CONSTRUÇÃO DE UM COMPILADOR PARA PASCAL STANDARD

Relatório Técnico Final

Unidade Curricular: Processamento de Linguagens e Compiladores

Realizado por:
Francisco Bonjardim Dias (A108561)
Pedro Rodrigues Campos (A108482)
Rodrigo Gonçalves Rocha (A108649)

1. Introdução

Este projeto teve como objetivo principal o desenvolvimento de um compilador para a linguagem Pascal Standard, capaz de analisar, interpretar e traduzir código Pascal para código máquina.

O compilador foi desenhado para suporar as princípais características da linguagem Pascal Standard, incluindo:

- Tipos de dados básicos (`Integer`, `Real`, `Boolean`,`String`) e estruturados (`Arrays`).
- Estruturas de controlo de fluxo (`IF`, `WHILE`, `FOR`, `REPEAT`).
- Subprogramas (`Functions` e `Procedures`).
- Distinção entre variáveis globais e locais.

O processo de compilação segue a seguinte estrutura:

Lexer → Parser → Semântica → Gerador de código máquina

2. Análise Léxica (`lexer.py`)

O analisador léxico tem como objetivo transformar o input recebido (código Pascal) numa série de tokens significativos. Utilizámos o módulo `ply.lex`.

2.1 Tratamento de palavras reservadas

Umas das principais implementações que fizemos no nosso lexer foi a utilização de um dicionário de palavras reservadas, para distinguir as palavras-chaves da linguagem (como `if`, `while`, `program`).

Esta decisão torna mais fácil a identificação destas palavras-chaves, e evita conflitos nas regras regex. Além disso, permite simplificar a lógica de verificação.

A estratégia que optámos foi a seguinte:

1. Definimos uma regra abrangente (`t_ID`) que captura qualquer sequência alfanumérica.
2. Dentro desta regra, verifica se a palavra capturada existe no dicionário `reserved`.
3. Se existir, o valor correspondente no dicionário é atribuído ao token desta palavra. Caso contrário, o token será `ID`.

```
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value.lower(), 'ID')
    return t
```

2.2 Outras decisões na implementação

- **Comentários:** São suportados dois tipos de comentários, o que garante que o compilador ignora os comentários corretamente:
  - Comentários de bloco: `{ ... }` ou `(* ... *)`.
  - Comentários de linha: `// ...`.

```
def t_COMMENT(t):
    # { [LE TUDO QUE NAO SEJA "}", repete até encontrar "}"
    # (* [QQ CHAR OU \n]*? -> repete até se encontrar a sequencia final *)
    r'\{[^}]*\} | \(\*(.|\n)*?\*\)'
     #Temos que fazer lazy, porque se nao alguns comentarios comia
demasiado. Assim para logo que encontre o primeiro fecho
    pass

def t_COMMENT_LINE(t):
    r'//.*'
    pass
```

- **Literais:** Os diferentes tipos de dados básicos (Inteiros, Reais e Strings) são distintos:

```
def t_REAL_VAL(t):
    r'\d+\.\d+'
    t.value = float(t.value)
    return t

def t_INT_VAL(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_STRING_LITERAL(t):
    r'\'([^\']| \'\')*\''
    t.value = t.value[1:-1]
    return t
```

3. Análise sintática (`parser.py`)

O analisador sintático valida a estrutura gramatical do programa. Utilizámos `ply.yacc`, que implementa um parser bottom-up.

Este parser constrói uma **Árvore de Sintaxe Abstrata (AST)**, que é representada por tuplos (ex: `('BINOP', '+', esq, dir)`), o que facilita a travessia da árvore nas próximas fases do projeto.

3.1 Estrutura geral do programa

Um programa Pascal é composto por um cabeçalho, declaração de variáveis globais, subprogramas e o bloco principal de execução.

```
def p_program(p):
    r'program : PROGRAM ID ";" vars lista_subprogramas conteudo "."'
    # ('PROGRAM', ID, vars, subprogramas,bloco_main)
    p[0] = ('PROGRAM', p[2], p[4], p[5], p[6])
```

Esta estrutura permite organizar o programa:

1. **Cabeçalho (`PROGRAM ID`):** Identificação do programa
2. **Variáveis Globais (`vars`):** Definição das variáveis globais.
3. **Subprogramas (`lista_subprogramas`):** Lista recursiva que aceita zero ou mais declarações de `FUNCTION` ou `PROCEDURE`.
4. **Conteúdo (`conteudo`):** Bloco principal de execução, delimitado por `BEGIN ... END`, que contém a lista de comandos a executar.

3.2 Declaração de variáveis (`vars`)

A secção de variáveis, iniciada pela palavra-chave `VAR`, permite declarar múltiplas variáveis do mesmo tipo numa única linha (ex: `x,y : integer;`).

O parser processa linha a linha e gera nós do tipo `DECL_VAR`, que agrupam a lista dos identificadores e o seu tipo.

```
def p_decl_var(p):
    r'decl_var : lista_ids ":" tipo ";"'
    # ('DECL_VAR', lista_ids, tipo) -> ('DECL_VAR', (['x', 'y'],
'INTEGER')
    p[0] = ('DECL_VAR', p[1], p[3])
```

3.3 Definição de Subprogramas

O compilador suporta tanto `FUNCTION` (tem valor de retorno), como `PROCEDURE` (sem valor de retorno). A gramática trata a lista de subprogramas de forma recursiva, e permite zero ou mais definições de subprogramas.

```
def p_decl_function(p):
    r'decl_function : FUNCTION ID "(" args_opt ")" ":" tipo ";" vars
conteudo ";"'
    p[0] = ('FUNCTION',p[2],p[4],p[7],p[9],p[10])

def p_decl_procedure(p):
    r'decl_procedure : PROCEDURE ID "(" args_opt ")" ";" vars conteudo
";"'
    p[0] = ('PROCEDURE', p[2],p[4],p[7],p[8])
```

3.4 Comandos

O corpo do programa (`conteudo`) é definido por um bloco `BEGIN ... END`, que contém uma lista de comandos. Os comandos suportados incluem atribuições, chamadas de procedimentos, ciclos, condicionais e INPUT/OUTPUT (`READLN/WRITELN`).

Atribuições

- A regra distingue entre atribuição a variáveis simples e posições de arrays.

```
def p_assignment(p):
    r'assignment : variavel ASSIGN exp'
    p[0] = ('ASSIGN', p[1],p[3])

def p_variavel_single(p):
    r'variavel : ID'
    p[0] = ('ID',p[1])

def p_variavel_array(p):
    r'variavel : ID "[" exp "]"'
    p[0] = ('ARRAY_ACESSO',p[1],p[3])
```

Chamadas de procedimentos

- Ao contrário das funções (que ocorrem dentro de expressões), os procedimentos são tratados como comandos independentes. A gramática captura o ID e a lista de argumentos (que pode ser vazia).

```
def p_procedure_call(p):
    r'procedure_call : ID "(" args_subp ")"'
    p[0] = ('PROC_CALL', p[1],p[3])

def p_args_subp_lista(p):
    r'args_subp : lista_exp'
    p[0] = p[1]

def p_args_subp_vazio(p):
```

```
    r'args_subp : '
    p[0] = []
```

Ciclos

- A gramática suporta as três principais estruturas de ciclos do Pascal Standard:

```
def p_while_comando(p):
    r'while_comando : WHILE exp DO comando'
    p[0] = ('WHILE',p[2],p[4])

def p_for_comando_to(p):
    r'for_comando : FOR ID ASSIGN exp TO exp DO comando'
    p[0] = ('FOR',p[2],p[4],p[6],p[8],'to')

def p_for_comando_downto(p):
    r'for_comando : FOR ID ASSIGN exp DOWNTO exp DO comando'
    p[0] = ('FOR',p[2],p[4],p[6],p[8],'downto')

def p_repeat_comando(p):
    r'repeat_comando : REPEAT lista_comandos UNTIL exp'
    p[0] = ('REPEAT',p[2],p[4])
```

Condicionais

A gramática suporta as duas variações do `IF`: com e sem o ramo `ELSE`, tendo assim duas regras distintas.

```
def p_if_comando_no_else(p):
    r'if_comando : IF exp THEN comando'
    p[0] = ('IF',p[2],p[4],None)

def p_if_comando_else(p):
    r'if_comando : IF exp THEN comando ELSE comando'
    p[0] = ('IF',p[2],p[4],p[6])
```

NOTA: A ambiguidade que pode ocorrer em `IF`'s alinhados é resolvida pela precedência shift-preference do PLY. Ou seja, o `ELSE` é associado ao `IF` mais próximo.

INPUT/OUTPUT

A gramática suporta as diferentes variações dos comandos de input e output `READLN` e `WRITELN`. Podem receber argumentos, ou não receber nada.

```
def p_io_comando_writeln_exp(p):
    r'io_comando : WRITELN "(" lista_exp ")"'
    p[0] = ('WRITELN',p[3])
```

```
def p_io_comando_writeln_vazio(p):
    r'io_comando : WRITELN'
    p[0] = ('WRITELN',[])

def p_io_comando_writeln_vazio2(p):
    r'io_comando : WRITELN "(" ")"'
    p[0] = ('WRITELN',[])

def p_io_comando_readln_var(p):
    r'io_comando : READLN "(" variavel ")"'
    p[0] = ('READLN',p[3])

def p_io_comando_readln_vazio(p):
    r'io_comando : READLN'
    p[0] = ('READLN',[])

def p_io_comando_readln_vazio2(p):
    r'io_comando : READLN "(" ")"'
    p[0] = ('READLN',[])
```

3.5 Expressões e Precedência dos operadores

Para resolver a precedência matemática, estruturámos a gramática das expressões em **4 níveis hierárquicos**. Isto força o parser a resolver operações prioritárias (como a multiplicação), antes de subir para as somas e comparações.

- **Nível 1 (`exp`):** Comparadores (`=`, `<>`, `<`, `>`, `<=`, `>=`):

```
def p_exp_eq(p):
    r'exp : exp_simples "=" exp_simples'
    p[0] = ('BINOP', "=", p[1],p[3])

def p_exp_neq(p):
    r'exp : exp_simples NEQ exp_simples'
    p[0] = ('BINOP', "<>", p[1],p[3])

def p_exp_lt(p):
    r'exp : exp_simples "<" exp_simples'
    p[0] = ('BINOP', "<", p[1],p[3])

def p_exp_gt(p):
    r'exp : exp_simples ">" exp_simples'
    p[0] = ('BINOP', ">", p[1],p[3])

def p_exp_le(p):
    r'exp : exp_simples LE exp_simples'
    p[0] = ('BINOP', "<=", p[1],p[3])

def p_exp_ge(p):
```

```
    r'exp : exp_simples GE exp_simples'
    p[0] = ('BINOP', ">=", p[1],p[3])

def p_exp_single(p):
    r'exp : exp_simples'
    p[0] = p[1]
```

- **Nível 2 (`exp_simples`):** Operações de soma (`+`), subtração (`-`) e disjunção (`OR`):

```
def p_exp_simples_add(p):
    r'exp_simples : exp_simples "+" term'
    p[0] = ('BINOP','+',p[1],p[3])

def p_exp_simples_minus(p):
    r'exp_simples : exp_simples "-" term'
    p[0] = ('BINOP','-',p[1],p[3])

def p_exp_simples_or(p):
    r'exp_simples : exp_simples OR term'
    p[0] = ('BINOP','OR',p[1],p[3])

def p_exp_simples_term(p):
    r'exp_simples : term'
    p[0] = p[1]
```

- **Nível 3 (`term`):** Operações de multiplicação (`*`), divisão(`/` e `DIV`), módulo (`MOD`) e conjunção (`AND`):

```
def p_term_mult(p):
    r'term : term "*" factor'
    p[0] = ('BINOP', '*', p[1],p[3])

def p_term_dividir(p):
    r'term : term "/" factor'
    p[0] = ('BINOP', '/', p[1],p[3])

def p_term_div_int(p):
    r'term : term DIV factor'
    p[0] = ('BINOP', 'DIV', p[1],p[3])

def p_term_mod(p):
    r'term : term MOD factor'
    p[0] = ('BINOP', 'MOD', p[1],p[3])

def p_term_and(p):
    r'term : term AND factor'
    p[0] = ('BINOP', 'AND', p[1],p[3])

def p_term_factor(p):
```

```
    r'term : factor'
    p[0] = p[1]
```

- **Nível 4 (`factor`):** Literais, variáveis, parênteses, negação (`NOT`), menos unário (`-`), a chamada da função pré-definida `LENGTH` e a chamada de `FUNCTION`'s.

```
def p_factor_variavel(p):
    r'factor : variavel'
    p[0] = p[1]

def p_factor_int(p):
    r'factor : INT_VAL'
    p[0] = ('INTEGER',p[1])

def p_factor_real(p):
    r'factor : REAL_VAL'
    p[0] = ('REAL',p[1])

def p_factor_string(p):
    r'factor : STRING_LITERAL'
    p[0] = ('STRING',p[1])

def p_factor_true(p):
    r'factor : TRUE'
    p[0] = ('BOOLEAN',True)

def p_factor_false(p):
    r'factor : FALSE'
    p[0] = ('BOOLEAN',False)

def p_factor_exp(p):
    r'factor : "(" exp ")"'
    p[0] = p[2]

def p_factor_not(p):
    r'factor : NOT factor'
    p[0] = ('UNOP', 'NOT', p[2])

def p_factor_uminus(p):
    r'factor : "-" factor'
    p[0] = ('UNOP', '-', p[2])

def p_factor_length(p):
    r'factor : LENGTH "(" exp ")"'
    p[0] = ('LENGTH',p[3])

def p_factor_call(p):
    r'factor : function_call'
    p[0] = p[1]

def p_function_call(p):
```

```
    r'function_call : ID "(" args_subp ")"'
    p[0] = ('FUNC_CALL',p[1],p[3])
```

Esta estrutura garante que as prioridades nas operações é respeitada. Por exemplo, na expressão `2 + Soma(a,b) * 4`, o parser reduz primeiro a multiplicação `Soma(a,b) * 4`.

4. Análise Semântica (`semantica.py`)

Está é a fase onde são validadas as regras da linguagem. Implementámos a classe `Semantica` que percorre a AST e utiliza uma Tabela de Símbolos.

A Tabela de Símbolos é uma lista de dicionários (`self.tabela = [{}]`), que funciona como uma pilha. Quando entra numa função, uma nova tabela é empilhadada (para as variáveis locais), e ao dar `return` é removida.

4.1 Regras Implementadas

ASSIGN

Ao tentar dar ASSIGN, valida se o tipo de expressão à direita é compatível com a variável da esquerda.

- Não é possível atribuir `INTEGER` a `BOOLEAN`/`STRING` ou `STRING` a `INTEGER` (por exemplo)
- A única excessão é que é permitido atribuir um `INTEGER` a uma variável `REAL` (mas não o contrário), para seguir o comportamento do Pascal.

Validar subprogramas

Na chamada de funções e procedimentos é preciso validar o seguinte:

- O número de argumentos passados na chamada deve ser igual ao número de parâmetros declarados;
- Os argumentos recebidos e os esperados devem corresponder ao tipo esperado (a posição interessa);

Comandos

- Para estruturas do tipo `IF`, `WHILE` e `REPEAT` é imposto que a expressão da condição seja estritamente `BOOLEAN`.
- Para estruturas do tipo `FOR` é necessário que o intervalo de range dado seja válido.
- `READLN` precisa que os argumentos passados sejam variáveis.
- `LENGTH` permite calcular tamanho de `STRING` e de `ARRAY`.

Acesso a Arrays

- O índice deve ser do tipo `INTEGER`.
- Se o índice for um valor literal (ex : `v[5]`), o compilador verifica se está dentro do intervalo declarado.

Operações binárias

Na visita a nós `BINOP`, verifica os tipos dos operandos (`esq`, `dir`) e o operador em questão:

- **Aritmética:** Ambos os operandos devem ser `INTEGER` ou `REAL`;
- **Concatenação:** É possível concatenar (`+`) `STRING`'s;
- **Lógica (`AND`/`OR`):** Exige que ambos os operandos sejam `BOOLEAN`;

- **Comparações:** É possível comparar números ou tipos iguais.

5. Gerar Código Máquina (`codigo.py`)

A fase final do compilador é a tradução da AST para a linguagem da Máquina Virtual (VM). A classe `Codigo` é responsável por esta tradução. Voltámos a usar uma tabela de variáveis, com a adição de um `offset`, para poder aceder às variáveis na memória.

5.1 Memória e gestão da pilha

Foi necessário implementar uma gestão das variáveis na pilha, para podermos aceder facilmente. As variáveis podem ser globais ou locais, por isso foi necessário utilizar dois contadores distintos, que calculam o seu `offset`.

1. **Global Pointer (`gp_offset`):** Para variáveis globais, declaradas no bloco principal.
2. **Frame Pointer (`fp_offset`):** Para variáveis locais dentro dos subprogramas.

```
def add_var(self, nome, tipo, categoria):
        # CATEGORIA -> GLOBAL OU LOCAL
        if categoria == 'GLOBAL':
            offset = self.gp_offset
            self.gp_offset += 1
        else:
            offset = self.fp_offset
            self.fp_offset += 1

        self.tabela[-1][nome] = (offset, tipo, categoria)
        return offset
```

Registámos juntamente com a variável, o seu offset e o tipo (global ou local).

5.2 Gestão de Subprogramas

Para declarar funções e procedimentos, utilizámos a seguinte estrutura:

- **Argumentos:** Têm offset negativo, pois foram empilhados antes da chamada da função (`CALL`).
- **Variáveis Locais:** Têm offset positivo, são alocados após o `CALL`.
- No caso das funções: É reservado um espaço na pilha antes dos argumentos para o valor de retorno.

```
def visita_FUNCTION(self, node):
    nome = node[1]
    args = node[2]

    self.emit(f"F{nome}:")
    self.append_tabela()
    self.fp_offset = 0

    lista_args = []
    for grupo in args:
        ids = grupo[0]
```

```
            tipo = grupo[1]
            for nome_arg in ids:
                lista_args.append((nome_arg, tipo))

        num_args = len(lista_args)

        for i, (nome_arg, tipo_arg) in enumerate(lista_args):
            offset = -(num_args - i)
            self.add_arg(nome_arg, tipo_arg, offset)

        self.add_var(nome, node[3], 'LOCAL')
        offset_retorno = self.get_var(nome)[0]

        self.visita(node[4])        #visita as vars locais, incrementa o FP
        if self.fp_offset > 0:
            self.emit(f"PUSHN {self.fp_offset}")

        self.visita(node[5])

        self.emit(f"PUSHL {offset_retorno}") # return value
        offset_retorno_slot = -(num_args + 1)
        self.emit(f"STOREL {offset_retorno_slot}")

        self.emit("RETURN")
        self.pop_tabela()
```

Chamada:

Na chamada de uma função (`visita_FUNC_CALL`), reservámos a variável de retorno `PUSHI 0`, emitimos `PUSHA` seguido de `CALL`. Após o retorno, a instrução `POP` limpa os argumentos da pilha, e deixa apenas o valor de retorno. Em caso de chamadas de procedures, o procedimento é o mesmo, exceto reservar a variável de retorno, pois este não existe.

```
def visita_FUNC_CALL(self, node):
    nome = node[1]
    args = node[2]

    self.emit("PUSHI 0") # Espaço para return
    for arg in args:
        self.visita(arg)

    self.emit(f"PUSHA F{nome}")
    self.emit("CALL")

    if len(args) > 0:
        self.emit(f"POP {len(args)}")

    return 'INTEGER'
```

```
Diagrama da VM ao chamar Soma := Soma(a,b)
-----------------------------------------
|  ...              |
|  Local Y          |   Offset +2    (Var Local extra)
|  Local X          |   Offset +1    (Var Local extra)
-----------------------------------------
|  VAR RETORNO      |   Offset  0    <-- FRAME POINTER (FP) aponta aqui!
-----------------------------------------
|  Argumento B      |   Offset -1    (Último arg. a entrar)
-----------------------------------------
|  Argumento A      |   Offset -2    (Primeiro arg. a entrar)
-----------------------------------------
|  SLOT RESULTADO   |   Offset -3    (Onde o Caller espera o valor final) -
> Posicao -(N+1), N = n_args
-----------------------------------------
```

5.3 Controlos de fluxo

Como os comandos `IF`, `WHILE`, `FOR` e `REPEAT` não existem na VM, temos de traduzir para sequências de testes e saltos (`JUMP` e `JZ`). Para isso, implementámos um sistema de etiquetas únicas (`L1`, `L2`,...).

EXEMPLO: Num comando `IF-THEN-ELSE`, o compilador gera o código da condição, seguido de um salto (`JZ`) para o bloco `ELSE`.

```
def visita_IF(self, node):
    l_else = self.new_label()
    l_fim = self.new_label()

    self.visita(node[1]) #Codigo da condiçao
    self.emit(f"JZ {l_else}")  #Se Falso, salta para o Else

    self.visita(node[2]) #Bloco THEN
    self.emit(f"JUMP {l_fim}") #Salta o ELSE

    self.emit(f"{l_else}:") #Etiqueta do else

    if node[3]: #Se existir else
        self.visita(node[3])
    self.emit(f"{l_fim}:") #FIM DO IF
```

5.4 Acesso a Arrays

O acesso a arrays exige o cálculo do endereço do indice. Como o pascal permite índices arbitrários (ex : `10..20`), o endereço físico é calculo como: `Endereço = Base + (Indice - LimiteInferior)`

O compilador emite instruções para carregar a base do array (`PUSHG/PUSHL`), calcula o índice, subtrai o limite inferior e acede à memória.

```
def visita_ARRAY_ACESSO(self, node):
    nome = node[1]
    indice = node[2]
    info = self.get_var(nome)
    #(offset, tipo, categoria)
    # 1. Carregar endereço do array ou string
    if info[2] == 'GLOBAL':
        self.emit(f"PUSHG {info[0]}")
    else:
        self.emit(f"PUSHL {info[0]}")

    # 2. Se for STRING, usa lógica diferente (CHARAT)
    if info[1] == 'STRING':
        self.visita(indice)
        self.emit("PUSHI 1") # Pascal index 1-based
        self.emit("SUB")     # VM index 0-based
        self.emit("CHARAT")  # Devolve ASCII code
        return 'INTEGER'     # ASCII é inteiro
    #EX : nome := 'MARIA'; letra := nome[2]
    # 3. Se for Array normal
    self.visita(indice)
    range_min = info[1][1][0]
    if range_min != 0:
        self.emit(f"PUSHI {range_min}")
        self.emit("SUB")
    self.emit("LOADN")
    return info[1][2]
```

O acesso a chars individuais numa String requer lógica dedicada porque, ao contrário dos arrays estáticos, as strings são objetos geridos pela VM. Utilizamos a instrução CHARAT para extrair o código ASCII do char. Adicionalmente, é necessário subtrair 1 ao índice fornecido pelo programador, visto que em Pascal os indices das strings começam em 1, e na VM começam em 0.

5.5 Restantes escolhas

As restantes operações são bastantes direntas. As operações binárias todas estão definidas na VM. O único pormenor é na operação unária (`-`), em que multiplicámos o número por `-1`.

6. Testes

6.1 Olá, Mundo!

INPUT

```
program HelloWorld;
begin
writeln('Ola, Mundo!');
end.
```

OUTPUT

```
START
JUMP main
main:
PUSHS "Ola, Mundo!"
WRITES
WRITELN
STOP
```

6.2 Fatorial

INPUT

```
program Fatorial;
var
n, i, fat: integer;
begin
writeln('Introduza um número inteiro positivo:');
readln(n);
fat := 1;
for i := 1 to n do
fat := fat * i;
writeln('Fatorial de ', n, ': ', fat);
end.
```

OUTPUT

```
PUSHN 3
START
JUMP main
main:
PUSHS "Introduza um número inteiro positivo:"
WRITES
```

```
WRITELN
READ
ATOI
STOREG 0
PUSHI 1
STOREG 2
PUSHI 1
STOREG 1
L1:
PUSHG 1
PUSHG 0
INFEQ
JZ L2
PUSHG 2
PUSHG 1
MUL
STOREG 2
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP L1
L2:
PUSHS "Fatorial de "
WRITES
PUSHG 0
WRITEI
PUSHS ": "
WRITES
PUSHG 2
WRITEI
WRITELN
STOP
```

6.3 Verificação de Número Primo

INPUT

```
program NumeroPrimo;
var
num, i: integer;
primo: boolean;
begin
writeln('Introduza um número inteiro positivo:');
readln(num);
primo := true;
i := 2;
while (i <= (num div 2)) and primo do
begin
if (num mod i) = 0 then
primo := false;
```

```
i := i + 1;
end;
if primo then
writeln(num, ' é um número primo')
else
writeln(num, ' não é um número primo')
end.
```

OUTPUT

```
PUSHN 3
START
JUMP main
main:
PUSHS "Introduza um número inteiro positivo:"
WRITES
WRITELN
READ
ATOI
STOREG 0
PUSHI 1
STOREG 2
PUSHI 2
STOREG 1
L1:
PUSHG 1
PUSHG 0
PUSHI 2
DIV
INFEQ
PUSHG 2
AND
JZ L2
PUSHG 0
PUSHG 1
MOD
PUSHI 0
EQUAL
JZ L3
PUSHI 0
STOREG 2
JUMP L4
L3:
L4:
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP L1
L2:
PUSHG 2
JZ L5
```

```
PUSHG 0
WRITEI
PUSHS " é um número primo"
WRITES
WRITELN
JUMP L6
L5:
PUSHG 0
WRITEI
PUSHS " não é um número primo"
WRITES
WRITELN
L6:
STOP
```

6.4 Soma de uma lista de inteiros

INPUT

```
program SomaArray;
var
numeros: array[1..5] of integer;
i, soma: integer;
begin
soma := 0;
writeln('Introduza 5 números inteiros:');
for i := 1 to 5 do
begin
readln(numeros[i]);
soma := soma + numeros[i];
end;
writeln('A soma dos números é: ', soma);
end.
```

OUTPUT

```
PUSHN 3
PUSHI 5
ALLOCN
STOREG 0
START
JUMP main
main:
PUSHI 0
STOREG 2
PUSHS "Introduza 5 números inteiros:"
WRITES
WRITELN
```

```
PUSHI 1
STOREG 1
L1:
PUSHG 1
PUSHI 5
INFEQ
JZ L2
PUSHG 0
PUSHG 1
PUSHI 1
SUB
READ
ATOI
STOREN
PUSHG 2
PUSHG 0
PUSHG 1
PUSHI 1
SUB
LOADN
ADD
STOREG 2
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP L1
L2:
PUSHS "A soma dos números é: "
WRITES
PUSHG 2
WRITEI
WRITELN
STOP
```

6.5 Conversão binário-decimal

INPUT

```
program BinarioParaInteiro;
var
bin: string;
valor: integer;

function BinToInt(bin: string): integer;
var
i, valor, potencia: integer;
begin
valor := 0;
potencia := 1;
for i := length(bin) downto 1 do
```

```
begin
if bin[i] = '1' then
valor := valor + potencia;
potencia := potencia * 2;
end;
BinToInt := valor;
end;

begin
writeln('Introduza uma string binária:');
readln(bin);
valor := BinToInt(bin);
writeln('O valor inteiro correspondente é: ', valor);
end.
```

OUTPUT

```
PUSHN 2
START
JUMP main
FBinToInt:
PUSHN 3
PUSHI 0
STOREL 2
PUSHI 1
STOREL 3
PUSHL -1
STRLEN
STOREL 1
L1:
PUSHL 1
PUSHI 1
SUPEQ
JZ L2
PUSHL -1
PUSHL 1
PUSHI 1
SUB
CHARAT
PUSHS "1"
CHRCODE
EQUAL
JZ L3
PUSHL 2
PUSHL 3
ADD
STOREL 2
JUMP L4
L3:
L4:
PUSHL 3
PUSHI 2
```

```
MUL
STOREL 3
PUSHL 1
PUSHI 1
SUB
STOREL 1
JUMP L1
L2:
PUSHL 2
STOREL 0
PUSHL 0
STOREL -2
RETURN
main:
PUSHS "Introduza uma string binária:"
WRITES
WRITELN
READ
STOREG 0
PUSHI 0
PUSHG 0
PUSHA FBinToInt
CALL
POP 1
STOREG 1
PUSHS "O valor inteiro correspondente é: "
WRITES
PUSHG 1
WRITEI
WRITELN
STOP
```

7. Conclusão

O desenvolvimento deste compilador para Pascal Standard permitiu consolidar de forma prática todos os conceitos teóricos dados nesta UC. O projeto foi concluído com sucesso, cumprindo todos os requisitos funcionais propostos no enunciado. A maior dificuldade que enfrentámos foi no gerador do código máquina, onde tivemos algumas dificuldade a gerir a stack e a memória.