Baby Duck

Por

Rolando Esteban Enríquez Limón

Profesores

Ivan Mauricio Amaya Contreras
Jesús Guillermo Falcón Cardona
Elda Guadalupe Quiroga González

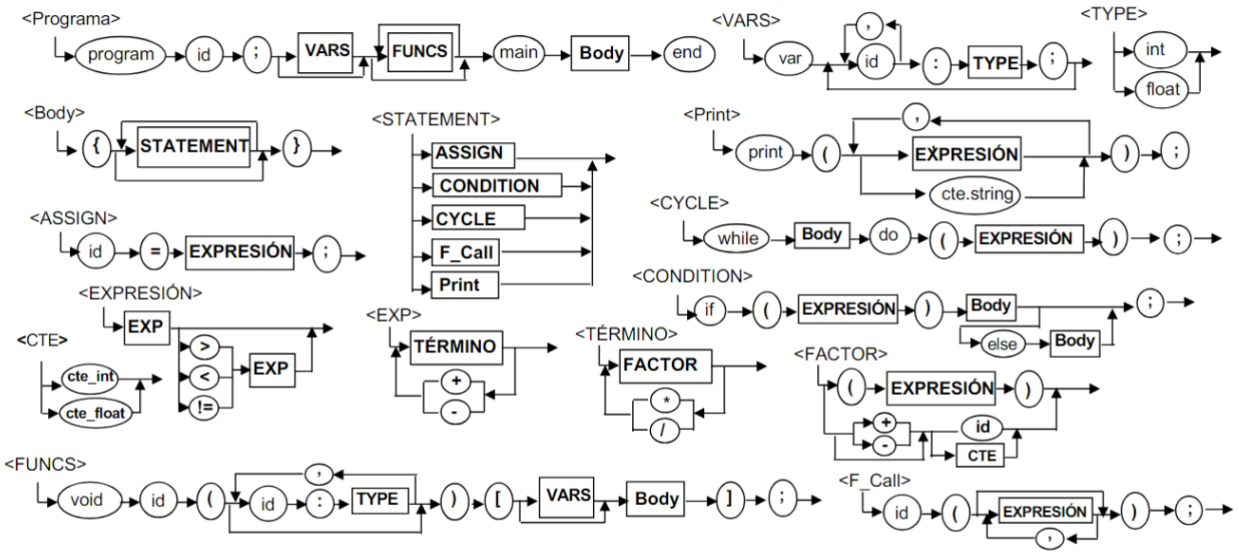
Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 501)

Fecha

15 nov 2023

Instituto Tecnológico y de Estudios Superiores de Monterrey

Gramática de BabyDuck



Primer entregable: Gramática

Código:

He utilizado la herramienta de Anklr como el generador de scanners y parsers automático, por lo que después de instalarla, desarrollé la gramática para un archivo g4 y luego de eso, utilicé python como vía de conexión entre el archivo y Anklr.

```
C:\Users\enriq\OneDrive\Escritorio\PythonProjects\CVEHunter>python baby_duckSecondParser.py
Enter rule:  ProgramRuleContext
Enter rule:  ProgramContext
Enter rule:  ProgramaContext
Exit rule: ProgramaContext
Enter rule:  VariablesContext
Enter rule:  VarContext
Exit rule: VarContext
Enter rule:  TypeContext
Enter rule:  IntContext
Exit rule: IntContext
```

Para Anklr era necesario crear dos archivos extra a los que te genera automáticamente, pues la herramienta te generaba el lexer y el parser, pero uno también debía crear lo que era el Listener y un archivo que junte las tres cosas y especifique el archivo que se tomará de ejemplo para la gramática.

Gramática en el primer entregable:

```
expression: exp ((>' | '<') exp | '!=' exp)*;
exp : term ('+' term | '-' term)*;
term: factor ('*' factor | '/' factor)*;
factor: ('+'| '-')? (Id | CTE_INT | CTE_STRING | '(' expression ')');
body: '{' statement* '}';
assign: Id '=' expression ';';
condition: if '(' expression ')' body (else body)? ';';
cycle: while body do '(' expression ')' ';';
print: 'print' '(' (expression | CTE_STRING) (',' (expression | CTE_STRING))* ')' ';';
functionCall: Id '(' (expression (',' expression)*)? ')' ';';
type: int | float;
variables: var listvars*;
listvars: listaId ':' type ';';
listaId : Id idExtra;
idExtra : (',' listaId)?;
function: void Id '(' parameters* ')' '[' variables? body;
parameters : parameter (',' parameter)*;
parameter: Id ':' type;
program: programa Id ';' variables?(function)* main body end;
statement: (assign | condition | cycle | print | functionCall);
```

Gramática resultante en formato BNF:

<programRule> ::= program EOF;

<programa> ::= 'program';
<Id> ::= [a-zA-Z]+[0-9]*;
<main> ::= 'main';
<end> ::= 'end';
<var> ::= 'var';
<void> ::= 'void';
<int> ::= 'int';
<float> ::= 'float';
<else> ::= 'else';
<if> ::= 'if';
<while> ::= 'while';
<do> ::= 'do';
<CTE_STRING> ::= ""."*?"";
<CTE_INT> ::= [0-9]+;
<CTE_FLOAT> ::= [0-9]+ '.' [0-9]+;
<WS> ::= [\t\r\n]+ -> skip;

<expression> ::= <exp> (('>' {symbol_table1.push_operator('>')} | '<' {symbol_table1.push_operator('<')}) <exp> | '!=' {symbol_table1.push_operator('!=')} <exp>)* {symbol_table1.push_term_mas_menos()};
<exp> ::= <term> {symbol_table1.push_term()} ('+' {symbol_table1.push_operator('+')} <exp> | '-' {symbol_table1.push_operator('-')} <exp>)*;

<term> ::= <factor> {symbol_table1.push_term_multiplication()} ('*'
{symbol_table1.push_operator('*')} <term> | '/' {symbol_table1.push_operator('/')} <term>)*;
<factor> ::= ('+'| '-')? (<Id> {symbol_table1.push_factor(<Id>.text, None, False)} | <CTE_INT>
{symbol_table1.push_factor(<CTE_INT>.text, "int", True)} | <CTE_FLOAT>
{symbol_table1.push_factor(<CTE_FLOAT>.text, "float", True)} | '('
{symbol_table1.push_parentesis('(')} <expression> ')' {symbol_table1.pop_parentesis()});

<body> ::= '{' <statement>* '}';
<assign> ::= <Id> '=' <expression> {symbol_table1.assign_value(<Id>.text, '=')}';';
<condition> ::= if '(' <expression> ')' {symbol_table1.push_if()} <body> (else
{symbol_table1.push_else()} <body>)? {symbol_table1.push_if_finish()}';';
<cycle> ::= while {symbol_table1.push_while()} <body> do '(' <expression> ')'
{symbol_table1.push_if()} {symbol_table1.push_while_end()} ';';
<print> ::= 'print' '(' (<expression> {symbol_table1.print_function()} | <CTE_STRING>) (','
(<expression> | <CTE_STRING>))* ')' ';' {symbol_table1.printSymbols();};
<functionCall> ::= <Id> '(' (<expression> (',' <expression>)*)? ')' ';';
<type> ::= int | float;
<variables> ::= <var> <listvars>*;
<listvars> ::= <listaId> ':' <type> ';' {symbol_table1.add_symbol(<listaId>.text, <type>.text,
current_scope);};
<listaId> ::= <Id> <idExtra>;
<idExtra> ::= (',' <listaId>)?;
<function> ::= {global current_scope}void {current_scope+=1} <Id> '(' <parameters>* ')' '['
<variables>? <body> {symbol_table1.add_function(<Id>.text, <parameters>.text, <variables>.text)}']'
';' {current_scope-=1; symbol_table1.pop_function(<Id>.text, <parameters>.text, <variables>.text);};
<parameters> ::= <parameter> (',' <parameter>)*;

<parameter> ::= <Id> ':' <type> {symbol_table1.add_symbol(<Id>.text, <type>.text, current_scope)};

<program> ::= <programa> <Id> ';' <variables>? {symbol_table1.add_function(<Id>.text, 0,
<variables>.text)} (<function>)* <main> <body> <end> {symbol_table1.maquina_virtual();};
<statement> ::= (<assign> | <condition> | <cycle> | <print> | <functionCall>);

Tabla de símbolos y tabla de funciones.

Para este entregable cambié un poco la gramática (por lo que también la actualicé en la primera página del documento), esto con el fin de poder implementar ciertas funciones en el archivo de gramática de Anklr, pues muchas veces resultaba complicado obtener cierta información de la variables de las respectivas funciones declaradas.

Con ello, es importante recalcar que se inicializó la clase de la tabla de símbolos desde la gramática para poder utilizar las clases dentro de la gramática. Esta clase es llamada SymbolTable y la utilicé como una estructura para guardar tanto las variables, como las funciones en sus respectivos diccionarios. Utilizar diccionarios en python fue la manera más

sencilla para mi, ya que al ser una estructura sencilla, es fácil obtener el dato que se desea sin tener que estar iterando tanto por toda la clase o el diccionario persé.

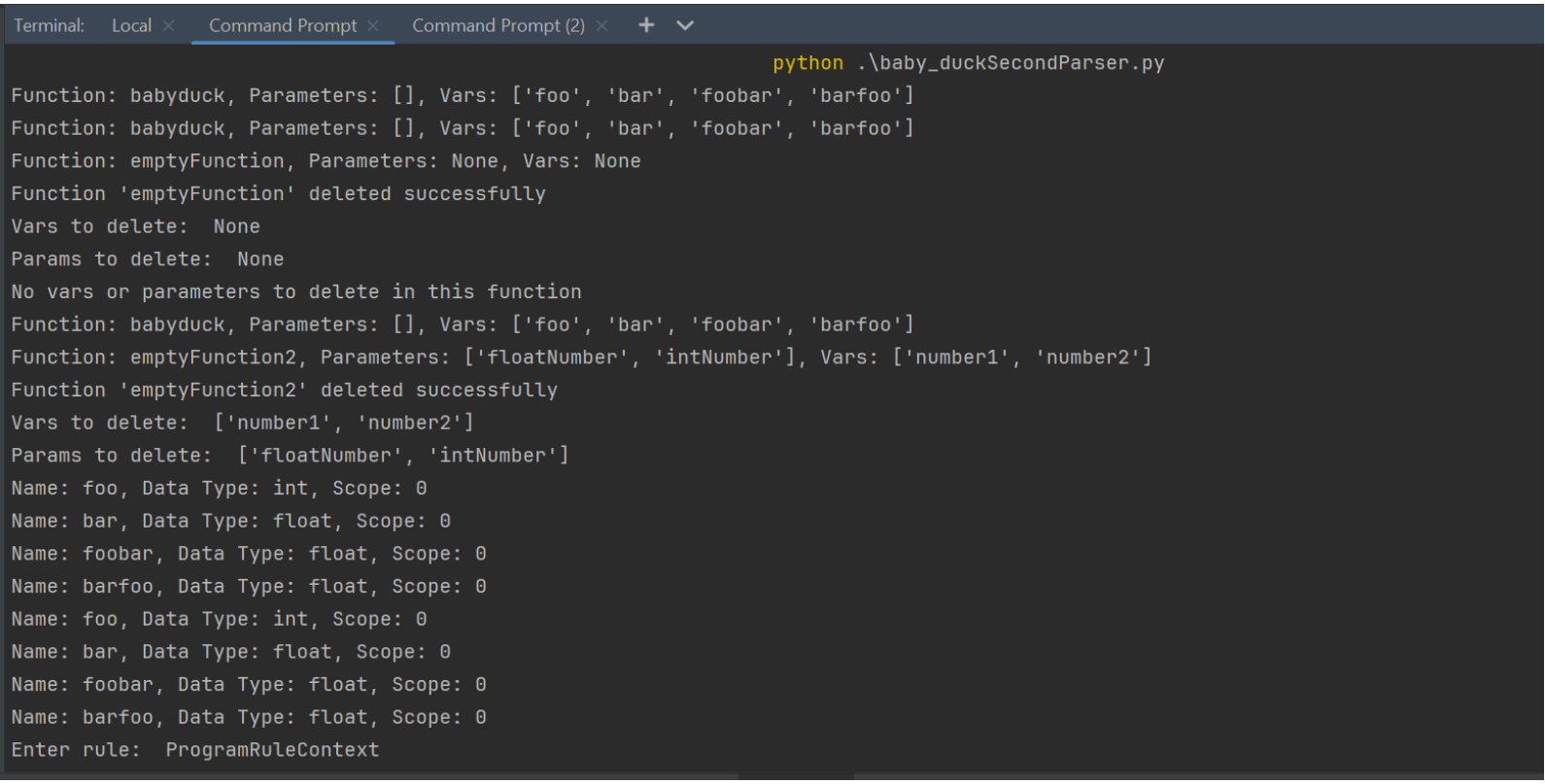
Estructura del diccionario que guarda los símbolos:

- <Nombre de la variable>
  - <Tipo de dato>
  - <Scope>
  - <Número de memoria>
  - <Valor>

Puntos neurálgicos:

En la gramática se encuentran como puntos neurálgicos las secciones de: *<variables>, <listavars>, <listaId> y el <idExtra>.* En esta sección se definen los tokens y como se maneja el alcance con las listas de variables (<variables>, <listvars>). Aquí son los puntos en donde se agrega a la tabla de símbolos.

Output del Entregable 2:



Explicación del output:

En este caso la aplicación muestra cómo <function>, <parameters>, <parameter> y <functionCall> son puntos neurálgicos importantes, pues aquí se definen y se llaman las

funciones, así como también la gestión de sus parámetros. También se muestra cómo se maneja el alcance de las funciones y cómo se relacionan con la tabla de símbolos.

Como ejemplo de la dinámica entre la tabla de símbolos y la tabla de funciones está la de <program> donde tratará a todas las variables que se asignen dentro de ahí con la mayor jerarquía, osea scope “0”. Después de esto siguen las otras asignaciones de funciones, que dependiendo a la cantidad de funciones embebidas, su scope irá aumentando.

En el output del programa está la emptyFunction, donde no hay parámetros ni variables asignadas en ella, por lo que en este caso no se agregará nada en la tabla y después de recorrer toda la función, está será borrada. Luego de aquello, sigue la emptyFunction2, la cual si tiene funciones y parámetros, con un scope a “1”, donde después de recorrer toda la función, estas variables y parámetros se borrarán (al igual que la función, claro está).

Al final imprimo todas las variables restantes de la tabla de símbolos. Estas se imprimen dos veces, porque la función se manda a llamar en el print(), el cual está dos veces en el código de ejemplo que usé.

Para este entregable no hice nada en el main, pues tiene asignaciones.

Código:

En este código usamos un diccionario llamado symbols donde se guardarán todas las variables junto a su tipo, su scope, su memoria y su valor. Este último se usará para guardar los valores en la máquina virtual.

```
def add_symbol(self, name, data_type, scope):
    newNames = name.split(",")
    for newName in newNames:
        if newName in self.symbols and self.symbols[newName]["scope"] == scope and self.symbols[newName]["scope"] == 0:
            raise ValueError(f"Symbol '{newName}' redeclared in the same scope")
        self.symbols[newName] = {"data_type": data_type, "scope": scope, "memory_data": 0, "valor": 0}
        self.add_memory_number(newName)
    return self.symbols
```

Para agregar las funciones, estas se agregaron también en un diccionario llamado functions. En este también se ligaron las funciones y sus variables con la función al momento de ser guardadas.

```python
def add_function(self, name, param_names, var_names):
    definitive_vars = []
    definitive_params = []
    if name in self.functions:
        raise ValueError(f"Function '{name}' redeclared")
    if var_names or param_names:
        if var_names:
            new_vars = var_names[3:].replace(";", ",").replace(":", ",")
            var_list = new_vars.split(",")
            for var in var_list:
                if var in self.symbols:
                    definitive_vars.append(var)

        if param_names and isinstance(param_names, str):
            new_params = param_names.replace( _old: ";",  _new: ",").replace( _old: ":",  _new: ",")
```

Para este entregable todavía no se alcanzó a realizar el cubo semántico y tampoco se hicieron cambios en la sección de main en el programa, puesto que ahí se encontraban las asignaciones, las cuales contendrían cuadruplos.

Entregable 3

Cuádruplos y el cubo semántico.

Para este entregable se hicieron cambios drásticos en la gramática, se creó una nueva clase de cubo semántico y se agregaron múltiples nuevos diccionarios y listas.

En este punto comienza la identificación de expresiones, se trabaja sobre las asignaciones y sobre los ciclos, generando en cada uno de estos los cuádruplos necesarios para poder comprender la estructura semántica del programa, así como identificar sus errores.

En este entregable decidí utilizar la sugerencia de la profesora de asignar direcciones de memoria a todos los tokens del programa, para así no tener problemas con los tipos de datos.

Las memorias se arreglan de la siguiente manera:

- Variables globales enteras: 1000 - 2000
- Variables globales flotantes: 2000 - 3000
- Variables locales enteras: 3000 - 4000
- Variables locales flotantes: 4000 - 5000
- Variables temporales enteras: 5000 - 6000
- Variables temporales flotantes: 6000 - 7000

- Variables temporales booleanas: 7000 - 8000
- Variables constantes enteras: 8000 - 9000
- Variables constantes flotantes: 9000 - 10000
- Variables constantes strings: 10000 - 11000

Los operadores también tienen su propio diccionario donde se les asigna un número de memoria. Después de esto, se crearon cuatro diferentes listas, las cuales serán utilizadas como stacks. Estas son: los stacks de operandos, operadores, de tipos y de saltos.

Cubo semántico.

El cubo semántico fue una clase sencilla con solo un atributo el cual es un diccionario llamado “cube” y dos métodos “define” y “get_result_type”.

En el método de “define” se toman 4 parámetros, los cuales son el operador, el tipo 1, el tipo 2 y el tipo resultante. Después, al atributo del cubo se le asignan los primeros tres parámetros y el resultado se guardará en result_type.

En el segundo método “get_resulty_type”, se tomarán 3 parámetros: operador, tipo 1 y tipo 2. Aquí intenta encontrar la llave del diccionario cubo que tenga estos 3 parámetros y si no la hay, regresará un error.

Al final en esta clase, se definirán las reglas de tipos con el método de define y si uno quiere hacer alguna comprobación tendrá que llamar el segundo método y en dado caso de que haya algún error, la gramática no será aceptada.

Como reglas en el cubo semántico tenemos que ningún tipo de dato se debe de combinar. Por ejemplo entero con flotante o booleano, por lo que las operaciones deben de estar acorde al tipo de dato que se está utilizando.

Ejemplo de las reglas impuestas en el cubo semántico:

```
self.define('/', 'int', 'int', 'int')
self.define('+', 'float', 'float', 'float')
```

Operador	Operando 1	Operando 2	Resultado
+	int	int	int
-	int	int	int
*	int	int	int
/	int	int	int
+	float	float	float
-	float	float	float

*	float	float	float
/	float	float	float
=	int	int	int
=	float	float	float
<	int	int	bool
<	float	float	bool
>	int	int	bool
>	float	float	bool

Cuadruplos para operaciones aritméticas y lógicas en expresiones.

Para el reconocimiento de expresiones todo comienza en *<fact>,* donde se obtiene el primer id y este es pusheado al stack de operandos y tipos. En esta parte de la estructura, se define si un token que llega es constante o ya se encontraba en la tabla de símbolos. Si un valor es constante, también se le define su número de memoria y se guarda en un diccionario diferente, el cual es el de constantes y tiene la siguiente estructura.

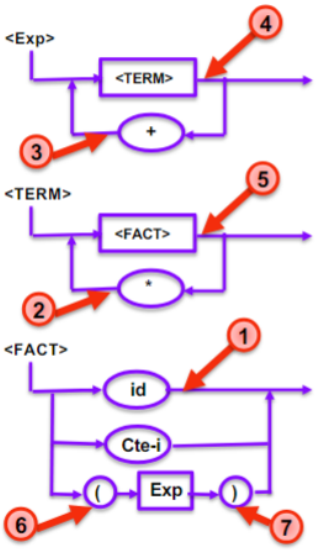
Estructura del diccionario de constantes:

- <Número de memoria>
    - <Valor de la constante>

Luego en base a la jerarquía, primero reconoce multiplicaciones y divisiones, luego sumas y restas y al final operaciones lógicas de mayor qué y menor qué, los cuales se guardan en los stacks de operadores.

Al llegar tanto <term> o <exp>, se ejecuta la función de push_term_multiplicación() para <term> y push_term() para <exp>, pero ambas tienen una funcionalidad similar.

- Verifica si hay operadores unarios pendientes de suma o resta (+, -) o si es en <term> es (*, /).
- Extrae los operandos y el operador.
- Determina el tipo resultante usando la semantic_cube.
- Asigna un nuevo temporal para el resultado y genera el cuádruplo correspondiente.
- Actualiza la pila de operandos (PilaO) y la pila de tipos (Ptypes) con el resultado.



Al final de eso, puede reconocer otro término que también representaría un id y operadores. Dentro de este id, puede venir también una constante u otra expresión con paréntesis. Todo esto se irá pusheando en los respectivos stacks de operadores, operandos y tipos.

```
expression: exp (('>' {symbol_table1.push_operator('>')} | '<'
{symbol_table1.push_operator('<')}) exp | '!='
{symbol_table1.push_operator('!=')}
exp)*{symbol_table1.push_term_mas_menos()};
exp : term {symbol_table1.push_term()}('+'
{symbol_table1.push_operator('+')} exp | '-'
{symbol_table1.push_operator('-')} exp)*;
term: factor {symbol_table1.push_term_multiplication()}('*'
{symbol_table1.push_operator('*')} term | '/'
{symbol_table1.push_operator('/')} term)*;
factor: ('+'| '-')? (Id {symbol_table1.push_factor($Id.text, None,
False)} | CTE_INT {symbol_table1.push_factor($CTE_INT.text, "int",
True)} | CTE_FLOAT {symbol_table1.push_factor($CTE_FLOAT.text,
"float", True)} | '(' {symbol_table1.push_parentesis('(')}
expression ')' {symbol_table1.pop_parentesis()});
```

Para este proceso, el código utiliza múltiples funciones que realizan los procesos:
1-. push_operator(): Pushea los operadores al stack de operadores.
2-. push_term_multiplication(): Toma el último valor del stack de operadores y verifica si es multiplicación o división, para después comparar los operandos y el operador con el cubo semántico, para así después realizar el cuádruplo usando el resultado de esa comparación.
3-. push_term(): Realiza lo mismo, pero con sumas y restas.
4-. push_term_mas_menos(): Realiza lo mismo con los signos de menor que y mayor que.
5-. push_factor(): Identifica los tokens, separándolos como constantes en diccionarios y agregando el valor en los stacks, con sus respectivas direcciones de memoria. Si no es constante, simplemente se agregan a las pilas de operandos y tipos.

Con esta estructura base, se guarda en una lista y se compara con el cubo semántico para comprobar que la semántica de la estructura en relación a los tipos de datos. Al tener ya el resultado, esto ya se puede meter al cuádruplo como:

(operador, operando1, operando2, resultado)

En esta última parte es importante recalcar que:

- Se utilizan temporales (ti, tf, tb) para almacenar resultados intermedios.
- Los cuádruplos generados se almacenan en la lista Quad.
- Se manejan casos de error si la operación no es válida según el cubo semántico.

Assign

El caso de assign es similar al de push_factor(). Este es un punto neurálgico que identifiqué el entregable pasado, pero que no realicé por el uso de cuádruplos. Como su nombre indica, esta función asigna valores a las variables que ya se encontraban en la tabla de símbolos.

Metodología

```
assign: Id '=' expression {symbol_table1.assign_value($Id.text,
'=')}';';
```

Lo primero que hice fue identificar el id, osea el nombre de una variable a la que se le asignará un valor.

Después de esto viene el operador, el cual siempre será “=” y luego de ello, vendrá la expresión, la cual asignará a la variable.

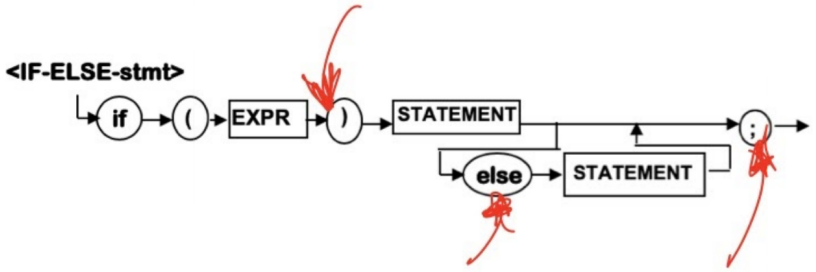
Proceso de asignación.

- Se agrega la dirección de memoria y el tipo de la variable a la pila (self.PilaO y self.Ptypes).
- Se agrega el código del operador de asignación (=) a la pila de operadores (self.Poper).
- Se recuperan los operandos y el operador de las pilas. Para los operandos son los últimos dos valores del stack de operandos y de tipos, para el operador es el último valor del stack de operadores.
- Se utiliza el método get_result_type para determinar el tipo resultante de la asignación.
- Se genera un cuádruplo con el operador de asignación, la variable de destino, el resultado (en este caso, None), y el operando de la expresión.
- El cuádruplo se agrega a la lista self.Quad.

Luego de ello, se genera el cuádruplo: **(operador, target, None, operando1)**

IF Statements

El IF se separó en 3 fases en este código.



Primera función del if:

- Primero la función verifica que la expresión de la condición sea de tipo booleano. Si no es booleano, se generará un error, pues habría un error en la concordancia de los stacks.
- Obtiene el resultado de la expresión de la condición.
- Si el resultado sí es booleano, se genera un cuádruplo GOTOF que se salta al código dentro del bloque if si la condición es falsa donde se genera un cuádruplo con el resultado de la variable booleana temporal.
- Almacena la posición del cuádruplo GOTOF en la pila Psaltos para futuras referencias.

Segunda función del if

- Genera un cuádruplo GOTO para saltar al final del bloque else después de ejecutar el bloque if. Al principio este es un cuádruplo vació para saltar a la instrucción de “GOTO”.
- Obtiene la posición del cuádruplo GOTOF almacenado previamente en la lista de cuádruplos.
- Modifica el cuádruplo GOTOF para que apunte al inicio del bloque else.
- Almacena la posición del nuevo cuádruplo GOTO en Psaltos para futuras referencias.

Tercera función del if

- Obtiene la posición de la instrucción GOTOF almacenada en Psaltos.
- Modifica el cuádruplo GOTOF para que apunte al final del bloque if.

- Imprime información de depuración.

En la gramática se ve representado así:

```
condition: if '(' expression ')' {symbol_table1.push_if()} body
(else {symbol_table1.push_else()} body)?
{symbol_table1.push_if_finish()}';';
```

While Statement

Para que el while funcione también se basa igualmente en tres funciones.

Primera función del while

- Almacena la posición actual en la lista de cuádruplos (Quad) en la pila Psaltos. Esto se hace para recordar dónde empieza el bucle y dónde debería apuntar el GOTO al final del bucle. Esto es importante porque en el último cuádruplo se debe de especificar la dirección donde comienza todo el while de nuevo para mantener un ciclo de loop.

Cuerpo del bucle “body”

- Se ejecuta el código dentro del bucle.

Segunda función del while

En la segunda sección del while volví a utilizar la primera función del if porque es básicamente la misma situación, donde se tiene que crear un cuádruplo temporal y se debe de guardar el salto a donde finaliza el statement.

- Se evalúa la expresión en la condición del bucle (expression).
- Se generan cuádruplos relacionados con la condición del bucle (por ejemplo, un GOTOF).

Tercera función del while

En la última parte del while, se genera un último cuádruplo donde se especifica el inicio a donde debe de regresar el loop.

- Obtiene la posición de inicio del bucle desde la pila Psaltos.

- Obtiene la posición actual en la lista de cuádruplos (Quad).
- Genera un cuádruplo GOTO que apunta al inicio del bucle.
- Modifica el cuádruplo en la posición de inicio del bucle para que apunte al final del bucle.

```
cycle: while {symbol_table1.push_while()} body do '(' expression
')' {symbol_table1.push_if()} {symbol_table1.push_while_end()}
';';
```

Entregable 4:

Máquina virtual

Para la máquina virtual se tomaron en cuenta todos los cuádruplos generados por el programa. La función de la máquina virtual realiza un loop que recorre toda la lista de los cuádruplos y uno por uno va verificando cuál es el operador, comparándolo con el diccionario de operadores ya asignados.

Cada operador tiene una condición diferente. Por ejemplo la suma, donde se toma cada cuádruplo y después de tomar el valor real de cada dirección de memoria de los operandos, se realiza la operación.

Para guardar los resultados de las operaciones, creé una lista de temporales. Esta lista sería después accesible por el operador “=”, la cual le asignará al operador correspondiente el valor del resultado.

Caso hipotético:

```
x = 10;
z = x;
while {x = x - 1; print(x);} do (
   x > 5
 );
```

Diccionario de símbolos

Key	Data Type	Memory Data	Valor	Scope
x	int	1001	10	0
tb7000	bool	7000	True	0
z	int	1002	10	0

Tabla de temporales

9
8
7
6
5

Esto sería así, hasta llegar a los GOTO, donde ahí se le asignaría el contador del while, al valor del resultado del cuádruplo actual. Lo que quiere decir que:

- Obtiene la posición a la cual saltar desde el cuádruplo actual (self.Quad[quad][3]).
- Actualiza la variable quad para que apunte a la nueva posición.

En el caso del GOTOF, se tomaría el resultado del cuádruplo anterior, pues es la evaluación booleana y si el resultado es falso, el contador sería igual al resultado del cuádruplo actual. Por lo que quiere decir que:

- Recupera el resultado de la última evaluación (resultado).
- Si resultado es False, obtiene la posición a la cual saltar desde el cuádruplo actual (self.Quad[quad][3]).
- Actualiza la variable quad para que apunte a la nueva posición.

El caso del “print” también es distinto, puesto que ahí busca que el operador sea “print” y aquí determina el valor a imprimir, que puede ser una variable, una constante o una expresión resultante de una operación.