COMPILADOR DECAF – DAVID CORZO & IAN JENATZ

FASE 1 – SCANNER Y LEXER:

En este README se encuentra en general cómo funcióna el lexer, sin embargo para detalles triviales hay documentación en cada método del algoritmo que se pueden consultar en cuyo caso se necesite.

El algoritmo consta de 4 clases: - class NFA - class DFA - class scanner - class error_msg

A continuación explicaré cada una.

DOCUMENTACIÓN DE FUNCIONES GLOBALES:

1. **def add_edge**:
    - Esta función chequea el diccionario de estados para poder insertar una llave y valor, primero chequea si la llave que corresponde a épsilon existe en el diccionario, si no existe hace un update agregando la llave y el valor (que es un set), si existe simplemente agrega al set correspondiente. Si la llave no es épsilon simplemente se hace un update al número como valor puesto a que sólo épsilon puede tener varias aristas, todo lo demás tiene aristas únicas.
2. **def num_label_maker**:
    - Generador para poder tener nombres estados únicos dentro del diccionario de estados, pero produce números únicos para nombrar los estados. Usados en el diccionario de estados.
3. **def letter_label_maker**:
    - Generador para poder tener nombres estados únicos dentro del diccionario de estados, lo mismo que la anterior, pero produce combinaciones de letras en lugar de números para nombrar los estados. Usados en estados de DFA.

DOCUMENTACIÓN DE CLASES USADAS:

1. **class enfa**:
    - Esta clase sirve como un 'struct' en el que se almacena el estado inicial y final del NFA. Sólo almacena valores y los métodos sólo sirven para imprimir el 'struct'
2. **class dfa**:
    - Sirve la exactamente la misma función que la clase anterior, pero se utiliza para el DFA.
3. **class NFA**:
    - En esta clase se construye el NFA de los estados y sus aristas. Los estados son guardados en un diccionario de la siguiente forma: `{<nombre_de_estado>:{<arista_1>:<siguiente_estado1>, ... ,<arista_n>:<sigiente_estado_n>}, <nombre_de_siguiente_estado>:{<aristas>}}` Ejemplo: `{1:{'a':2}, 2:{'b':3}, 3:{}}` La única arista que tiene más de un siguiente estado es la de None que es la representación que usamos para denotar épsilon. Todas las demás arístas son únicas. Se utilizó el algorimo de thompson para construir cada operación especificada del regex. Cada operación se hace push a un stack, si este stack no tiene un length de 1 al finalizar la operación la expresión regular es incorrecta (usualmente es por que falta una concatenación).
    - Esta clase toma una expresión regular y devuelve un épsilon-nfa. Contiene los siguientes métodos:
    - **def infix_to_postfix**: Este método usa el atributo de clase 'self.infix', que es una expresión regular en forma normal, la transformamos a una postfix por conveniencia del algoritmo. Luego procede a hacer una expresión regular que está en forma de postfix. Ejemplo: a+b+(c+d) -> cd+a+b+
    - **def regex_or**: Operación or. Si la última operación y la penultima fueron 'c' y '|' estamos hablando de una operación tipo or encadenada, no crearemos más nodos con epsilon, usaremos los que ya están y sólo agregaremos la nueva arista de épsilon al nfa 'c'. De lo contrario creamos dos nuevos estados y unimos el estado inicial a los dos nfas que le dimos pop, a los dos nfas que les dimos pop les agregamos aristas al nuevo nodo final y le damos push al stack. Registramos operación '|' a menos que nos indiquen lo contrario.
    - **def regex_character**: Hacemos la operación de character. Creamos dos estados. Agregamos una arista del nodo inicial al final por medio del character que nos manden. Hacemos push al stack. Registramos la operación como 'c' en el stack.
    - **def regex_concatenation**: Opeación concatenación. Le damos pop dos veces al stack. Si son characteres no unimos con épsilon sino simplemente usamos las mismas aristas de caracteres para poder concatenar. de lo contrario usamos épsilon para unir dos nfas. Registramos la operacion '·' al stack a menos que nos especifiquen no hacerlo.
    - **def regex_question_mark**: Realizamos la operación de ?. que es (epsilon|nfa). Basicamente agregamos una arista de épsilon en el nodo inicial nfa que esta en el top del stack al final node del mismo nfa. Y registramos la operación ? a menos que no especifiquen no hacerlo.
    - **def regex_kleene**: Se encarga de realizar la operación de asterisko o kleene. Hacemos pop del último nfa que esta en el stack. Creamos los estados necesarios para la operación unidos con épsilon respectivamente. Agregamos dicho nfa de nuevo al stack. Registramos como última operación el asterisco a menos que nos especifiquen que no lo hagamos con el argumento push_to_op_stack.
    - **def thompson_construction**: Esto convierte la expresión regular en epsilon nfa, epsilon estará representado por 'None' Convertimos el regex a postfix. Checkeamos los operadores y llamamos a las funciones según la operación especificada. Si está precedida por \ si escapa el siguiente character, si el character siguiente es n o t los registramos como \n y \t, de lo contrario omitimos \ y solo agarramos el siguiente character. Si el char input no es ningun operador entonces asumimos que es un character. Si es épsilon no lo agregamos al input alphabet.
    - **extra funcionalidad**: el método **draw_nfa** importa librerías que grafican el NFA para poder visualizar el grafo. Este método permanece comentado puramente por razónes de rendimiento.
4. class DFA:
    - Toma como argumento en su contructor a la clase NFA anterior. Los estados del NFA a continuación se les calcula el closure de cada uno y se almacena en un diccionario. Después, empezando por el estado inicial construimos la tabla de closure recursivamente, mientras van apareciendo nuevos estados del dfa los agregamos al registro de estados del dfa de la siguiente manera: `{<tupla_de_estados_de_closure_nfa>:{<arista>:<tupla_siguiente_estado_de_dfa>}}` Ejemplo: `{(3,4,5,6):{'a':(3,4,5)}, (3,4,5):{'b':(8)}, (8):{}}` Al terminar por conveniencia cambiamos los nombres de tuplas a un string autogenerado de letras y números.
    - **def calculate_closure_of_all_states**: Este método calcula el closure de todos los estados. Toma los estados de las variables globales e itera en sus claves. Luego, para cada clave de estado, calculamos el closure utilizando el método self.e_closure.
    - **def closure_of_all_states**:
    - **def closures**: Este método asume que el método calculate_closure_of_all_states se ejecutó primero. Este método toma un conjunto de estados, que en nuestro caso significa un conjunto de números. Estos estados son claves del diccionario de variables globales de closings_of_all_states. Este método simplemente agrega el closure de todos los estados que están contenidos en la variable de argumento set_of_states.
    - **def consume**: Avanza un solo caracter en el dfa, si no hay por donde avanzar retornamos false para denotar que no es un match.
    - **def dfa_states_prettify**: Cambia el nombre de los estados de tuplas a letras autogeneradas. En el algorimo no se usa puesto a que sólo se uso para debug más fácil.
    - **def e_closure**: Este método calcula recursivamente el closure de un sólo estado. Primero declaramos un set que contendrá todos los estados con los cuales podemos llegar con epsilon. Agregamos el estado actual. revisamos si la llave de épsilon existe, si no existe este es el base case. de lo contrario iteramos por todos los estados en los cuales podemos llegar con epsilon. Por cada de estos estados hay que verificar si tienen más aristas de épsilon entonces hacemos la llamada recursiva. Al terminar el for retornamos el closure del estado inicial que se pidió.

- **def match**: Nos permite meter un buffer de characteres y ver si hace match a la expresión regular actual que ya está en forma de DFA.
- **def subset_construction**: Turns nfa to dfa usando el algoritmo de subset construction especificado en el libro con epsilon closure.
- **def turn_to_dfa**: Llama a los métodos correpondientes para construir el DFA a partir de una construcción de NFA. Primero calculamos el closure de todos los estados indiscriminadamente. Guardamos el estado de inicio del dfa para registrarlo en el objeto. Construimos el DFA con el algoritmo de subset construction. Por ultimo, puesto que las llaves del diccionario son tuplas dichas tuplas las remplazamos por una letra autogenerada única para hacer que debugging sea más fácil.
- **extra funcionalidad**: el método **draw_dfa** importa librerías que grafican el DFA para poder visualizar el grafo. Este método permanece comentado puramente por razónes de rendimiento.

5. class scanner:
    - Esta clase abre el archivo de tokens y las mete a un OrderedDict, en el archivo de tokens se asume que hay precedencia, la primera regex que haga match al buffer de texto ingresado es la que gana por precedencia, de esta manera logramos diferenciar entre 'if' y 'id', 'if' aparece primero en el archivo y por ello tiene mayor precedencia. Cada token y su expresión regular son convertidas a NFA y después a DFA para poder trabajar con ellas. Scanner también abre el archivo a leer y empieza a hacer el match de acuerdo a reglas establecidas en el método de 'scan'. De cualquier error llegar a ocurrir mandamos el número de línea y el número de columna a una instancia de clase de error_msg para que imprima dónde las coordinadas del archivo al igual que en qué caracter falló.
    - build=True builds the regular expressions, build=False tries to load the regular expressions. save=True saves the regular expression bank into a file. save=False doesnt save them.
    - **def append_to_list_of_token**: Se encarga de agregar el token detectado al stream de tokens, Si el match contiene ':None' el match no se agrega sino se agrega como token terminal para ser identificado en el parser.
    - **def debug**: esta función imprime el resultado del lexer a un archivo llamado scan_debug.txt.
    - **def isascii**: Chequea si los characteres en el stream son ascii.
    - **def load_automata**: En la ocación que los DFAs ya se encuentren construidos y almacenados en un archivo, este método abre dicho archivo e importa los objetos serializados para ahorrarnos el trabajo de construir los NFAs y después DFAs, y en lugar de esto simplemente impotarlo.
    - **def next_char**: Avanza al siguiente caracter en la cadena de input.
    - **def peek_current**: Nos permite leer el caracter actual en el que estamos. Si ya estamos en EOF retorna None. Consiguientemente terminando el proceso.
    - **def peek_next**: Nos permite ver el siguiente caracter en el archivo para tomar decisiones.
    - **def produce_automata**: Toma los tokens del archivo tokens.yaml y los compila a NFA y después a DFA y los agrega al diccionario deterministic_finite_automata como el \: \.
    - **def recognize**: Toma un texto que es posiblemente un candidato a ser un token y las compara con todo el banco de expresiones regulares. El primero que haga match es el que retorna, si ninguno hace match retorna None y se produce un error. El primero significa que hay una precedencia, por ello ID esta de último.
    - **def save_automata**: Este método se encarga de tomar los DFAs ya construidos y guardalos con un serializador de objetos a un archivo, así la proxima vez podemos simplemente importar los DFAs y no construirlos, esto hace que corra más eficientemente y ahorra trabajo a la máquina.
    - **def scan**: Con los DFAs ya construidos hace el scanning y le pasa el texto al lexer para reconocer los tokens.
6. class error_msg:
    - Esta clase únicamente despliega errores al usuario si llegase a haber alguno. Después de imprimir el error en pantalla termina el programa con -1 por que hubo error.
    - Esta clase es la de mensajes de error. Illegal characters: toma como argumento la instancia del scanner y extrae el caracter y la posición de dicho caracter. Desupués para la ejecución. No_regex_match: toma con argumento la instancia del scanner y extrae las coordenadas. Después para ejecución.
    - **def illegal_character**: Despliega error de character illegal, puesto a que viene un caracter no aceptado en el lenguaje.
    - **def no_regex_match**: Despliega error de no match para el lexema dado.
    - **def unmatching_doublequotes**: Despliega error por que el segundo " está ausente, o hay uno de más en un string.
    - **def unmatching_singlequotes**: Despliega error por que el segundo ' está ausente, o hay uno de más en un character.

FASE 2 - PARSER:

DOCUMENTACIÓN DE FUNCIONES GLOBALES:

- **def is_nonterminal**: esta función chequea si un elemento es un no terminal chequeando si el primer caracter es '\<' y el último es '>'
- **def is_terminal**: esta función chequea si un elemento es un terminal chequeando si el primer caracter no es '\<' y el último no es '>'
- **def is_terminal_with_value**: esta función chequea si el elemento es tiene el símbolo '%' al principio y al final y si tiene el character ':' en medio, esto indica que usaremos el valor y descartaremos la etiqueta. Ejemplo ('%keyword:None%', 'if') significa que usaremos el 'if' descartaremos el primer elemento de la tupla.
- **def is_pseudo_terminal**: esta función chequea si el elemento es tiene el símbolo '%' al principio y al final, esto significa que tendrá 2 nodos en el árbol cuando parseamos.
- **def print_dict**: esta función simplemente imprime un diccionario de manera bonita.
- **def num_label_maker**: esta función genera números para nombrar los estados del DFA de manera única, no se usa puesto a que sólo se usó para debuggear.
- **def printable**: remplaza 'None' por el character 'ε' para poder imprimirlo de manera más fácil.
- **def parser_std_error**: esta función imprime el mensaje de error y para ejecución del programa al ser llamada, además imprime en qué función fallo para poder debuggear más fácil.

DOCUMENTACIÓN DE CLASES USADAS:

1. class lr_0
    - Esta clase se ocupa de poder armar el DFA de producciones usando el algoritmo de LR(0)
    - **def advance_dot_by_one**: Esta función simplemente toma un ítem de tupla, separa la izquierda de la derecha y la derecha avanza el punto a la vez.
    - **def calculate_transitions**: calcula las transiciones de un estado a otro tomando los items del estado y retorna los items con el '•' ya movido una posición para adelante si en cuyo caso es posible.
    - **def closures**: En esta función calculamos el cierre de todos los ítems de forma indiscriminada. Esto se hace para evitar la recursividad innecesaria de elementos calculados, en su lugar simplemente acceda al cierre de ese elemento específico en el diccionario de ya elementos calculados.
        1. self.closure_of_current_item es el estado actual que estamos fabricando, es un conjunto que almacena los índices del elemento actual. Ejemplo: `{0, 1, 2, 3}` estos son índices de artículos.
        2. self.productions_processed realiza un seguimiento de las producciones que ya tenemos procesado cuando estamos calculando recursivamente el cierre en self.get_closure_of_an_item. Siempre comenzamos registrando el inicio de la producción, ya que ya lo hemos procesado. en su totalidad porque es la producción inicial.
        3. llamar a la función self.get_closure_of_an_item, esta función nos ayudará a hacer estados más adelante, ya que los estados no son más que el cierre de varios rubros.
            - Este método hace uso de self.closure_of_current_item para agregar lo que self.get_closure_of_an_item encuentra de forma recursiva. el método no devuelve nada que se llame a sí mismo de forma recursiva, sino que devuelve datos al self. atributo, y agrega las producciones que ha procesado al self.productions_processed.
        4. Agregamos el cierre calculado al diccionario self.closure_table, que es el índice del elemento como un clave y su cierre como valor. Ejemplo: ` `{0: {1,2,3,4}, ...}` el elemento 0 tiene cierre: `{1,2,3,4}`

5. Repita esto para todos los elementos.

- **def create_state**: Esta función se ocupa de crear todos los estados recursivamente primero tomando el closure, formando un state con todos los items, posteriormente calculando todas las transiciones, los casos bases con cuando las transiciones son 0 y cuando el estado ya ha sido calculado.
- **def create_states**: Esta función crea todos los estados.
    1. self.create_state `(set ([0]))` crea todos los estados de forma recursiva en una sola llamada simplemente proporcionando el estado de inicio del kernel (la producción inicial con un '•' como primer elemento de rhs).
    2. Los estados y las transiciones ahora se han creado de forma recursiva, ejemplo: `{ Elementos de estado: (0, 2, 6, 10, 13, 16, 19, 21, 23, 25): estados de tránsito: {'<B>': (11, 17), 'd': (14,), 'g': (20,), 'h': (24,), '<A>': ( 3,), Ninguno: (22, 26), '<C>': (7,), '<S>': (1,)} ... }`
    3. Las claves del diccionario ya son estados, que se calculan con el cierre de un inicio conjunto de elementos, pero las transiciones son conjuntos de elementos no cerrados, lo que significa que se acaban de agregar, no son estados, sino que son los elementos que, si se realiza el cierre, se convierten en estados. En el siguiente ciclo for reemplazamos los elementos no cerrados por los estados reales. Ejemplo: `{ (0, 2, 6, 10, 13, 16, 19, 21, 23, 25): {'<C>': (7,), '<B>': (11, 17), 'h': (24,), '<A>': (3,), Ninguno: (22, 26 ), 'd': (14,), '<S>': (1,), 'g': (20,)} ... }`
        - Tenga en cuenta que a veces los estados no cerrados tienen un cierre de ellos mismos, por lo tanto, los estados no cerrados ya son estados, pero no lo sabemos hasta que los reemplazamos por los estados cerrados.
- **def determine_epsilon_states**: Este método se encarga de detectar estados que tengan reducciones pos épsilon, sin en cuyo caso las hay hace un append a la lista de estados reducibles por épsilon. También chequea que sólamente haya 1 reducción por épsilon, de lo contrario error por lenguaje ambiguo.
- **def get_actual_items**: Toma un set de indices al diccionario de items, va y toma las items como tal a partir de los índices y devuelve las items como tal. Transforma los índices a los items a los items en sí.
- **def get_closure_of_a_set_of_items**: toma un set de indexes de items, extrae del diccionario los items correspondientes y las busca en el diccionario de closures, después agrega todos los closures a un set y lo retorna, esto constituye la creación de un estado en el DFA.
- **def get_closure_of_an_item**: Este método calcula el cierre de un solo artículo.
    1. Primero, toma un conjunto de índices (índices que representan elementos en el diccionario self.items). Estos índices son siempre parte del cierre del artículo, por eso lo primero que hacemos es agréguelo al atributo self.closure_of_current_item al que este método tiene acceso global.
    2. El conjunto de índices de los elementos es inútil como índices, necesitamos los elementos reales, por lo que obtenemos los elementos utilizando el diccionario self.items y accediendo a través de la tecla al elemento real apuntado.
    3. Producciones pendientes: se trata de un conjunto que almacena los no terminales que están pendientes de procesar de forma recursiva cada vez que encontramos un elemento que tiene un '•' y luego un no terminal, debemos agregar todos los rhs de ese no terminal con un '•' al principio, lo haremos más tarde para que quede pendiente.
    4. Caso base, si no tenemos producciones pendientes debemos sumar, terminamos con la recursividad y retorno, por lo tanto, terminando la recursividad.
    5. Si no hemos terminado, es decir, no hemos roto la recursividad con el caso base, debemos procesar el pendiente. producciones, con un bucle for iteramos a través del conjunto de lhs pendientes para ser procesados. Hay una posibilidad ya hemos procesado ese lhs en una llamada recursiva anterior, por lo que debemos verificar cada lhs en espera de que no lo han agregado, ahorrando así la recursividad innecesaria, si no se ha agregado a las autoproducciones agregadas entonces debemos calcularlo, ya tenemos todas las reglas con el punto al principio almacenadas en self.closed_indexes simplemente vamos, los recuperamos y los agregamos (los índices) al conjunto pendiente_items que contiene los elementos derivados de El conjunto de pendientes_producciones, lo último que hacemos es registrar la producción como ya procesada para saber no procesarlo de nuevo en futuras llamadas recursivas.
    6. Llamada recursiva, hacemos la llamada recursiva, el set_of_items en este es el set pendiente_items, hacemos esto para verificar si cualquier otro elemento que agregamos tiene un '•' seguido de un no terminal que necesita cerrarse.
- **def get_or_create_renamed_state**: Este método se ocupa de crear el nuevo estado del diccionario de estado, si no existe el estado en el diccionario entonces lo crea, si existe entonces lo retorna.
- **def load_grammar**:
    1. Abrir archivo y leer contenido: Primero abre el archivo e importa los contenidos que son los producciones que definen la gramática, esta se almacena en self.grammar y se representa en forma de diccionario cuyas claves son los lados izquierdos y quién es el valor es el lado derecho. Ejemplo: `{'<S´>': [['<S>', '$']], '<S>': [['<A>', '<C>', '<B>'],. ..}.`
    2. Calcular elementos: luego tomamos la gramática recién importada y fabricamos los elementos de todos las producciones de la gramática, estas se almacenarán en self.items.
        - La variable de compensación se debe a que no queremos agregar un '•' después del dólarsigno ya que ese es el estado de aceptación, el desplazamiento nos impide iterar enmás tiempo y fabricando el artículo con el punto como último elemento después del dólar.
        - también estamos seguros de hacer una copia de la producción antes de agregar el punto, de lo contrario python asumirá que nos referimos al puntero a las reglas gramaticales reales, y modificará eso cuando queremos una copia de la regla y luego agregamos el punto.
        - los elementos se almacenan con un índice como clave y una representación de tupla del elemento como un valor, un ejemplo: `{0: ('<S´>', '•', '<S>', '$'), 1: ('<S´>', '<S>', '• PS`
        - El primer elemento de la representación de tupla es el lado izquierdo y el resto de la los elementos posteriores al primero son el lado derecho de la producción.
    3. Calcule self.reverse_item: self.reverse_item exactamente igual que self.items solo que las claves son las tuplas y los valores son índices, ejemplo: `{('<S´>', '•', '<S>', '$'): 0, ('<S´>', '<S>', '•', '$'): 1 , ...}.` Esto es solo por conveniencia y eficiencia, los elementos de acceso a los mapas de hash son un tiempo constante, Sin embargo, desea buscar por valor, debe tomar un tiempo lineal, por lo tanto, es mejor tener un versión inversa del mapa hash.
    4. Calcular self.closed_rules son una copia de self.grammar_rules pero todas las producciones tienen un punto en el comienzo del lado derecho. Esto es útil a la hora de calcular el cierre ya que si tenemos un punto seguido de un no terminal debemos sumar todas las producciones de ese no terminal con el punto en el principio, las reglas autocerradas obtienen como clave las lhs como de costumbre, y las rhs es la lista de producciones con el punto añadido en la primera posición. El lhs es el primer elemento de la tupla, el rhs es todo elementos restantes después del primero. Ejemplo: `{'<S´>': [('<S´>', '•', '<S>', '$')], '<S>': [('<S>', '•' , '<A>', '<C>', '<B>'), ...}`
    5. Calcular self.closed_indexes es exactamente lo mismo que self.closed_rules solo que en lugar de tener realmente el reglas con el punto tenemos el índice de los elementos que calculamos en el paso 2. Recuerde que los elementos tenían un índice en el diccionario / mapa hash estamos usando ese índice, ya que es solo un número único, no una tupla, por lo que ocupa menos espacio, porque un DFA de producciones para una gramática compleja puede ocupar fácilmente millones de estados, mejor Sea eficiente con la cantidad de memoria que toman los estados.
- **def load_lr_0_dfa**: va a buscar el archivo en el que se guardó el DFA de producciones serializado y lo importa para no tener que construirlo.
- **def save_lr_0_dfa**: toma el DFA de producciones ya construido y lo guarda en un archivo así la próxima vez podemos simplemente importar el DFA de producciones en lugar de tener que construirlo.

2. class lr_t_0:
    - Esta clase se ocupa de poder armar la tabla a partir del DFA de producciones usando el algoritmo de LR(0) PARSING TABLE.
    - **def construct_lr_0_table**: esta función se encarga de construir la tabla de lr_0. Primero toma cada estado normal y calcula las transiciones, si la transición es por elemento no terminal entonces hace GOTO, si es por un elemento terminal entonces hace SHIFT. Despues por cada estado hoja que se detecto se le agregar REDUCE a la tabla donde corresponda. Posteriormente por cada estado marcado como épsilon se agrega el REDUCE correspondiente. Si se encuentra que una posición ya está ocupado por otra operación entonces error por lenguaje ambiguo.
    - **def get_first**: esta método se ocupa de calcular la operación first sobre la gramática. Este método no se utilizó en el algoritmo puesto a que no se usó SLR sino LR(0) PARSING TABLE.

- **def unit_test**: este es unit test para chequear si la operación first y la operación follow están dando el resultado correctamente, no se utiliza por que no terminamos usando SLR.
- **def calculate_follow**: este método se ocupa de calcular la operación follow.
- **def beta_contains_epsilon**: Este método
- **def is_nullable**: simplemente chequea épsilon está en la tabla de firsts, no se usa por que LR(0) PARSING TABLE no lo requiere.
- **def follows**: Hace la operación follow recursivamente.
- **def calculate_firsts**: Hace la operación firsts recursivamente.
- **def first_sequence**: Se utiliza para el firsts, pero no usamos el firsts operation puesto a que cambiamos de algoritmo de SLR a LR(0) PARSING TABLE.
- **def index_grammar_rules**: toma cada producción de la gramática y asigna un índice único para poder hacer reduce por una producción en específico.
- **def detect_left_recursion**: Este método detecta 'left recursion' que era un problema para SLR, pero no es problema para el LR(0) PARSING TABLE, entonces no la usamos.
- **def make_pending_follows**: Se utiliza para el follow, pero no usamos el firsts operation puesto a que cambiamos de algoritmo de SLR a LR(0) PARSING TABLE.
- **def firsts**: calcula la operación firsts recursivamente.
- **def calculate_feasable_first**: calcula la operación firsts que no requieren recursión, es decir, los casos bases para poder ahorrarnos esto en la recursión de la operación first.

3. class parser
    - **def str_tree**: esta función devuelve el AST en string, en un formato entendible para debug en la terminal, no es llamada puesto a que sólo está ahi para poder debuggear.
    - **def debug**: imprime el resultado del parser, que es el AST a un archivo llamado tree_debug.txt.
    - **def parse**: usa la tabla de la fase anterior para armar el AST con las operaciones correspondientes.
    - **def shift**: Hace la operación SHIFT.
    - **def reduce**: Hace la operación REDUCE.
    - **def goto**: Hace la operación GOTO.
    - **def print_tree**: esta función imprime el AST en un formato entendible para debug en la terminal, no es llamada puesto a que sólo está ahi para poder debuggear.
    - **def error**: depliega error si el parser detecta que el stream de tokens está incorrecto gramáticamente.

FASE 3 - SEMANTIC:

DOCUMENTACIÓN DE FUNCIONES GLOBALES:

- **def semantic_std_error**: despliega el error y el método en el que ocurrió.

DOCUMENTACIÓN DE CLASES USADAS:

1. class semantic:
    - Los chequeos semánticos especificados son:
        1. Chequeo de qué retornan las expresiones y si son compatibles al ser asignadas a una variable.
        2. Chequeo de unicidad de variables, que las variables sean únicas en cada scope.
        3. Chequeo de declaración de variables, que las variables estén declaradas antes de ser usadas.
    - **def traverse**: esta función recorre el AST recursivamente para encontrar producciones de interes así poder conducir el chequeo semántico.
    - **def return_t_of_var**: esta función retorna el tipo de una variable declarada, si no está declarada entonces error.
    - **def head_expr_return**: esta función se ocupa de retornar el tipo de dato que retornará una expresión, al detectar el tipo que retornará entonces se almacena el puntero en un diccionario de expresiones junto con el tipo de dato que retornará para poder chequear a la hora de hacer una asignación si son tipos compatibles.
    - **def debug**: imprime el diccionario a un archivo, en el cual se almacena el puntero a cada expresión junto con el tipo que retornará. Esto se utiliza para chequear si lo que devuelve la expresión y las asignaciones a variables son compatibles.
    - **def closing_curly**: cuando se detecta '}' se debe hacer un pop a el scope_stack en donde se desechan todas las variables del scope actual y se hace pop de la tabla de símbolos como tal para simbolizar que hemos terminado con este scope.
    - **def check_if_var_exists**: chequea si la variable está declarada ya sea en el scope actual o en el scope anterior.
    - **def assignment_statement**: se encarga de llamar a head_expr_return para ver qué retorna la expresión y chequea si lo que retorna es compatible con el tipo de varible a la cual se le hará la asignación, si no es compatible entonces error.
    - **def var_not_declared**: error de variable utilizada sin antes haberla declarado.
    - **def add_variable_to_scope**: agrega la variable al scope actual.
    - **def unique_variables_and_scope_access**: hace un append de un nuevo scope, sigue atravesando el árbol en búsqueda de variables. Al terminal hace un pop al stack de scopes.
    - **def open_curly**: cuando se detecta un '{' se debe hacer un push de una nueva tabla de símbolos al scope_stack, puesto a que entramos a un nuevo scope.
    - **def unique_var_error**: el error despliegado al detectar que hay dos variables detectadas en un mismo scope con el mismo nombre.
    - **def method_decl**: almacena el tipo returnado por method_decl, además de agregar el nombre del método a la tabla de símbolos para poder chequear que los métodos en una misma clase tengan nombres únicos.
    - **def id_**: se encarga de agregar variables a la tabla de símbolos del scope actual, si no es declaración de variables entonces se asume que se está usando la variable y se va a chequear a scopes anteriores y el actual para ver si existe, sino existe error.

FASE 4 – GENERACIÓN DE CÓDIGO:

DOCUMENTACIÓN DE FUNCIONES GLOBALES:

tag_gen_with_name tag_gen_for_dot_data codegen_std_error codegen_std_warning instruction occupy_temp_reg unoccupy_temp_reg method_name_gen

DOCUMENTACIÓN DE CLASES USADAS:

1. class codegen:
    - La clase codegen se ocupa de generar el código de assembler mips. Contiene una función por cada producción de interes en el lenguaje y a partir de esto hace la generación de código.
    - **def write_executable**: escribe el archivo de assembler.
    - **def method_decl_kleene**: se encarga de la producción \<method_decl*>
    - **def expr**: se encarga de la producción \<expr>
    - **def literal**: se encarga de la producción \<literal>

- **def args_list**: se encarga de la producción \<args_list>
- **def field_decl**: se encarga de la producción \<field_decl*>
- **def else_block**: se encarga de la producción \<else_block>
- **def statement_kleene**: se encarga de la producción \<statement*>
- **def program_dash_handler**: inicia la recursión de \<program">
- **def get_subscript_val**: agarra el valor del subscript.
- **def block**: se encarga de la producción \<block>
- **def get_var**: va a buscar la variable al scope actual y si no la encuentra va a buscar a scopes anteriores.
- **def var_decl_kleene**: se encarga de la producción \<var_decl*>
- **def add_to_scope_space**: agrega espacio de la variable al scope para poder saber cuánto allocar y desallocar.
- **def add_var_to_scope**: agrega una variable al scope actual.
- **def field_decl_aux**: se encarga de la producción \<field_decl*_aux>
- **def statement**: se encarga de la producción \<statement>
- **def field_decl_kleene**: se encarga de la producción \<field_decl*>
- **def initiate**: se encarga de la llamar a los métodos correspondientes para iniciar con la generación de código.
- **def field_decl_handler**: se encarga iniciar la recursión a la producción \<field_decl>
- **def method_call**: se encarga de la producción \<method_call>
- **def callee_header**: se encarga guardar los registros al stack correspondientes a una función.
- **def get_pseudo_terminal_value**: retorna el valor apuntado por una producción pseudo terminal. Por ejemplo: `%int_lilteral% -> 56` , retorna el `56` .
- **def callee_header_main**: se encarga de guardar los registros al stack correspondientes de la función 'main'.
- **def assignment_statement**: se encarga de la producción \<assignment_statement>
- **def method_args**: se encarga de la producción \<method_args>
- **def method_arguments_appender**: se encarga de agregar variables como argumentos al scope para así poder allocar y desallocar los arguments correspondientes.
- **def method_decl_aux**: se encarga de la producción \<method_decl*_aux>
- **def print_var**: se encarga de la producción \<print_var> que imprime un registro en mips.
- **def traverse**: se encarga buscar la producción \<program"> y empezar la recursión.
- **def append_instructions**: se encarga de agregar instrucciones en mips producidas a la sección correspondiente.
- **def var_array_list**: se encarga de la producción \<var_array_list>
- **def add_space_to_scope**: agrega espacio al scope para saber cuando allocar y desallocar en \<var_decl>
- **def callee_ender**: se encarga de restaurar los registros correspondientes al terminar una función.
- **def method_args_dash**: se encarga de la producción \<method_args">.
- **def print_str**: se encarga de la producción \<print_str> que en el programa imprime un string literal en mips.
- **def opening_curly**: se encarga agregar un scope al scope_stack en el cual se agregarán las variables correspodientes.
- **def callee_ender_main**: se encarga de restaurar los registros correspondientes al terminar el callee de la función 'main'.
- **def get_var_reg**: se encarga de hacer el load en mips para traer la variable de RAM a un registro.
- **def check_for_warnings**: se encarga de chequear que los registros estén desocupados para asegurar que ningun método se quedó con un registro y nunca se desallocó.
- **def type_or_void**: se encarga de la producción \<type|void>
- **def method_decl_handler**: se encarga de la producción \<method_decl*> iniciando la recursión al método self.method_decl_kleene.
- **def var_decl**: se encarga de la producción \<var_decl> agarra cada nombre de variable, su tipo, y el offset del stack correspondiente.
- **def closing_curly**: se encarga de la } que significa que debemos desallocar todas las variables del scope actual y darle pop al stack de scopes.

COMMAND LINE INTERFASE

El command line interfase se usa de la siguiente manera:

```
python <archivo.py> --target <stage> --debug <stage --o <new_name>
```