simplemente importas, defines gramática, parseas.

Flexibilidad: si tienes una gramática no trivial u ambigua, el algoritmo Earley permite manejarla.

Compatible con los tokens definidos mediante expresiones regulares, lo cual se alinea bien con la parte de “scanner” + “parser” que tú tienes.

Desventajas

Aunque LALR(1) es rápido, si usas algoritmos más generales (Earley) puede haber penalización en velocidad/memoria para gramáticas muy grandes.

Puede que la sintaxis de definición de la gramática (al estilo Lark) requiera adaptaciones si vienes de un estilo “lex+yacc” clásico.

Andrea Galindo Yáñez
A01368483
Entrega 1

PLY (Python Lex-Yacc)

Descripción general

PLY es una implementación 100 % en Python de las herramientas clásicas Lex y Yacc, que permiten construir un analizador léxico (scanner) y un analizador sintáctico (parser) mediante la técnica LALR(1).

Características técnicas destacadas

Separación clara entre el scanner (ply.lex) y el parser (ply.yacc): defines las reglas lexicas (t_TOKENNAME = r'regex') y luego la gramática en funciones docstring (por ejemplo def p_expression(p): 'expression : expression PLUS term').

Soporte para producciones vacías, reglas de precedencia, recuperación de errores, gramáticas ambiguas (aunque con limitaciones).

Muy adecuada para enseñanza y para seguir el modelo clásico de “scanner + parser + acciones semánticas”.

Ventajas

Muy didáctico: ideal para clases de compiladores donde se quiere exponer la mecánica de scanner + parser.

Integración sencilla con Python, con control total sobre las reglas de tokenización y de análisis sintáctico.

Desventajas

Menos “automático” que Lark: tienes que definir más manualmente cada parte (scanner, parser, manejo de errores).

Basado en LALR(1) — si tu gramática no es LALR(1) (por ejemplo con muchas ambigüedades o construcciones complejas), habrá más ajustes.

Andrea Galindo Yáñez
A01368483
Entrega 1

ANTLR 4

Descripción general

ANTLR (Another Tool for Language Recognition) es un generador de analizadores sintácticos ampliamente usado en industria y academia. La versión 4 (ANTLR 4) es moderna, soporta múltiples lenguajes objetivo (Java, Python, C#, JavaScript, etc.).

Características técnicas destacadas

Gramática se escribe típicamente en un archivo .g4 que contiene reglas para lexer (tokens) y parser (producciones).

Algoritmo LL(*), lo que le da flexibilidad en el análisis sintáctico top-down (diferente de LALR). Puede manejar gramáticas más complejas o ambiguas mediante ciertas estrategias.

Generación de código para múltiples lenguajes, estructura de árboles de análisis, visitantes/listeners para recorrer ASTs, integración con IDEs.

Buena capacidad de debug, visualización de árboles, soporte para manejar errores, modos léxicos avanzados.

Ventajas

Potente, robusta, con ecosistema grande: ejemplo de uso comercial, muchas gramáticas pre-hechas, plugins de IDE.

Multilenguaje objetivo: puedes generar parsers en Python, Java, C#, etc., lo que es útil si tu lenguaje o herramienta puede cambiar de entorno.

Muy estructurada: la separación de lexer/parser, soporte para árbol de análisis, visitas, puede facilitar fases posteriores (semántica, generación de código).

Desventajas

Requiere un paso de generación: escribes .g4, ejecutas ANTLR, generas código, luego lo usas. Esto añade complejidad de “tooling” comparado con librerías que corren directamente en Python.

Para proyectos pequeños o iterativos (“vamos rápido”), puede sentirse “pesada”.

Andrea Galindo Yáñez
A01368483
Entrega 1

Formato de Definición de Reglas de Construcción del Lenguaje

Las reglas de construcción del lenguaje Patito fueron implementadas en formato Python, utilizando la librería PLY (Python Lex-Yacc), la cual permite definir tanto el scanner (análisis léxico) como el parser (análisis sintáctico) con base en expresiones regulares y reglas gramaticales tipo BNF.

Definición de las Expresiones Regulares (Scanner)

Las reglas léxicas se declararon como funciones o variables t_ dentro del archivo scanner.py.
Cada token se define mediante una expresión regular, que identifica los elementos básicos del lenguaje

```
# Palabras reservadas
reserved = {
   'programa': 'PROGRAMA',
   'vars': 'VARS',
   'entero': 'ENTERO',
   'flotante': 'FLOTANTE',
   'inicio': 'INICIO',
   'fin': 'FIN',
   'escribe': 'ESCRIBE',
   'si': 'SI',
   'sino': 'SINO',
   'mientras': 'MIENTRAS',
   'haz': 'HAZ',
   'nula': 'NULA'
}

# Operadores y delimitadores
t_OP_SUMA    = r'\+'
t_OP_RESTA   = r'-'
t_OP_MULT    = r'\*'
t_OP_DIV     = r'/'
t_OP_ASIG    = r'='
t_OP_IGUAL   = r'=='
t_OP_DIF     = r'!='
t_OP_MAYOR   = r'>'
t_OP_MENOR   = r'<'
t_PAR_ABRE   = r'\('
t_PAR_CIERRA = r'\)'
t_LLAVE_ABRE = r'\{'
t_LLAVE_CIERRA = r'\}'
t_PUNTO_Y_COMA = r';'
```

Andrea Galindo Yáñez
A01368483
Entrega 1

```
# Identificadores y constantes
t_ID       = r'[a-zA-Z_][a-zA-Z0-9_]*'
t_CTE_ENT  = r'\d+'
t_CTE_FLOT = r'\d+\.\d+'
t_LETRERO  = r'\"([^\\\n]|(\\.))*?\"'
```

Definición de las Reglas Gramaticales (Parser)

Las reglas gramaticales se definieron en el archivo parser.py usando funciones p_<nombre>() con docstrings en formato BNF, especificando las producciones del lenguaje

```
def p_programa(p):
    'programa : PROGRAMA ID PUNTO_Y_COMA pro_vars pro_funcs INICIO cuerpo FIN'
    p[0] = ('programa', p[2], p[4], p[5], p[7])

def p_vars(p):
    'vars : VARS ID vars_todo'
    p[0] = ('vars', p[2], p[3])

def p_expresion(p):
    'expresion : exp expresion_exp'
    p[0] = ('expr', p[1], p[2])
```

Principales Test Cases de Validación

Se desarrolló un archivo de pruebas automatizadas llamado tests_scanner_parser.py que valida el correcto funcionamiento del scanner y parser.
Cada prueba se ejecuta automáticamente desde la terminal con python tests_scanner_parser.py

Andrea Galindo Yáñez
A01368483
Entrega 1

Casos de éxito

Caso	Descripción	Objetivo
PROGRAM_MIN	Programa mínimo con estructura `programa … inicio … fin	Validar sintaxis base
PROGRAM_VARS	Declaración de variables y operaciones aritméticas	Validar sección `vars` y expresiones
PROGRAM_FUNCS	Funciones con variables locales y llamadas	Validar definiciones de funciones
PROGRAM_FLOW	Estructuras `si`, `sino`, y `mientras`	Validar control de flujo
PROGRAM_REL_PRINT	Expresiones relacionales e impresión	Validar operadores relacionales
PROGRAM_CALLS_EDGE	Llamadas a funciones con distintos argumentos	Validar sintaxis de llamadas
PROGRAM_UNARY_SIGNS	Operadores unarios `+` y `-`	Validar signos sobre ctes o ids

Casos de Error

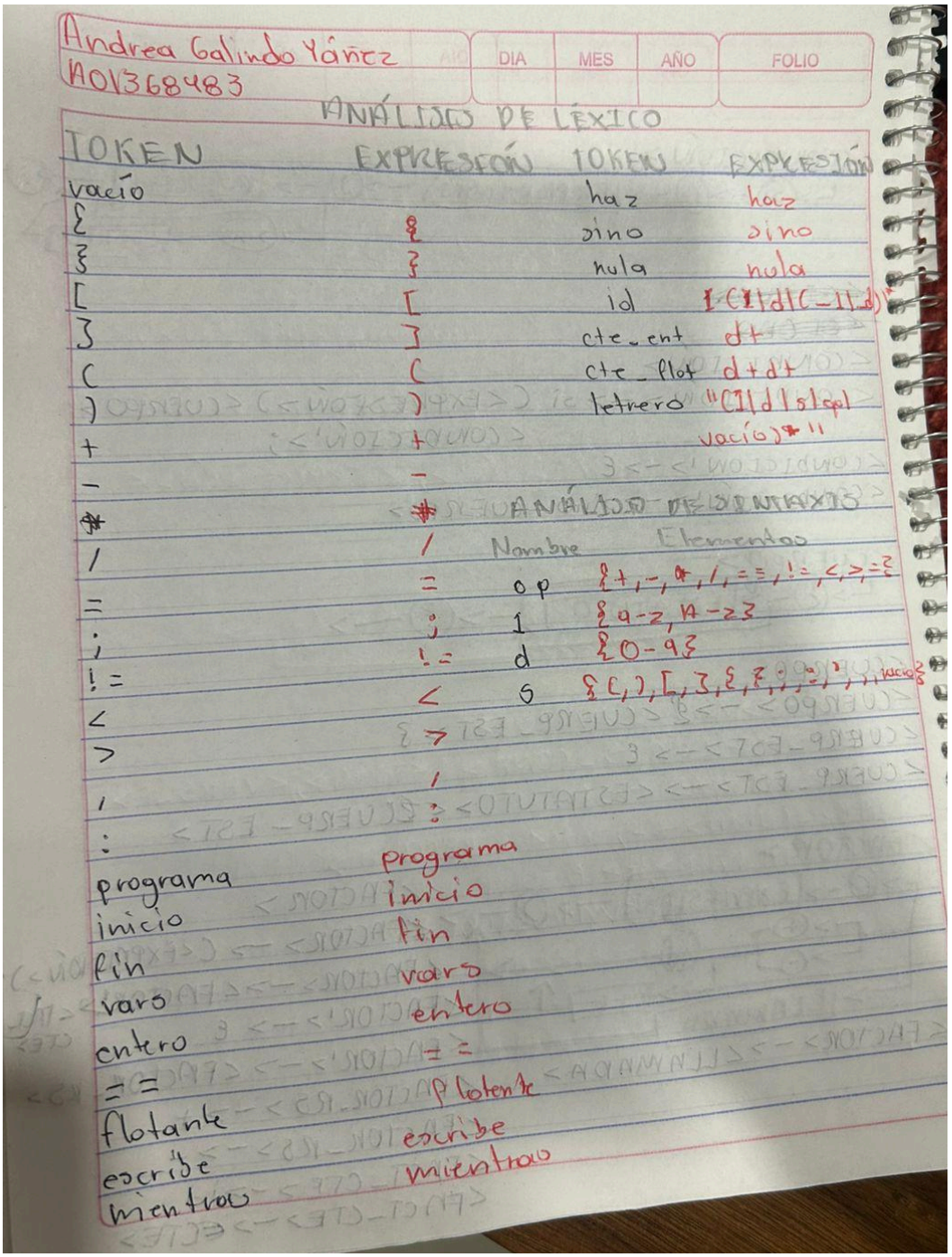
Caso	Descripción	Error Detectado
BAD_MISSING_SEMI	Falta `;` en asignación	Error de sintaxis
BAD_UNBALANCED_BRACE	Llave de cierre faltante	Error de estructura de bloque
BAD_PROGRAM_NO_ID	Falta el identificador del programa	Error léxico
BAD_WHILE_SYNTAX	Falta palabra clave `haz`	Error gramatical
BAD_EXTRA_VARS_BLOCK	Doble bloque `vars` consecutivo	Error de gramática

Andrea Galindo Yáñez
A01368483
Entrega 1

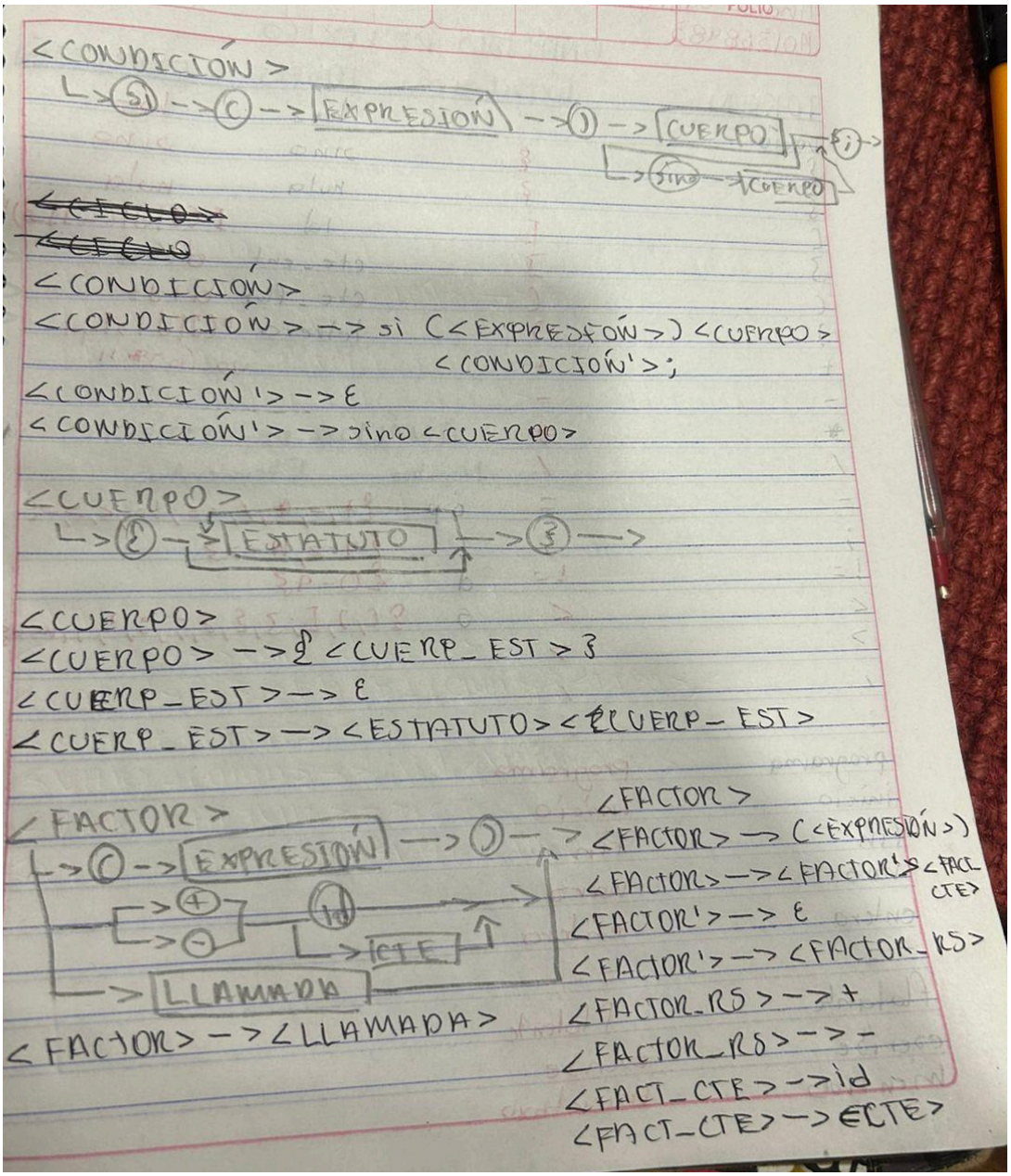
Liga del Github :
https://github.com/AndyGalindo0411/Desarrollo-de-aplicaciones-avanzadas-de-ciencias-computacionales

Referencias
https://manpages.ubuntu.com/manpages/jammy/man7/lark.7.html
https://www.cs.vu.nl/~jansa/ftp/BK0/ply.html
https://tomassetti.me/antlr-mega-tutorial/

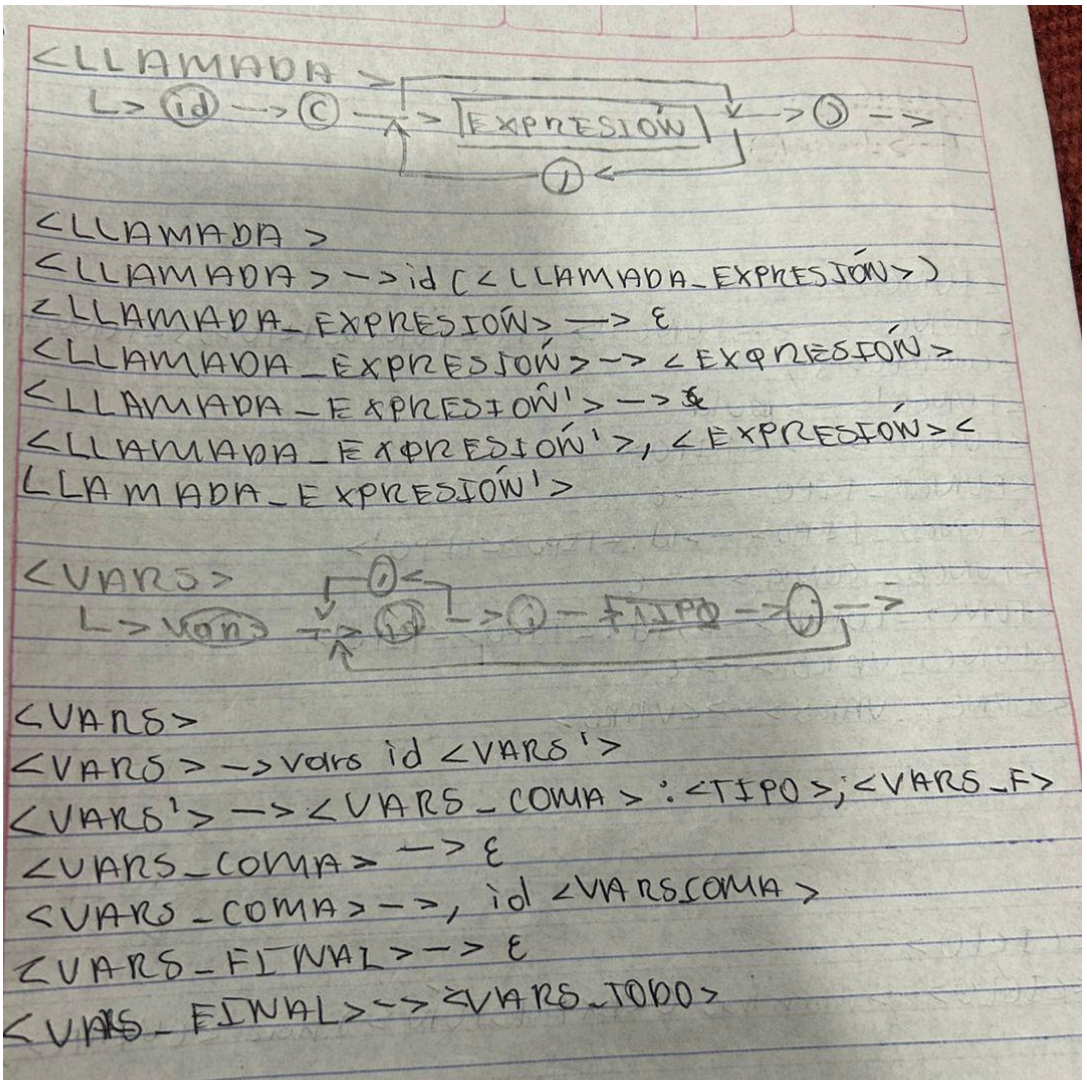
Andrea Galindo Yáñez
A01368483
Patito - entrega #0



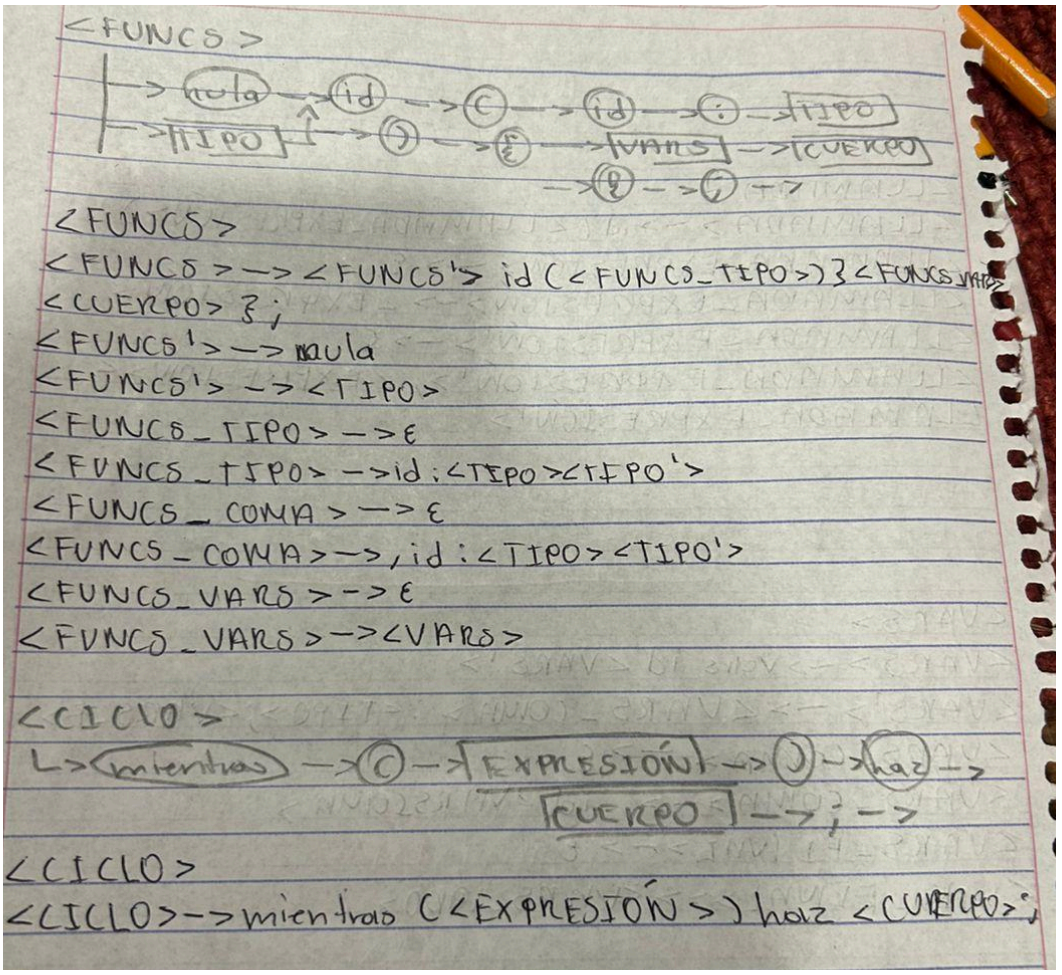
Andrea Galindo Yáñez
A01368483
Patito - entrega #0



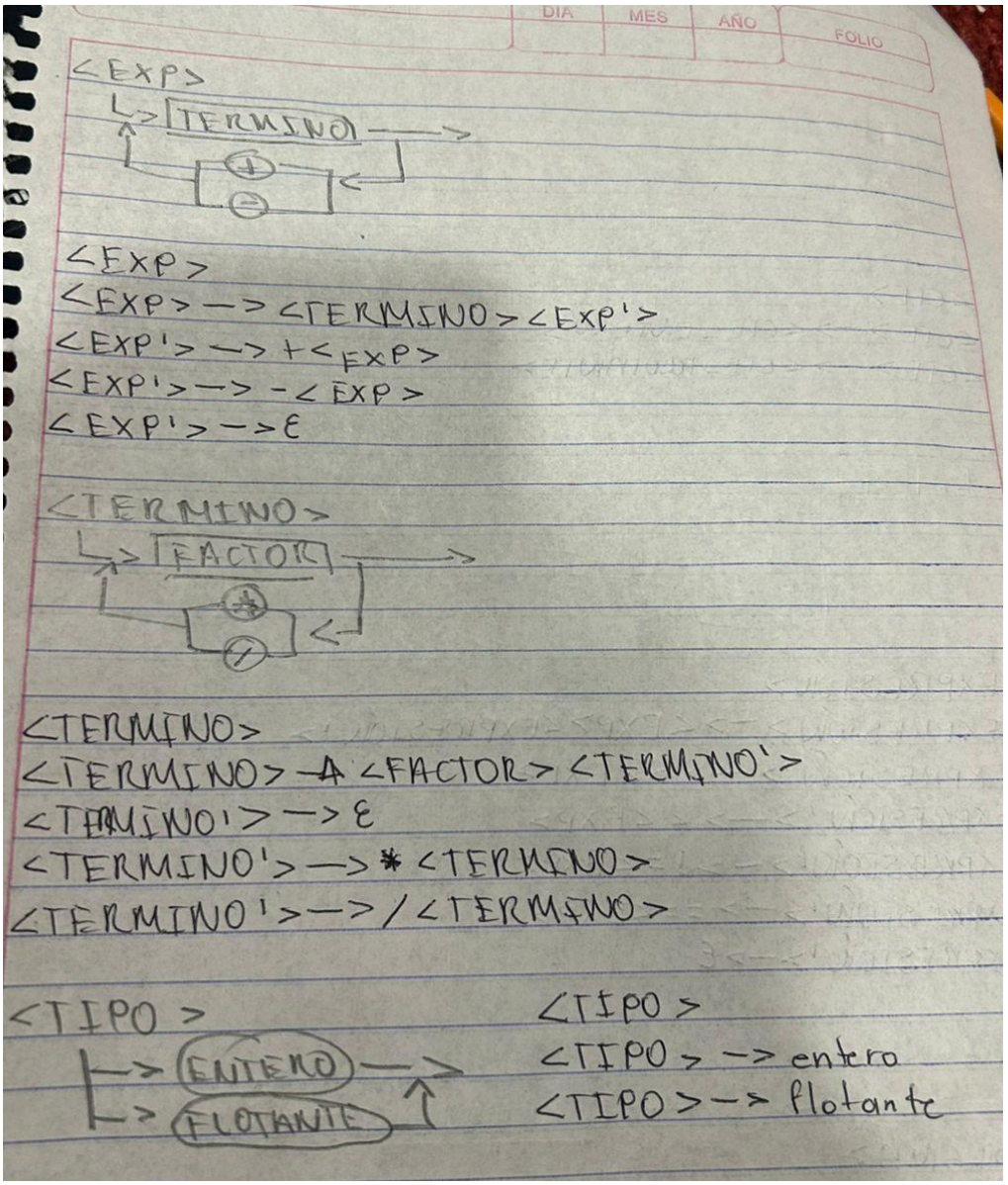
Andrea Galindo Yáñez
A01368483
Patito - entrega #0



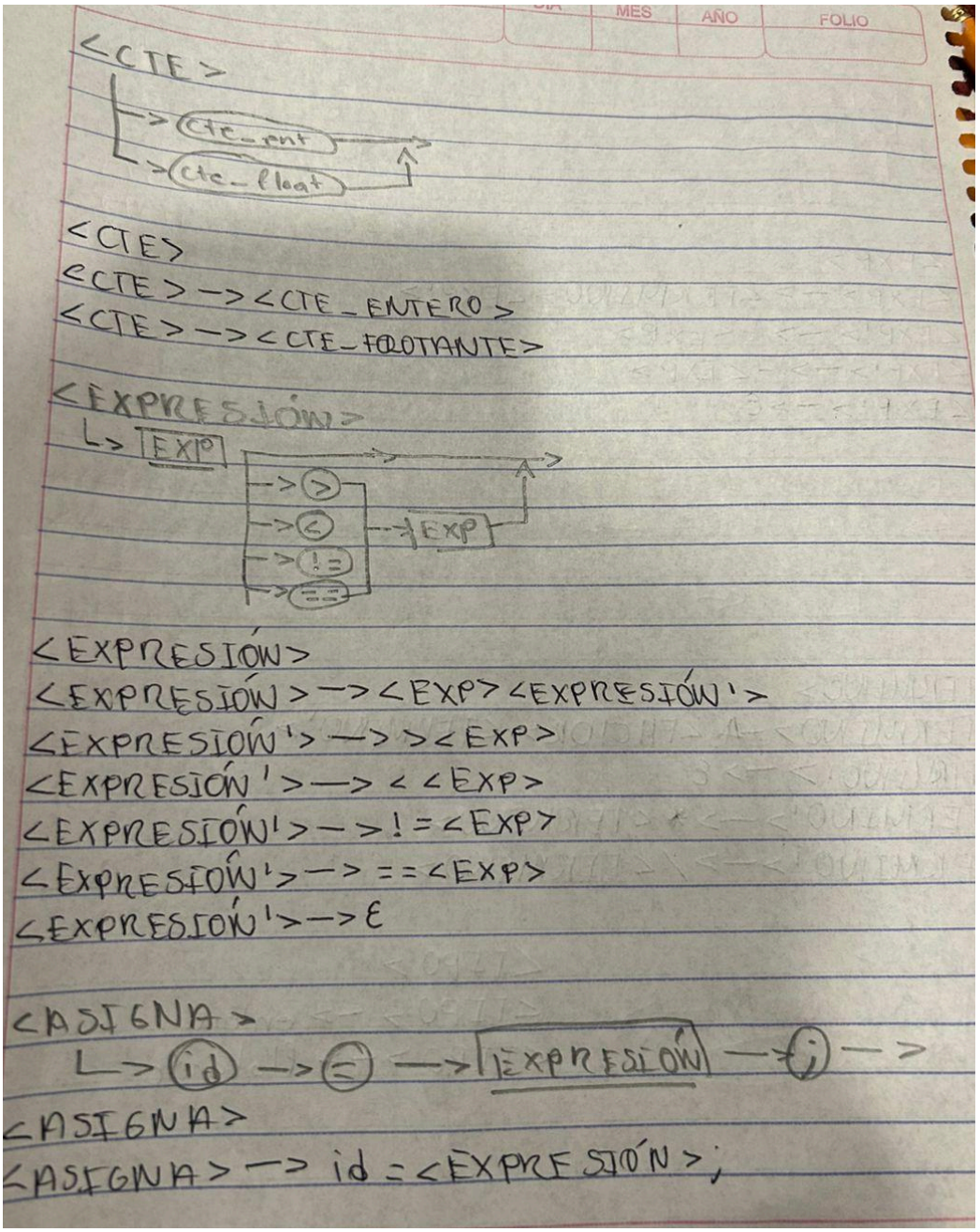
Andrea Galindo Yáñez
A01368483
Patito - entrega #0



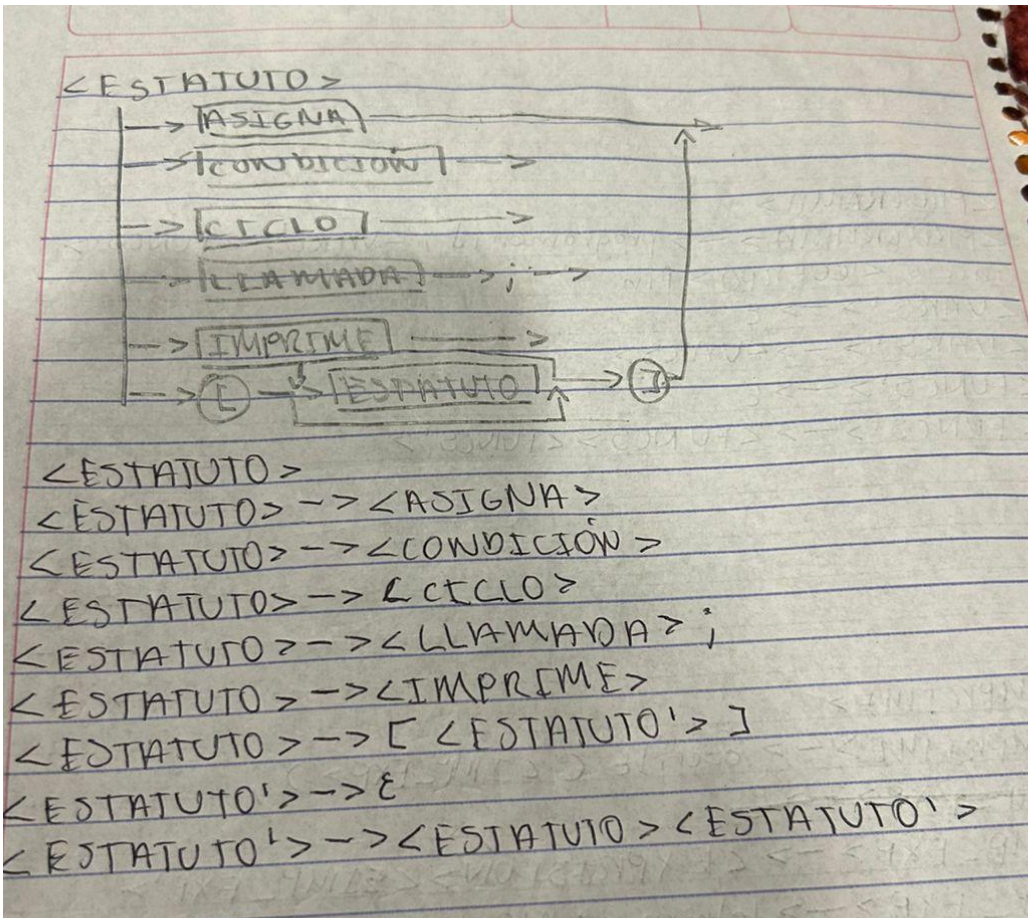
Andrea Galindo Yáñez
A01368483
Patito - entrega #0



Andrea Galindo Yáñez
A01368483
Patito - entrega #0



Andrea Galindo Yáñez
A01368483
Patito - entrega #0



Andrea Galindo Yáñez
A01368483
Patito - entrega #0