DOCUMENTACION TALLER COMPILADOR

El lenguaje funciona de manera similar como lo haría Python, es decir, no se necesita algún termino que indique que la línea termino, el programa lo entendera. (“;”)

Análisis léxico.

```
12  ID    [a-zA-Z0-9]+
13  STRING '[a-zA-Z0-9_]+'
14
15  IF be|Be
16  ELSE sickin|Sickin
17  %%
18
19  "=\.="                      { return IGUAL;}
20  "\+_\+"                     { return SUMA; }
21  "-_-"                       { return RESTA; }
22  "\*-\*"                     { return MULTIPLICACION; }
23  "\/-\\"                     { return DIVISION; }
24
25  ">.>"   { return GREATER; }
26  "<.<"   { return LESSER; }
27  "=>.>"   { return GTEQ; }
28  "<.<="   { return LSEQ; }
29  "=_="   { return EQ; }
30
31
32  {IF} { return IF; }
33  {ELSE} { return ELSE; }
34  "be sickin" {return ELIF; }
35  "(" {return LPAR;}
36  ")" {return RPAR;}
37  "{" {return LBRAC;}
38  "}" {return RBRAC;}
39
40  ";" {return SEMICOLON;}
41  "for" {return FOR;}
42  "while" {return WHILE;}
43  "+" {return PLUS;}
44
45  "def" {return FUNCTION;}
46  "return" {return RETURN;}
47  "call" {return CALL;}
48  "," {return COMA;}
49
50  "imprimir"     { return IMPRIMIR; }
```

Para el análisis léxico, el lenguaje que se proporciona, en este caso, mediante el archivo de texto se recorren todas las líneas buscando las distintas interpretaciones que se definen en el código anterior. Para luego ir asignándolos como **tokens** para que pueda leerlos el procesador sintactico (bison). En caso de no estar definido algún texto, este arrojaría un error.

Análisis Sintactico

Para el análisis sintactico en esta fase, los **tokens** generados por el lexer se pasan a un analizador sintáctico (parser), para ir interpretándolos acorde a las reglas establecidas, reduciendo o saltando, como un árbol.

```
PROGRAM:
    /* vacio */
    | PROGRAM EXPRESSION
    ;

EXPRESSION:
    OPERATION {
        if (get_current_execution_context()) {

        }
    }
    | LOGIC {
        if (get_current_execution_context()) {

        }
    }
    | ASSIGNMENT
    | CONDITIONAL
    | LOOP
    | FUNCTION_DECLARATION
    | FUNCTION_CALL
    | IMPRIMIR LPAR OPERATION RPAR  { imprimir((void*)&$3, 0); }
    | IMPRIMIR LPAR TEXTO RPAR  { imprimir((void*)&$3, 1); }
    ;
```

Análisis semántico

Esta es la fase del compilador en la que se verifican y validan las relaciones lógicas y contextuales entre los componentes del programa, después de que la sintaxis ha sido validada. El propósito de esta fase es asegurar que el programa sea semánticamente correcto en función de las reglas del lenguaje y el contexto. Esto incluye verificar que las variables estén declaradas antes de ser utilizadas, que las operaciones sean válidas para los tipos de datos involucrados, y que las llamadas a funciones se hagan con el número correcto de parámetros.

```c
struct Variable variables[MAX_VAR];
int var_count = 0;

struct Function functions[MAX_FUNC];
int func_count = 0;

int return_stack[MAX_STACK];
int execution_stack[MAX_STACK];
int stack_top = -1;

/* functions */

void some_predefined_function(int params[]) {
    printf("Ejecutando función con parámetros: ");
    for (int i = 0; i < MAX_PARAMS && params[i] != 0; i++) {
        printf("%d ", params[i]);
    }
    printf("\n");
}


int is_variable_declared(char *name) {
    for (int i = 0; i < var_count; i++) {
        if (strcmp(variables[i].name, name) == 0) {
            return 1;
        }
    }
    return 0;
}
```

Características propias del compilador

Las características propias de este compilador, sería que la idea principal para programar era usando “caras” hechas a partir de la simbología correspondiente, además que para los condicionales usar una lengua indígena, en este caso, Ckunza. La cual se define a continuación:

Be: Significa Si

Sickin: Significa No, aunque en distintos diccionarios, esta puede definirse de otras dos maneras. (acknu, ackanu)

GRAMATICA

Gramática Símbolos

Para la simbología del compilador, en la siguiente tabla se detalla las equivalencias de los simbolos

símbolo	Tipo de operatoria
+_+	Suma (+)
-_-	Resta (-)
-	Multiplicacion (*)
/-	Division (/)
=.=	Asignacion (=)
=_=	Equivalencia (==)
<.<	Menor que (<)
>.>	Mayor que (>)

Tipos de variables

Las variables pueden ser enteras (**int**) o de texto (**string**). Estas se definen implícitamente dependiendo del tipo que se le asigne, por ejemplo:

var =.= 0 Seria una variable del tipo Int.

Foo =.= 'test' Seria una variable del tipo String

Por ahora no se aceptan asignación de la variable por si sola (int var).

Gramática condicional

Actualmente se tienen las condiciones IF, ELIF, ELSE.

Definición del lenguaje	Condicional
be / Be	IF
be sickin	ELSE IF
sickin / Sickin	ELSE

Ejemplo:

```
be (condición1) {
        Foo=.=1
} be sickin (condicion2){
```

```
        Foo=.=10
} sickin {
        Foo=.='texto'
}
```

Gramática Ciclos

Dentro del compilador, existen dos ciclos. El ciclo **FOR** y el ciclo **WHILE**, ambos deben tener condiciones, en ambos casos, se definen normalmente como en cualquier otro lenguaje, véase el ejemplo.

Definicion	Equivalencia
for	Ciclo FOR
while	Ciclo WHILE

Ejemplo:

Gramática Funciones

Para las funciones, estas deben definirse con la palabra clave "**def**" y para llamar la función debe usarse la palabra "**call**".

Estas funciones pueden o no retornar algo, y aceptan valor. En caso de retornar algo la palabra reservada seria "**return**" como en cualquier otro lenguaje.

```
def name(params){
        ...
        return 1
}
call name(100)
```

Palabra	Propósito
def	Definir función
call	Llamar función
return	Devolver valor de la función

Gramática funciones auxiliares

En este caso, solamente existe 1 sola función auxiliar para poder imprimir valores o operaciones. Este seria de la siguiente manera y simplemente imprime por pantalla lo que haya en el valor pasado por parámetro a la función.

Imprimir(val)

Ejemplo:

var =.= 1

Imprimir(var)

Output

‘1’