

Trabalho Final INF1022 2024.1

Lucas Toscano Pimentel Appolinário Cerqueira - 2110695

João Gabriel da Cunha Vasconcellos - 2211302

- **Introdução**

- **Informações gerais**

- Optou-se por realizar a compilação do Provol-One para C.

- **Maneira de executá-lo**

- Para executar, basta rodar o arquivo main.py. Este programa traduz todos os arquivos txt em “./inputs” e coloca os respectivos arquivos em c no diretório “./outputs”.

- **O que foi implementado**

- Parte Léxica**

- Tokens e Palavras Reservadas

- Os tokens foram definidos para palavras reservadas como INICIO, MONITOR, EXECUTE, TERMINO, ENQUANTO, FACA, FIM, IF, ELSE, ENDIF, ZERO, EVAL, entre outros. As palavras reservadas foram mapeadas para seus respectivos tokens usando um dicionário (reserved), garantindo que esses tokens sejam reconhecidos de forma distinta dos identificadores e números.

```
tokens = (  
    'INICIO', 'MONITOR', 'EXECUTE', 'TERMINO', 'ENQUANTO', 'FACA', 'FIM',  
    'ID', 'NUMBER', 'EQUALS', 'ZERO', 'PLUS', 'MULTIPLY', 'COMMA', 'LPAREN', 'RPAREN',  
    'IF', 'THEN', 'ELSE', 'EVAL', 'NEG', 'ENDIF'  
)  
  
reserved = {  
    'INICIO': 'INICIO',  
    'MONITOR': 'MONITOR',  
    'EXECUTE': 'EXECUTE',  
    'TERMINO': 'TERMINO',  
    'ENQUANTO': 'ENQUANTO',  
    'FACA': 'FACA',  
    'FIM': 'FIM',  
    'ZERO': 'ZERO',  
    'IF': 'IF',  
    'ENDIF': 'ENDIF',  
    'THEN': 'THEN',  
    'ELSE': 'ELSE',  
    'EVAL': 'EVAL',  
}
```

Expressões Regulares

Expressões regulares foram utilizadas para definir tokens como `t_EQUALS` (igual a `=`), `t_PLUS` (soma `+`), `t_NEG` (subtração `-`), `t_MULTIPLY` (multiplicação `*`), `t_LPAREN` (parêntese esquerdo `(`), `t_RPAREN` (parêntese direito `)`), e `t_COMMA` (vírgula `,`). Essas expressões regulares são associadas a funções que processam e retornam os tokens correspondentes, como `t_ID` para identificadores (ID) e `t_NUMBER` para números (NUMBER).

```
t_EQUALS = r'='  
t_PLUS = r'\+'  
t_NEG = r'\-'  
t_MULTIPLY = r'\*'  
t_LPAREN = r'\('  
t_RPAREN = r'\)'  
t_COMMA = r','
```

Funções de Tokenização

Funções específicas foram implementadas para cada tipo de token. Por exemplo, `t_ID` reconhece sequências de letras e números iniciadas por uma letra ou sublinhado, atribuindo o token ID ou uma palavra reservada correspondente se aplicável. `t_NUMBER` reconhece sequências de dígitos e converte o valor para um inteiro Python.

```
def t_ID(t):  
    r'[a-zA-Z_][a-zA-Z0-9_]*'  
    t.type = reserved.get(t.value, 'ID')  
    return t  
  
def t_NUMBER(t):  
    r'\d+'  
    t.value = int(t.value)  
    return t  
  
t_ignore = ' \t'  
  
def t_newline(t):  
    r'\n+'  
    t.lexer.lineno += len(t.value)  
  
def t_error(t):  
    print(f'Caractere ilegal '{t.value[0]}' na linha {t.lexer.lineno}')  
    t.lexer.skip(1)
```

Parte Sintática

Definição de regras

As regras foram estabelecidas para capturar a estrutura sintática dos programas em Provolone. Por exemplo, a regra programa : INICIO varlist MONITOR monitor EXECUTE cmds TERMINO define o formato geral do programa, incluindo declarações de variáveis (varlist), comandos de monitoramento (monitor), e sequências de comandos (cmds). Cada regra é associada a uma função que recebe os elementos correspondentes e constrói a representação sintática adequada em forma de código C.

```
vars_to_monitor = []

def p_programa(p):
    'programa : INICIO varlist MONITOR monitor EXECUTE cmds TERMINO'
    p[0] = f"#include <stdio.h>\n\nint main() {{\n{p[2]}\n{p[4]}\n{p[6]}\n\nreturn 0;\n}}}"

def p_monitor(p):
    '''monitor : varlist
    |          '''
    p[0] = p[1]
    # Armazena as variáveis monitoradas
    global vars_to_monitor
    qtd_monitored_vars = len(p[1].split(';')) - 1
    vars_to_monitor = vars_to_monitor[-qtd_monitored_vars:]

def p_varlist(p):
    '''varlist : ID COMMA varlist
    |          | ID'''
    print("p_varlist")
    if len(p) == 4:
        p[0] = f"int {p[1]};\n{p[3]}"
    elif len(p) == 2:
        p[0] = f"int {p[1]};"

    print("p[0] = ", p[0])
    global vars_to_monitor
    vars_to_monitor.append(p[1])
```

Precedência e Associatividade

A análise sintática também especifica a precedência e associatividade de operadores, como PLUS (adição) e MULTIPLY (multiplicação), utilizando regras como precedence = (('left', 'PLUS', 'MULTIPLY'), ('right', 'NEG')). Isso garante que as expressões matemáticas sejam avaliadas corretamente durante a geração do código C.

```
precedence = (
    ('left', 'PLUS', 'MULTIPLY'),
    ('right', 'NEG'),
)
```

Funções de Redução

Funções de redução foram definidas para interpretar cada tipo de comando em Provolone. Por exemplo, `p_cmd` define as ações para comandos de atribuição (`X = 5`), laços (`ENQUANTO X FAÇA cmds FIM`) e estruturas condicionais (`IF expr THEN cmds ELSE cmds ENDIF`). Dentro dessas funções, o código C correspondente é gerado com base nos tokens e estruturas sintáticas recebidas.

Escolha de implementação

Na implementação da análise sintática, decidiu-se seguir a gramática especificada para a linguagem Provol-One, que inclui produções para programas (programa), listas de variáveis (varlist), comandos (cmds), e estruturas condicionais (IF-THEN e IF-THEN-ELSE). As produções foram configuradas para capturar a estrutura sintática correta dos programas, garantindo que os comandos sejam interpretados adequadamente e convertidos em código C equivalente.

```
def p_cmd(p):
    # Versao nova
    '''cmd : ID EQUALS expr
        | ZERO LPAREN ID RPAREN
        | ENQUANTO ID FAÇA cmds FIM
        | IF expr THEN cmds ELSE cmds ENDIF
        | IF expr THEN cmds ENDIF
        | EVAL LPAREN expr COMMA expr COMMA cmds RPAREN
        | EVAL cmds NUMBER FIM'''

    global vars_to_monitor
    print(f"Variáveis monitoradas: {vars_to_monitor}")
    if p[2] == '=':
        p[0] = f"{p[1]} = {p[3]};"
        if(p[1] in vars_to_monitor):
            p[0] += f"\nprintf(\"\\'{p[0]}\\' => Z = %d\\n\", {p[1]});" # Adicio

    elif p[1] == 'ZERO':
        p[0] = f"{p[3]} = 0;"
        if(p[3] in vars_to_monitor):
            p[0] += f"\nprintf(\"\\'{p[0]}\\' => Z = %d\\n\", {p[3]});" # Adicio

    elif p[1] == 'ENQUANTO':
        p[0] = f"while ({p[2]} != 0) {{\n{p[4]}\n}}"
    elif p[1] == 'IF':
        if len(p) == 8:
            p[0] = f"if ({p[2]}) {{\n{p[4]}\n}} else {{\n{p[6]}\n}}"
        else:
            p[0] = f"if ({p[2]}) {{\n{p[4]}\n}}"
    elif p[1] == 'EVAL' and len(p) == 9:
        p[0] = f"for (int i = {p[3]}; i > {p[5]}; --i) {{\n{p[7]}\n}}"
    elif p[1] == 'EVAL' and len(p) == 5:
        p[0] = f"for (int i = 0; i < {p[3]}; ++i) {{\n{p[2]}\n}}"
```

Além disso, optou-se por implementar o comando EVAL sem retorno, onde a lista de comandos é executada um número específico de vezes determinado pelos parâmetros fornecidos (num1, num2). A expressão IF termina obrigatoriamente em ENDIF, assegurando que todas as estruturas condicionais sejam fechadas corretamente durante a análise sintática.

- **Gramática**

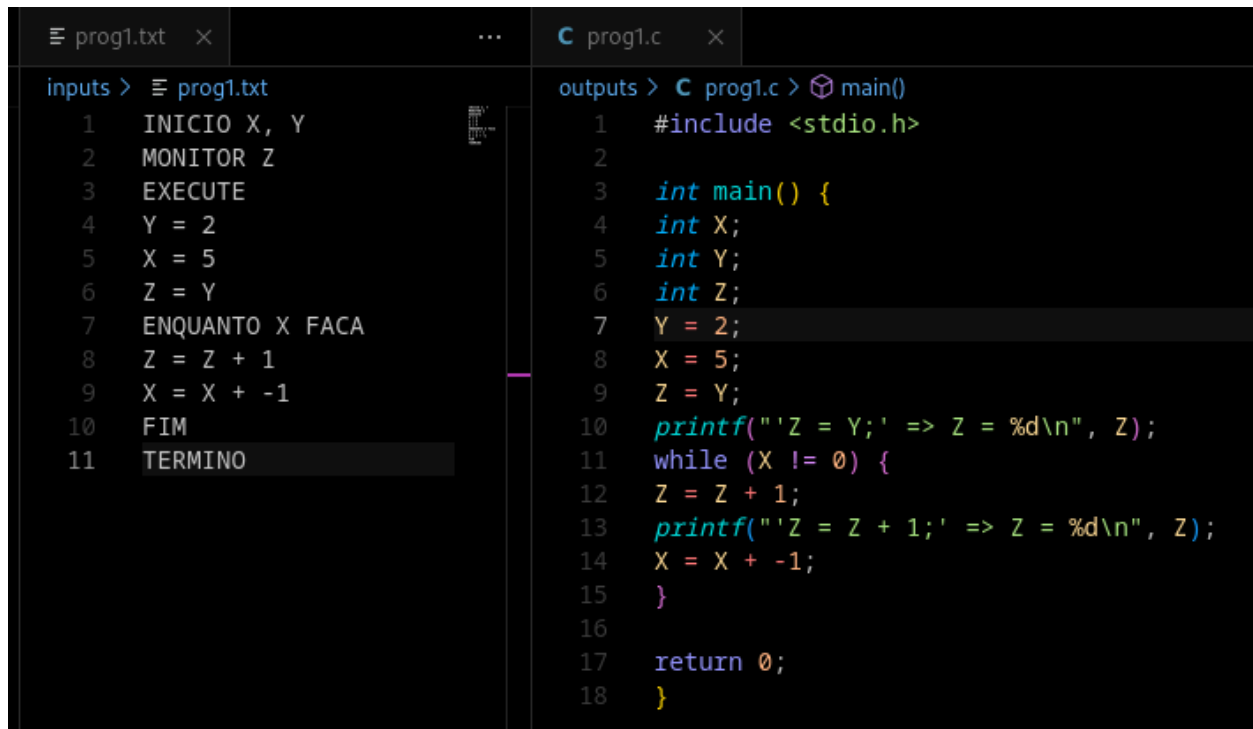
```
1  programa : INICIO varlist MONITOR monitor EXECUTE cmds TERMINO
2
3  monitor : varlist
4
5  varlist : ID ',' varlist
6  | ID
7
8  cmds : cmd cmds
9  | cmd
10
11 cmd : ID EQUALS expr
12 | ZERO LPAREN ID RPAREN
13 | ENQUANTO ID FAÇA cmds FIM
14 | IF expr THEN cmds ELSE cmds ENDIF
15 | IF expr THEN cmds ENDIF
16 | EVAL LPAREN expr COMMA expr COMMA cmds RPAREN
17 | EVAL cmds NUMBER FIM
18
19 expr : expr PLUS expr
20 | expr MULTIPLY expr
21 | LPAREN expr RPAREN
22 | NUMBER
23 | ID
24 | NEG expr %prec NEG
```

- **Testes utilizados**

Os testes foram desenvolvidos baseados nos três programas modelo disponibilizados no enunciado do trabalho. Foram realizadas algumas alterações para adequá-los às escolhas de implementação das funcionalidades e para ter uma apresentação mais clara do objetivo de cada programa.

Prog1

O programa começa declarando variáveis X e Y, e especificando que a variável Z deve ser monitorada. Em seguida, atribui valores a Y e X, copia o valor de Y para Z, e executa um laço ENQUANTO que incrementa Z e decrementa X até que X seja zero. O código inclui comandos básicos de controle de fluxo, como laços e condicionais, e realiza monitoramento de variáveis, imprimindo seus valores após certas operações.



```
inputs > prog1.txt
1  INICIO X, Y
2  MONITOR Z
3  EXECUTE
4  Y = 2
5  X = 5
6  Z = Y
7  ENQUANTO X FACA
8  Z = Z + 1
9  X = X + -1
10 FIM
11 TERMINO

outputs > C prog1.c > main()
1  #include <stdio.h>
2
3  int main() {
4  int X;
5  int Y;
6  int Z;
7  Y = 2;
8  X = 5;
9  Z = Y;
10 printf("'Z = Y;' => Z = %d\n", Z);
11 while (X != 0) {
12 Z = Z + 1;
13 printf("'Z = Z + 1;' => Z = %d\n", Z);
14 X = X + -1;
15 }
16
17 return 0;
18 }
```

O resultado gerado está correto, conforme evidenciado pelas saídas do programa C. Após a inicialização de Z com o valor de Y (que é 2), o programa entra no laço ENQUANTO, que decrementa X até que ele seja igual a 0. Durante cada iteração do laço, Z é incrementado por 1, conforme indicado pelas impressões (printf). Essas saídas confirmam que a lógica de atribuição inicial e o incremento de Z foram traduzidos e executados corretamente, seguindo a intenção do código original em Provol-One.

```
'Z = Y;' => Z = 2
'Z = Z + 1;' => Z = 3
'Z = Z + 1;' => Z = 4
'Z = Z + 1;' => Z = 5
'Z = Z + 1;' => Z = 6
'Z = Z + 1;' => Z = 7
```

Prog2

O programa começa declarando as variáveis X, Y e B, e especifica que a variável Z deve ser monitorada. As variáveis X e Y recebem valores iniciais de 7 e 2, respectivamente, enquanto B e Z são inicializadas com 0. O código inclui um laço EVAL que itera de X até Y, e dentro do laço, há uma condição IF que verifica o valor de B. Dependendo do valor de B, Z é incrementado por 1 ou 2, e B alterna entre 0 e 1.

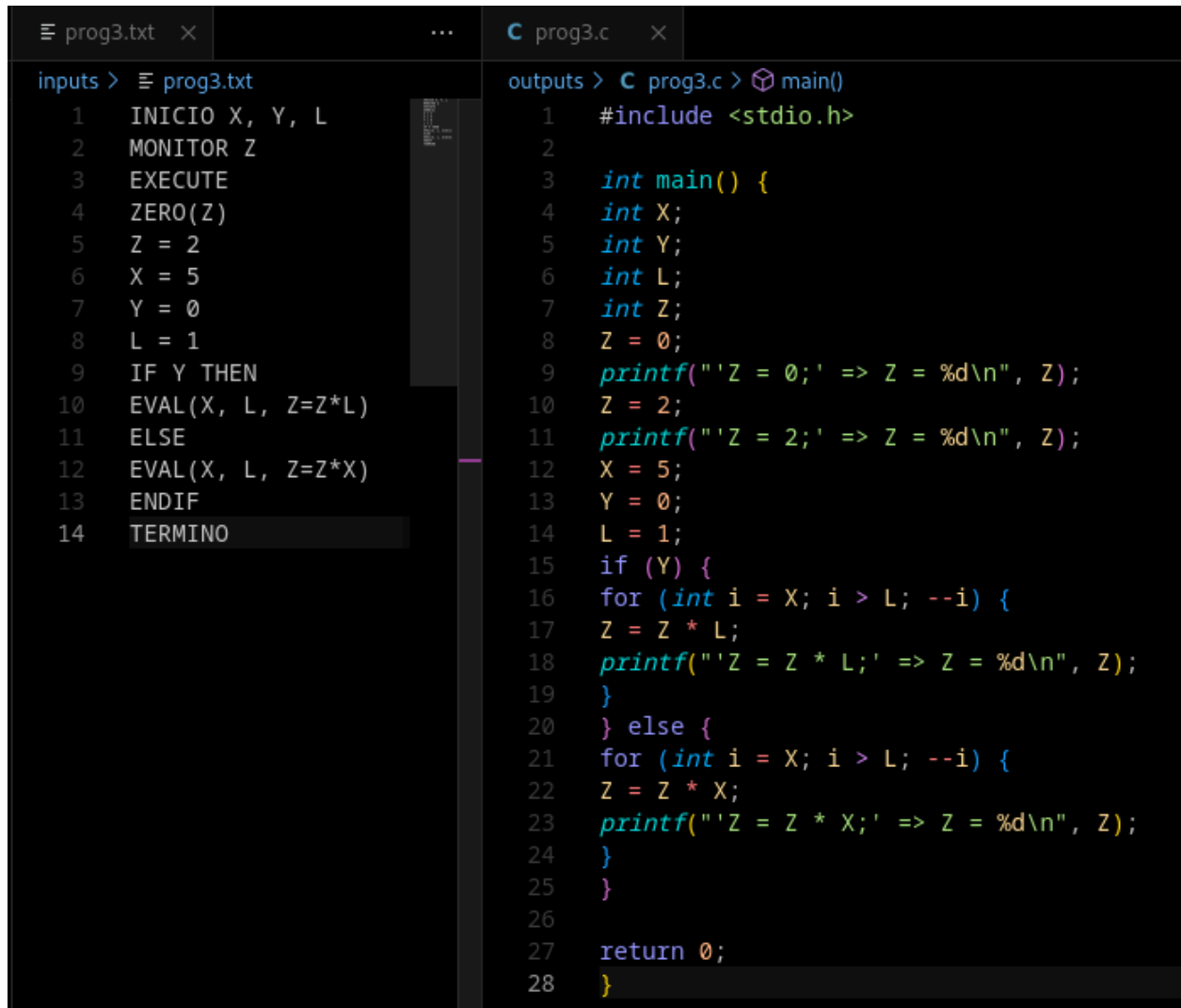
prog2.txt	prog2.c
<pre>inputs > prog2.txt 1 INICIO X, Y, B 2 MONITOR Z 3 EXECUTE 4 X = 7 5 Y = 2 6 B = 0 7 Z = 0 8 EVAL(X, Y, 9 IF B THEN Z = Z + 2 10 B = 0 11 ELSE Z = Z + 1 12 B = 1 13 ENDIF) 14 TERMINO</pre>	<pre>outputs > C prog2.c > main() 1 #include <stdio.h> 2 3 int main() { 4 int X; 5 int Y; 6 int B; 7 int Z; 8 X = 7; 9 Y = 2; 10 B = 0; 11 Z = 0; 12 printf("'Z = 0;' => Z = %d\n", Z); 13 for (int i = X; i > Y; --i) { 14 if (B) { 15 Z = Z + 2; 16 printf("'Z = Z + 2;' => Z = %d\n", Z); 17 B = 0; 18 } else { 19 Z = Z + 1; 20 printf("'Z = Z + 1;' => Z = %d\n", Z); 21 B = 1; 22 } 23 } 24 25 return 0; 26 }</pre>

O resultado gerado está correto, conforme evidenciado pelas saídas do programa C. Após a inicialização de Z com 0, o programa entra no laço for e executa as operações de incremento de Z e alternância de B conforme esperado. As impressões (printf) mostram claramente o valor de Z em cada etapa: Z começa em 0, é incrementado para 1, depois para 3, seguido por 4, 6, e finalmente 7. Essas saídas confirmam que a lógica condicional e o incremento de Z foram traduzidos e executados corretamente, preservando a intenção do código original em Provol-One.

```
'Z = 0;' => Z = 0  
'Z = Z + 1;' => Z = 1  
'Z = Z + 2;' => Z = 3  
'Z = Z + 1;' => Z = 4  
'Z = Z + 2;' => Z = 6  
'Z = Z + 1;' => Z = 7
```


Prog3

O programa começa declarando as variáveis X, Y e L, e especifica que a variável Z deve ser monitorada. A variável Z é inicializada com zero, e em seguida recebe o valor 2. As variáveis X, Y e L recebem valores iniciais de 5, 0 e 1, respectivamente. O código inclui uma estrutura condicional IF-ELSE que verifica o valor de Y. Como Y é zero, o ramo ELSE é executado, e um laço FOR é usado para multiplicar Z por X em cada iteração, conforme especificado pelo comando EVAL.



```
inputs > prog3.txt
1 INICIO X, Y, L
2 MONITOR Z
3 EXECUTE
4 ZERO(Z)
5 Z = 2
6 X = 5
7 Y = 0
8 L = 1
9 IF Y THEN
10 EVAL(X, L, Z=Z*L)
11 ELSE
12 EVAL(X, L, Z=Z*X)
13 ENDIF
14 TERMINO

outputs > C prog3.c > main()
1 #include <stdio.h>
2
3 int main() {
4 int X;
5 int Y;
6 int L;
7 int Z;
8 Z = 0;
9 printf("'Z = 0;' => Z = %d\n", Z);
10 Z = 2;
11 printf("'Z = 2;' => Z = %d\n", Z);
12 X = 5;
13 Y = 0;
14 L = 1;
15 if (Y) {
16 for (int i = X; i > L; --i) {
17 Z = Z * L;
18 printf("'Z = Z * L;' => Z = %d\n", Z);
19 }
20 } else {
21 for (int i = X; i > L; --i) {
22 Z = Z * X;
23 printf("'Z = Z * X;' => Z = %d\n", Z);
24 }
25 }
26
27 return 0;
28 }
```

O resultado gerado está correto, conforme evidenciado pelas saídas do programa C. Após a inicialização de Z com 0 e sua atualização para 2, o programa entra na estrutura condicional if-else. Como Y é igual a 0, o segundo ramo (else) é seguido, executando o laço for que multiplica Z pelo valor de X em cada iteração. As impressões (printf) mostram claramente o valor de Z em cada passo do laço: Z começa em 0, é atualizado para 2, e depois é multiplicado por 5 sucessivamente, resultando em 10, 50,

250, e finalmente 1250. Essas saídas confirmam que a lógica condicional e o cálculo de Z foram traduzidos e executados corretamente, seguindo a intenção do código original em Provol-One.

```
'Z = 0;' => Z = 0  
'Z = 2;' => Z = 2  
'Z = Z * X;' => Z = 10  
'Z = Z * X;' => Z = 50  
'Z = Z * X;' => Z = 250  
'Z = Z * X;' => Z = 1250
```