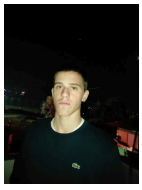
Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

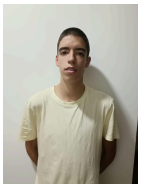
Ano Letivo de 2024/2025

Compilador para Pascal Standard

Rui Dantas
a104008



Rodrigo Dantas
a104009



Rodrigo Lima
a104181

Junho, 2025

CG

Data da Receção	
Responsável	
Avaliação	
Observações	

Compilador para Pascal Standard

Rui Dantas
a104008

Rodrigo Dantas
a104009

Rodrigo Lima
a104181

Junho, 2025

Índice

1. Introdução

A construção de um compilador é um dos desafios mais enriquecedores no estudo das ciências da computação, pois integra conhecimentos teóricos e práticos sobre linguagens de programação, algoritmos e arquitetura de computadores. Este relatório descreve o desenvolvimento de um compilador para a linguagem Pascal Standard, realizado no âmbito da disciplina de Processamento de Linguagens, para ser utilizado na máquina virtual EWVM.

Para a implementação, foram utilizadas ferramentas como PLY (Python Lex-Yacc), que facilitam a criação de analisadores léxicos e sintáticos. Além disso, o compilador foi testado com diversos exemplos de programas Pascal, incluindo estruturas como condicionais (if), ciclos (while, for), funções e manipulação de arrays.

2. Arquitetura

Na implementação deste projeto optamos por uma arquitetura simples e de fácil compreensão, mas facilmente modular para acrescentar/retirar regras da gramática e refletir essas alterações no compilador.

2.1. Lexer

O lexer é o primeiro componente a entrar em ação no processo de compilação, sendo responsável pela análise léxica do código-fonte. No âmbito deste projeto, foi implementado um analisador léxico para a linguagem Pascal utilizando o módulo lex da biblioteca PLY (Python Lex-Yacc).

A implementação do lexer foi estruturada de forma a reconhecer os diversos elementos que constituem a linguagem Pascal, incluindo palavras-chave como PROGRAM, VAR, BEGIN e END, tipos de dados como integer, real e boolean, operadores aritméticos e lógicos, símbolos especiais e diferentes tipos de literais. Todos os tokens foram definidos explicitamente através de regras específicas, garantindo uma abordagem uniforme e consistente para o processo de tokenização.

Uma das características notáveis desta implementação é o tratamento dado aos literais. Os números inteiros e reais são automaticamente convertidos para os seus tipos correspondentes em Python durante o processo de análise, enquanto as strings são processadas para remover as aspas que as delimitam no código fonte. Os comentários são identificados por um token, mas descartados.

O lexer inclui também mecanismos de tratamento de erros, que permitem identificar e reportar caracteres inválidos no código fonte, indicando a linha onde ocorrem.

```
tokens = (
    'PROGRAM', 'VAR', 'BEGIN', 'END', 'FUNCTION', 'FORWARD', 'EXTERNAL',
    'IF', 'THEN', 'ELSE', 'WHILE', 'DO', 'FOR', 'TO', 'DOWNTO',
    'IDENTIFIER', 'DIGSEQ', 'REALNUMBER', 'CHARACTER_STRING', 'NIL',
    'TREAL', 'TINTEGER', 'TBOOLEAN', 'TSTRING', 'TCHAR',
    'ASSIGNMENT', 'COLON', 'SEMICOLON', 'DOT', 'COMMA', 'DOTDOT',
    'PLUS', 'MINUS', 'STAR', 'SLASH', 'DIV', 'MOD', 'AND', 'OR', 'NOT',
    'EQUAL', 'NOTEQUAL', 'LT', 'GT', 'LE', 'GE', 'IN',
    'LPAREN', 'RPAREN', 'LBRAC', 'RBRAC', 'STARSTAR', 'UPARROW', 'COMMENT', 'ARRAY',
    'OF'
)
```

Figura 1: Lista de Tokens

2.2. Parser

Após a análise léxica realizada pelo lexer, o parser assume um papel fundamental no processo de compilação, sendo responsável por validar a estrutura sintática do código-fonte de acordo com as regras gramaticais que desenvolvemos. Utilizando os tokens gerados pelo lexer, o parser constrói uma árvore sintática abstrata (AST) que representa hierarquicamente o programa, verificando a correta formação de declarações, expressões, estruturas de controle e outros elementos da linguagem. Neste projeto, o parser foi implementado com o módulo yacc do PLY.

2.2.1. Gramática

A gramática que desenvolvemos é bastante simples e demos prioridade à sua clareza. Um dos principais aspetos com o qual tivemos maior dificuldade foi a correta organização da gramática de forma a garantir a prioridade de operações, bem como garantir que todos os ciclos estavam corretamente descritos na mesma.

2.2.2. Árvore Sintaxe Abstrata

Para simplificar o subsequente processo de tradução do código Pascal para a EWVM, optamos por adotar a construção de uma Abstract Syntax Tree (AST) durante o parsing. Nesse sentido, em vez das produções retornarem diretamente o código traduzido para a EWVM, elas retornam classes que representam os nós da nossa AST.

Todas as ASTs produzidas têm como raiz a classe AbstractSyntaxTree, que armazena todos os seus nós filhos numa lista. Esta classe é composta por métodos para mostrar a sua representação, realizar comparações e de tradução. Todos os nós filhos seguem uma estrutura semelhante, podendo ter mais ou menos atributos conforme a necessidade, e representam uma variedade de expressões que podem ocorrer no código Pascal.

Tal como se pode ver no exemplo de código que se segue, a produção p_ast() do nosso parser devolve o nodo AbstractSyntaxTree passando como argumento toda a gramática que foi reconhecida.

```
def p_program(self, p):
    '''program : program_heading SEMICOLON block DOT'''
    p[0] = ast.Program(p[1], p[3])
```

Figura 2: Regra p_program

2.3. Tradutor

2.3.1. Auxiliares

A estrutura de estado da tradução é composta por diversos dicionários e contadores que gerenciam variáveis e controlam o fluxo durante a compilação. O self.global_variables armazena variáveis globais como tuplos (índice, tipo, limite inferior, tipo elemento), gerindo e controlando posições de memória(stack). O self.local_variables organiza variáveis locais por função. O self.variable_counter incrementa índices de memória, enquanto self.if_counter, self.while_counter e self.for_counter geram etiquetas únicas para estes ciclos. O self.function_addresses mapeia funções e self.function_signatures valida parâmetros. O self.current_function identifica a função ativa, e self.predefined_procedures/functions definem instruções pré-construídas, com self.predefined_function_signatures especificando argumentos.

```
self.global_variables: Dict[str, Tuple[int, str, Optional[int], Optional[str]]] = {}
self.local_variables: Dict[str, Dict[str, Tuple[int, str, Optional[int], Optional[str]]]] = {}
self.variable_counter = 0
self.if_counter = 0
self.while_counter = 0
self.for_counter = 0
self.function_addresses: Dict[str, str] = {}
self.function_signatures: Dict[str, int] = {}
self.current_function: Optional[str] = None
self.bool_label_counter = 0
self.predefined_procedures = {
    "writeln": ["writeln"],
    "write": [],
    "readln": ["read"],
    "read": ["read"]
}
self.predefined_functions = {
    "length": ["strlen"],
    "charat": ["charat"]
}
self.predefined_function_signatures = {
    "length": 1,
    "charat": 2
}
```

Figura 3: Estruturas Auxiliares do Tradutor

2.3.2. Programa

Processa declarações globais com declare_variable se houver variáveis no bloco, gerando código de inicialização. Em seguida, adiciona um salto para a main e, se existirem funções, traduz cada uma com visit_function_declaration. Depois, marca o início com main: e start, executa as instruções do bloco e termina com stop, assegurando uma estrutura coesa e funcional.

```
1  jump main
2  main:
3  start
4  pushs "Ola, Mundo!"
5  writes
6  writeln
7  stop
```

Figura 4: Programa Simples - Teste Exemplo 1

2.3.3. Funções

2.3.3.1. Declarar Funções

Extrai o nome da função e o número de parâmetros do cabeçalho, registando-os em function_addresses e function_signatures para uso posterior em chamadas. Define o scope atual com self.current_function e inicializa um dicionário de variáveis locais para a função. Se existirem parâmetros, itera sobre eles, alocando memória local com índices incrementados e gerando código para cada um. Declara variáveis locais adicionais, se presentes, usando declare_variable. Caso o corpo da função seja um Block, traduz suas instruções com evaluate; se for um tuplo, processa diretamente. Finaliza com a instrução return, marcando o fim da função.

```
Power:
pushi 0
storeg 5
pushi 0
storeg 6
pushi 1
storeg 6
pushi 1
storeg 5
for0:
pushg 5
pushg 4
infeq
jz endfor0
pushg 6
pushg 3
mul
storeg 6
pushg 5
pushi 1
add
storeg 5
jump for0
endfor0:
pushg 6
return
```

Figura 5: Exemplo do Código gerado para declarar uma função

2.3.3.2. Chamar Funções

Identifica o nome da função e os parâmetros fornecidos, contando o número de argumentos. Se for uma função predefinida como length ou charat, valida o número de parâmetros usando predefined_function_signatures e gera código para os argumentos com evaluate, seguido das instruções EWVM correspondentes do predefined_functions. Para funções definidas pelo utilizador, verifica se existe em function_addresses, valida os parâmetros com function_signatures, avalia cada argumento e adiciona pusha com o endereço da função, finalizando com call para executar a chamada.

```
48    pusha Power
49    call
```

Figura 6: Exemplo do Código gerado para chamar uma função

2.3.4. Procedimentos

2.3.4.1. Chamar Procedimentos

Identifica o nome do procedimento e os argumentos fornecidos, convertendo o nome para minúsculas para consistência. Se for um procedimento predefinido como writeln ou readln, processa os argumentos com evaluate, adaptando o código com base no tipo (ex.: writei para inteiros, writes para strings) e adiciona instruções específicas como writeln para quebras de linha ou read para entrada. Para procedimentos definidos pelo utilizador, verifica em function_addresses, valida os parâmetros com function_signatures, avalia os argumentos e usa pusha com call para execução, assegurando a correta integração com a EWVM. Sabemos que esta é outra limitação do nosso projeto e que no futuro será necessário criar métodos especificos para reconhecer procedimentos em vez de usarmos os métodos das funções, visto que estes podem ser declarados de maneiras diferentes.

```
4   pushs "Ola, Mundo!"
5   writes
6   writeln
```

Figura 7: Exemplo do Código gerado para um procedimento(writes)

2.3.5. Variáveis

2.3.5.1. Declaração Variáveis

Analisa o tipo da variável a partir do type_denoter (definido na gramática). Se for um array, calcula o tamanho com base nos limites inferior e superior, alocando espaço com pushn e registando no dicionário apropriado (global_variables ou local_variables) com um tuplo contendo índice, tipo (“array”), limite inferior e tipo dos elementos, queremos dizer que apesar da implementação do pushn ainda não está 100% funcional. Para tipos simples como inteiros ou strings, inicializa com valores padrão (pushi 0 ou pushs “”) e armazena na memória com storeg, incrementando variable_counter.

```
1   pushi 0
2   storeg 0
3   pushi 0
4   storeg 1
5   pushi 0
6   storeg 2
7   jump main
```

Figura 8: Exemplo do Código gerado para declarar variaveis antes do inicio do programa

2.3.5.2. Variáveis Indexadas

Verifica se a variável é array ou string em local_variables ou global_variables, avaliando o índice com evaluate. Para arrays, ajusta o índice com o limite inferior e usa pushg para carregar o valor. Para strings, usa charat após subtrair 1 do índice, garantindo acesso correto.

2.3.5.3. Tradução Variáveis Indexadas

Calcula o índice da array com evaluate, ajusta pelo limite inferior se aplicável, soma o índice base da array e usa storeg para armazenar o valor, garantindo atribuição precisa em arrays, com validação de tipo.

2.3.6. Expressões

2.3.6.1. Assign Statement

Avalia a expressão à direita com expression.evaluate(self), obtendo o valor a ser atribuído, e infere seu tipo com infer_expression_type(função que irá ser detalhada posteriormente). Se a variável à esquerda for um VariableAccess, identifica o nome e verifica se é local ou global, usando local_variables ou global_variables para obter o índice e tipo, lançando erros se houver incompatibilidade de tipos ou variável não declarada, e finaliza com storeg para armazenar o valor. Para IndexedVariable, lida com arrays, gerando código de acesso ao índice com translate_indexed_variable_assignment, assegurando que o valor seja armazenado na posição correta da memória.

2.3.6.2. Expressões Binárias

Avalia as expressões à esquerda e à direita com evaluate, gerando o código correspondente para cada uma. Em seguida, obtém o operador (ex.: “+”, “=”, “<”) e utiliza um dicionário op_map para mapear o operador Pascal para instruções EWVM, como add para “+”, equal para “=”, ou inf para “<”. Se o operador não for suportado, lança um erro. Para operadores que requerem múltiplas instruções, como “<>”, que é traduzido como equal seguido de not, retorna uma lista com essas instruções; caso contrário, retorna uma única instrução, concatenando o código das sub-expressões e a operação.

```
22    pushg 6
23    pushg 3
24    mul
```

Figura 9: Exemplo do Código gerado para operações(multiplicação)

2.3.6.3. Expressões com Sinais

Avalia a expressão associada com expression.evaluate(self), gerando o código correspondente. Em seguida, verifica o sinal da expressão: se for “+”, retorna o código da expressão sem alterações, mantendo o valor original. Se for “-”, adiciona as instruções pushi −1 e mul ao código da expressão, multiplicando o valor por −1 para inverter o sinal.

2.3.6.4. Expressões de Exponenciação

Avalia a base com base.evaluate(self), gerando o código correspondente. Se o expoente for uma constante, usa evaluate_constant para obter o seu valor. Para expoente zero, retorna pushi 1. Para outros valores inteiros, duplica a base na pilha com dup 1 (n-1) vezes e aplica mul (n-1) vezes, como em base dup 1 mul para expoente 2.

2.3.6.5. Expressões Not

Avalia a expressão associada com expression.evaluate(self), gerando o código que calcula o valor booleano a ser negado. Em seguida, adiciona a instrução not, que inverte o valor na pilha (0 torna-se 1 e 1 torna-se 0), refletindo a lógica de negação.

2.3.6.6. Expressões Constantes

Extrai o valor da constante e verifica seu tipo. Para inteiros, gera pushi seguido do valor. Para reais, usa pushf. Para booleanos retorna pushi 1 ou pushi 0. Para caracteres ou strings de um caractere, converte o valor para o código ASCII com pushi. Para strings maiores, usa pushs.

```
44    charat
45    pushi 105
46    equal
```

Figura 10: Exemplo do Código gerado para comparação de um caracter

2.3.6.7. Avaliar Constantes

A função evaluate_constant converte expressões constantes em valores inteiros, verificando se o nó é Constant e extraindo o valor como inteiro ou float, lançando erro se não for constante numérica, usada para processar limites de arrays.

2.3.7. Ciclos

2.3.7.1. For

Declara a variável de controle, inicializa-a com o valor inicial, e usa for_counter para etiquetas. Compara a variável com o valor final usando infeq ou supeq conforme a direção (to ou downto), saltando para endfor se a condição falhar. Executa o corpo, ajusta a variável com add ou sub, e retorna ao início com jump, finalizando com endfor.

```
for0:
pushg 5
pushg 4
infeq
jz endfor0
pushg 6
pushg 3
mul
storeg 6
pushg 5
pushi 1
add
storeg 5
jump for0
endfor0:
```

Figura 11: Exemplo do Código gerado para um ciclo for

2.3.7.2. While

Incrementa while_counter para etiquetas únicas, avalia a condição e usa jz para saltar para endwhile se falsa. Traduz o corpo do ciclo e adiciona jump para voltar ao início do while, fechando com a etiqueta endwhile, implementando a repetição condicional.

```
jz endwhile0
pushg 0
pushg 1
mod
pushi 0
equal
jz endif0
pushi 0
storeg 2
endif0:
pushg 1
pushi 1
add
storeg 1
jump while0
endwhile0:
```

Figura 12: Exemplo do Código gerado para um ciclo while

2.3.7.3. If

Primeiro, incrementa if_counter para criar etiquetas únicas, avalia a condição com evaluate, e gera um salto condicional jz para a etiqueta endif (ou else se houver um ramo else). Traduz o bloco then e, se houver um else, adiciona um salto jump para endif após o then, seguido do código do else, finalizando com a etiqueta endif, garantindo o controlo de fluxo correto.

```
jz else0
pushg 0
pushg 2
sup
jz else1
pushg 0
storeg 3
jump endif1
else1:
pushg 2
storeg 3
endif1:
jump endif0
```

Figura 13: Exemplo do Código gerado para ifs encadeados.

2.3.8. Deteção de Erros

Ao longo de todas estas funções e secções explicadas anteriormente já fomos explicando a forma como faziamos a deteção de erros, ou seja, à medida que vamos gerando código vamos tratando dos erros e parando quando encontramos um. Com vista a melhorar a deteção de erros, criamos a função infer_expression_type infere o tipo de uma expressão, analisando nós como Constant, VariableAccess, IndexedVariable ou BinaryExpression. Para constantes, determina o tipo pelo valor; para variáveis, consulta local_variables ou global_variables; para arrays, retorna o tipo do elemento; para operações, resolve com base nos operandos, lançando erro se indeterminado.

```
Processing ./Tests/Error/test1.txt:
--------------------------------------------------
Translation error in ./Tests/Error/test1.txt: Variable 'x' not declared
--------------------------------------------------
```

Figura 14: Exemplo de Deteção de Erros

3. Trabalho Futuro

A tradução do código Pascal para a máquina virtual EWVM, implementada na classe PascalEWVMTranslator, reflete um esforço notável, embora incompleto, para conectar a linguagem Pascal à EWVM. Apesar dos avanços em estruturas como atribuições, condições e ciclos, muitas áreas do Pascal permanecem por explorar. Procedimentos, embora parcialmente tratados, carecem de suporte completo, enquanto statements como manipulação de ficheiros ou exceções estão ausentes. Existem funções que definimos no translator, com instruções EWVM, mas nem todas estão funcionais devido a inconsistências na máquina virtual, detectadas em testes preliminares (load, store). Além disso, algumas funções do PascalEWVMTranslator, destinadas a operações avançadas ou tipos complexos, permanecem não operacionais, uma consequência de desafios técnicos e constrangimentos de tempo que impediram testes exaustivos. Este relatório, portanto, captura apenas um esboço do trabalho. Apesar disto, a base estabelecida é sólida, oferecendo um ponto de partida para futuras melhorias. Reconhecemos estas lacunas como oportunidades para refinamento, motivando-nos a superar os obstáculos e expandir esta ponte entre Pascal e EWVM.

4. Conclusão

O compilador para Pascal Standard, desenvolvido para a disciplina de Processamento de Linguagens, funciona bem e traduz programas para a máquina virtual EWVM, usando o PLY e uma arquitetura modular com lexer, parser e tradutor. Não está 100% operacional para todos os pedidos em Pascal, com limitações como o pushn incompleto para arrays e falta de métodos próprios para procedimentos, mas está pronto para os exemplos pedidos pelo docente, lidando com condicionais, ciclos, funções e arrays de forma eficaz.