Universidade do Minho

Departamento de Informática

Licenciatura em Engenharia Informática

Processamento de Linguagens

Grupo 06

Mariana Morais

Sofia Oliveira

Tomás Pinto

Mariana Morais A100662
Sofia Oliveira A104536
Tomás Pinto A104448

Maio de 2025

Conteúdo

1 Introdução

Com este trabalho pretendemos implementar um compilador que lida com código escrito em linguagem de programação Pascal para código da máquina virtual.

Este projeto tem como principais objetivos aprofundar a experiência na área da engenharia de linguagens e da programação generativa baseada em gramáticas. Procuramos reforçar a competência na escrita de gramáticas, tanto livres de contexto (GIC) como gramáticas tradutoras (GT), e promover o desenvolvimento de processadores de linguagem através do método de tradução dirigida pela sintaxe, partindo de uma gramática tradutora.

Adicionalmente, visa-se a criação de um compilador capaz de gerar código para um alvo específico, recorrendo a ferramentas de geração automática de compiladores baseadas em gramáticas tradutoras — nomeadamente o Yacc, na versão PLY para Python, complementado pelo Lex, igualmente nesta versão.

Este projeto tem os seguintes objetivos:

- Suportar programas Pascal
- Suportar variáveis;
- Suportar expressões aritméticas;
- Suportar condicionais;
- Suportar ciclos;
- Suportar subprogramas (procedure e function)

Pretendemos também evoluir na UC e testar os nossos conhecimentos adquiridos ao longo do semestre.

1.1 Pascal

Pascal é uma linguagem de programação de alto nível que se destaca pela sua clareza e organização. A sua sintaxe é simples de seguir, baseada em blocos bem definidos por palavras-chave como *begin e end*, o que torna o código mais fácil de ler. Trabalhar com Pascal implica declarar explicitamente variáveis, constantes e até os tipos de dados, o que ajuda a manter tudo bem estruturado e sem ambiguidades. Também é possível criar subprogramas, como procedimentos (procedure) e funções (function), o que facilita muito a reutilização de código e a divisão do programa em partes mais pequenas e fáceis de gerir.
A linguagem oferece um bom conjunto de estruturas de controlo, como as instruções condicionais *if...then...else*, ciclos *while e for*, e ainda a instrução *case*, útil quando há várias opções possíveis. Para além disso, permite definir tipos compostos como arrays, registos (records), conjuntos (sets) e ficheiros (files), o que dá ao programador bastante flexibilidade. Uma das suas características mais importantes é o facto de ser fortemente tipada, ou seja, é preciso declarar todas as variáveis com o seu tipo antes de usá-las, o que ajuda a evitar muitos erros na fase de desenvolvimento.

1.2 EWVM

A EWVM é uma máquina virtual orientada a stack. Para além da stack principal onde são realizadas operações aritméticas e lógicas, a EWVM disponibiliza também uma call stack para controlo de chamadas de funções e procedimentos, uma string heap para armazenamento de cadeias de caracteres, e uma struct heap para gestão de estruturas de dados complexas.

2 Metodologia

Com base no que estudamos, o nosso trabalho está dívidido em 4 fases:

- Análise léxica;
- Análise sintática;
- Analise semântica;
- Geração de Código;

De forma simples, o analisador léxico é responsável por ler os caracteres individuais do código-fonte e convertê-los em tokens significativos.

O analisador sintático, por sua vez, verifica se esses tokens estão organizados de acordo com as regras gramaticais da linguagem, formando estruturas válidas.

Já a análise semântica, que ocorre aquando da análise sintática, tem como objetivo assegurar que essas estruturas fazem sentido, ou seja, que o código é coerente em termos de tipos, declarações e contexto.

Cada uma destas fases é fundamental no processo de compilação ou interpretação de um programa.

3 Análise Léxica

Criamos uma lista de todos os tokens que o lexer deve reconhecer. Cada token é uma sequência de caracteres que representa uma unidade básica na linguagem Pascal. Para cada token na lista tokens, é definida uma expressão regular que define como o token é reconhecido na entrada.

```
tokens = [
    'PROGRAM',
    'VAR',
    'BEGIN',
    'END',
    'WRITE',
    'WRITELN',
    'READLN',
    'IF',
    'THEN',
    'ELSE',
    'FOR',
    'TO',
    'DOWNTO',
    'DO',
    'WHILE',
    'AND',
    'OR',
    'NOT',
    'FUNCTION',
    'PROCEDURE',
    'IDENTIFIER',
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'INTDIV',
    'MOD',
    'ASSIGN',
    'LPAREN',
    'RPAREN',
    'LBRACKET',
    'RBRACKET',
    'SEMICOLON',
    'COLON',
    'COMMA',
    'DOT',
    'LT',
    'GT',
    'LE',
    'GE',
    'EQ',
    'NE',
    'INTEGER',
    'BOOLEAN',
    'STRING',
    'REAL',
    'CHAR',
    'ARRAY',
    'OF',
    'TEXT',
    'TRUE',
    'FALSE',
    'DOTDOT',
    #'COMMENT',
]
```

Listing 1: Python example

4 Análise Sintática

Nesta fase implementamos um analisador sintático para a linguagem de programação Pascal usando a biblioteca PLY. Ele converte comandos Pascal em instruções específicas. As regras gramaticais são definidas como funções Python que processam os tokens de entrada e geram instruções correspondentes.

A gramática seguinte constitui a estrutura e as regras para a linguagem mencionada.

```
Z : expressao
expressao : declaracao_programa declaracao_var bloco_programa
declaracao_programa : PROGRAM IDENTIFIER SEMICOLON
declaracao_var : VAR declaracoes_variaveis
               | empty
declaracoes_variaveis : lista_variaveis COLON tipo SEMICOLON
                      | declaracoes_variaveis lista_variaveis COLON tipo SEMICOLON
lista_variaveis : IDENTIFIER
                | IDENTIFIER COMMA lista_variaveis
tipo : INTEGER
     | REAL
     | STRING
     | BOOLEAN
     | CHAR
     | array_tipo
array_tipo : ARRAY LBRACKET NUMBER DOTDOT NUMBER RBRACKET OF tipo
bloco_programa : bloco DOT
bloco : BEGIN lista_comandos END
lista_comandos : comando
       | lista_comandos SEMICOLON comando
               | lista_comandos comando
comando : comando_simples
        | comando_composto
comando_simples : atribuicao
                | write
                | writeln
                | read
                | comando_if
                | comando_while
                | comando_for
comando_composto : bloco
atribuicao : IDENTIFIER ASSIGN expressao_aritmetica
           | IDENTIFIER LBRACKET expressao_aritmetica RBRACKET ASSIGN expressao_aritmetica
expressao_aritmetica : expressao_aritmetica PLUS termo
                     | expressao_aritmetica MINUS termo
                     | expressao_aritmetica TIMES termo
                     | expressao_aritmetica DIVIDE termo
                     | expressao_aritmetica INTDIV termo
                     | expressao_aritmetica MOD termo
                     | termo
termo : LPAREN expressao_aritmetica RPAREN
      | NUMBER
      | IDENTIFIER
      | TEXT
      | TRUE
      | FALSE
      | LENGTH LPAREN IDENTIFIER RPAREN
      | IDENTIFIER LBRACKET expressao_aritmetica RBRACKET
write : WRITE LPAREN argumentos RPAREN
writeln : WRITELN LPAREN argumentos RPAREN
read : READLN LPAREN argumentos RPAREN
argumentos : argumento
           | argumento COMMA argumentos
argumento : TEXT
          | IDENTIFIER
          | IDENTIFIER LBRACKET expressao_aritmetica RBRACKET
expressao_condicional : expressao_condicional AND expressao_condicional
                      | expressao_condicional OR expressao_condicional
                      | expressao_aritmetica GT expressao_aritmetica
                      | expressao_aritmetica LT expressao_aritmetica
                      | expressao_aritmetica GE expressao_aritmetica
                      | expressao_aritmetica LE expressao_aritmetica
                      | expressao_aritmetica EQ expressao_aritmetica
```

```
                      | expressao_aritmetica NE expressao_aritmetica
                      | NOT expressao_condicional
                      | LPAREN expressao_condicional RPAREN
                      | expressao_aritmetica
comando_if : IF expressao_condicional THEN comando
           | IF expressao_condicional THEN comando ELSE comando
comando_while : WHILE expressao_condicional DO comando
comando_for : FOR IDENTIFIER ASSIGN expressao_aritmetica TO expressao_aritmetica DO comando
            | FOR IDENTIFIER ASSIGN expressao_aritmetica DOWNTO expressao_aritmetica DO comando
empty :
```

5 Análise Semântica

A análise semântica tem como objetivo verificar se as instruções fazem sentido no contexto do programa. Nesta fase, verificamos a coerência do código em termos de tipos de dados e declarações de variáveis.

Para a implementação da análise semântica, usamos uma abordagem em que, à medida que o código é analisado, são feitas verificações para garantir que as variáveis estão corretamente declaradas antes de serem usadas, que os tipos são compatíveis nas expressões e que as operações estão semanticamente corretas.

A seguir, apresentamos um exemplo de uma verificação semântica realizada:

```
for var in var_list:
        if isinstance(tipo, tuple) and tipo[0] == 'ARRAY':
            start_idx, end_idx = tipo[1], tipo[2]
            size = end_idx - start_idx + 1
            address = len(symbol_table)

            symbol_table[var] = {
                'type': 'ARRAY',
                'range': (start_idx, end_idx),
                'element_type': tipo[3],
                'address': address
            }

            for i in range(size):
                symbol_table[f"{var}[{i}]"] = {
                    'address': address + i,
                    'type': tipo[3]
                }


        else:
            if var not in symbol_table:
                address = len(symbol_table)
                symbol_table[var] = {
                    'address': address,
                    'type': tipo,
                    'value': None
                }
            else:
                print(f"Erro: Variavel '{var}' j   declarada.")
```

No exemplo, verificamos se a variável já foi declarada antes. Caso a variável já exista na tabela de símbolos, é emitido um erro ("Erro: Variável 'var' já declarada.").

Definição de Arrays: Se a variável for um array, o código define o intervalo de índices e cria uma entrada separada na tabela para cada elemento do array.

Atribuição de Endereço: Para cada variável (ou elemento de array), é atribuído um endereço de memória (baseado no tamanho da tabela de símbolos).

6 Geração de Código

A fase de geração de código corresponde ao momento em que o nosso compilador traduz diretamente as construções da linguagem Pascal para instruções que podem ser executadas pela máquina virtual de destino. Esta tradução é feita de forma imediata e integrada no próprio analisador sintático, seguindo o princípio da **tradução dirigida pela sintaxe**.

Para cada estrutura reconhecida — como expressões, atribuições, comandos de controlo, entre outros — foram definidas ações semânticas que geram, em tempo real, o código correspondente. Este código é construído através da concatenação de instruções numa variável global, respeitando a lógica sequencial e imperativa esperada pela máquina virtual.

Ao longo desta fase, assegura-se que cada elemento da linguagem fonte é corretamente transformado em instruções que realizam as mesmas operações e produzem o mesmo comportamento durante a execução. Nesta secção, descrevemos em detalhe como cada tipo de construção do Pascal foi mapeada para o conjunto de instruções da máquina virtual, ilustrando os principais mecanismos utilizados.

6.1 Operações Aritméticas

As expressões aritméticas são implementadas com regras recursivas na gramática, respeitando a precedência dos operadores. Para cada operação binária (`+`, `-`, `*`, `/`, `div`, `mod`), os operandos são empilhados com `PUSHI` ou `PUSHG`, e a operação é aplicada com a instrução apropriada da máquina virtual (`ADD`, `SUB`, `MUL`, `DIV`, `MOD`).

Por exemplo, a expressão:

```
x := a + b * 2;
```

Gera:

```
PUSHG <a>
PUSHG <b>
PUSHI 2
MUL
ADD
STOREG <x>
```

As expressões entre parênteses são tratadas recursivamente, respeitando a precedência definida na gramática.

6.2 Variáveis

Durante a análise sintática e semântica, o compilador constrói uma tabela de símbolos onde associa a cada variável o seu tipo, endereço e valor (quando inicializado). A memória global da máquina virtual é utilizada para armazenar os valores das variáveis, sendo os acessos realizados com `PUSHG` (leitura) e `STOREG` (escrita).

Por exemplo:

```
x := 10;
```

Gera:

```
PUSHI 10
STOREG <endereço de x>
```

Arrays

Arrays são tratados como blocos contínuos de memória. Na declaração, o compilador calcula o intervalo de índices e atribui um endereço de base, reservando espaço para todos os elementos. Cada acesso ao array é traduzido numa operação de aritmética de endereços.

Se o índice for constante:

```
arr[2] := 5;
```

Gera:

```
PUSHI 5
STOREG <endereço base + offset>
```

Se o índice for dinâmico, é gerado código para:

- Avaliar o índice e guardá-lo temporariamente.
- Verificar se o índice está dentro dos limites.
- Escolher o endereço correto com base no valor calculado.

Também é gerado código que imprime uma mensagem de erro em tempo de execução se o índice estiver fora dos limites:

```
PUSHS "Index out of bounds"
WRITES
```

6.3 Comandos de Escrita: `write` e `writeln`

As instruções `write` e `writeln` suportam múltiplos argumentos e são adaptadas ao tipo de dado:

- `WRITES` para strings.
- `WRITEI` para inteiros e caracteres.
- Para booleanos, imprime-se "true"ou "false"com lógica condicional.

Exemplo com booleano:

```
PUSHG <addr>
JZ L_false
PUSHS "true"
WRITES
JUMP L_end
L_false:
PUSHS "false"
WRITES
L_end:
```

6.4 Leitura com `readln`

O comando `readln` gera instruções para leitura de dados da entrada padrão:

- `READ` + `ATOI` para inteiros.
- `READC` para caracteres.
- `READ` para strings.

No caso de arrays, o valor lido é armazenado num endereço calculado com base no índice.

6.5 Expressões Condicionais

As comparações (`=`, `<>`, `<`, `>`, `<=`, `>=`) são traduzidas para operações da máquina virtual como `EQ`, `SUP`, `INF`, etc. Expressões lógicas compostas (`AND`, `OR`, `NOT`) usam rótulos e saltos para simular avaliação com curto-circuito. Exemplo de `AND`:

```
<expr1>
JZ L_false
<expr2>
JZ L_false
PUSHI 1
JUMP L_end
L_false:
PUSHI 0
L_end:
```

6.6 Comando `if`

O comando `if` é traduzido com instruções de salto:

```
<condição>
JZ L_else
<bloco then>
JUMP L_end
L_else:
<bloco else>
L_end:
```

6.7 Ciclos `while` e `for`

O ciclo `while` é gerado como:

```
L_start:
<condição>
JZ L_end
<bloco>
JUMP L_start
L_end:
```

O ciclo `for` inclui:

- Inicialização da variável.
- Comparação com o limite.
- Incremento ou decremento.
- Repetição do corpo do ciclo.

A versão com `downto` usa `SUB` em vez de `ADD`, e a comparação é invertida.

6.8 Outras Funcionalidades

Implementámos ainda suporte à função `length`, que pode ser usada sobre variáveis do tipo `string`. Esta função gera a instrução `STRLEN`, devolvendo o comprimento da string.

Exemplo:

```
PUSHG <endereço>
STRLEN
```

6.9 Considerações Finais

A geração de código foi implementada de forma modular, com base nas ações semânticas associadas às regras da gramática. A arquitetura do compilador facilita a expansão futura com funcionalidades como procedimentos, funções e estruturas de dados compostas.

O uso de rótulos auxiliares e instruções de salto permite controlar o fluxo de execução, e garante que a semântica dos programas Pascal seja respeitada no código gerado. Esta fase foi essencial para garantir que os programas reconhecidos são corretamente traduzidos e executáveis na máquina virtual.

7 Testes

1. Exemplo Inicial

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

```
    Resultado:
PUSHS "Ola, Mundo!"
WRITES
```

2. Exemplo Operação Aritmética

```
program SomaDoisInteiros;
var
  num1, num2, soma: Integer;
begin
  num1 := -5;
  num2 := 10;

  soma := num1 + num2;

  writeln('A soma dos dois numeros e: ', soma);
end.
```

```
    Resultado:
PUSHI -5
STOREG 0
PUSHI 10
STOREG 1
PUSHG 0
PUSHG 1
ADD
STOREG 2
PUSHS "A soma dos dois numeros e: "
WRITES
PUSHG 2
WRITEI
```

3. Exemplo IF/ELSE

```
program TesteIfElse;
var
  num: Integer;
begin
  num := -3;
  if num > 0 then
    writeln('Positivo');
  else
    writeln('Negativo');
end.
```

```
    Resultado:
PUSHI -3
STOREG 0
PUSHG 0
PUSHI 0
SUP
JZ L0
PUSHS "Positivo"
WRITES
JUMP L1
L0:
PUSHS "Negativo"
WRITES
L1:
```

4. Exemplo Loop

```
program Loop;
var
  i: Integer;
begin
  i := 0;
  while i < 3 do
  begin
    writeln('i = ', i);
    i := i + 1;
  end;
end.
```

```
    Resultado:
PUSHI 0
STOREG 0
L0:
PUSHG 0
PUSHI 3
INF
JZ L1
PUSHS "i = "
WRITES
PUSHG 0
WRITEI
PUSHG 0
PUSHI 1
ADD
STOREG 0
JUMP L0
L1:
```

5. Exemplo Booleanos

```
program TestaBooleanos;
var
  a, b: integer;
begin
  a := 5;
  b := 10;
  if not (a < b) or (a = 5 and b = 10) then
    writeln('Expressao booleana funciona!');
end.
```

```
    Resultado:
 PUSHI 5
STOREG 0
PUSHI 10
STOREG 1
JZ L0
PUSHS "Expressao booleana funciona!"
WRITES
L0:
```

8 Conclusão e Trabalho Futuro

Com a conclusão do trabalho prático, torna-se pertinente uma reflexão crítica sobre o mesmo. Pretendemos evidenciar não apenas os sucessos alcançados, mas também os principais obstáculos enfrentados. O projeto revelou-se uma etapa importante na construção de um sistema de processamento de linguagem, estabelecendo uma base consistente para a análise léxica.

A implementação do lexer abriu caminho para fases subsequentes, como a análise sintática e a geração de código, essenciais para o desenvolvimento de um compilador ou interpretador completo da linguagem Pascal, destinada à execução numa máquina virtual.

Apesar dos avanços, o trabalho apresentou desafios significativos, sobretudo na definição e constante adaptação da gramática. Foi necessário garantir que esta acomodasse novas funcionalidades sem comprometer as estruturas já estabelecidas, o que exigiu uma atenção contínua ao longo do desenvolvimento.