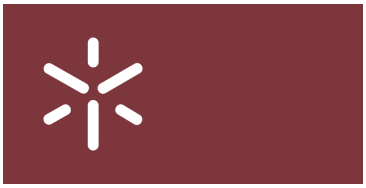


Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2024/2025

Trivial Pascal - Grupo 45

Afonso Pedreira A104537
Fábio Magalhães A104365
André Pinto A104267

Maio, 2025

PL

Trivial Pascal - Grupo 45

Afonso Pedreira A104537

Fábio Magalhães A104365

André Pinto A104267

Maio, 2025

Índice

1. Introdução

Este relatório tem como objetivo apresentar o trabalho prático realizado no âmbito da unidade curricular de Processamento de Linguagens. O projeto consistiu no desenvolvimento de um compilador para a linguagem Pascal Standard, cuja especificação foi fornecida pelos docentes.

O principal objetivo do projeto foi construir um compilador capaz de processar programas escritos em Pascal, traduzindo-os para código compatível com a EWVM — uma máquina virtual disponibilizada para este efeito. Para tal, desenvolvemos um sistema que inclui análise léxica, análise sintática, análise semântica e geração de código.

Para a construção do compilador, recorremos à biblioteca ***Python PLY***, que fornece ferramentas para a criação de analisadores léxicos e sintáticos, nomeadamente os módulos **ply.lex** e **ply.yacc**.

Ao longo deste relatório, iremos descrever a arquitetura do compilador, os detalhes da implementação e justificar as decisões tomadas durante o desenvolvimento.

2. Arquitetura do Projeto

Sendo esta uma fase crucial no desenvolvimento de qualquer projeto, foi aqui que o grupo optou por se focar maioritariamente, de modo a reduzir qualquer erro que pudesse surgir de forma inesperada. Tal como foi lecionado nesta unidade curricular, é necessário, antes de qualquer outra tarefa, definir uma **GIC** que serve de base para o desenvolvimento.

Posteriormente, procede-se à definição do **lexer**, com o objetivo de tokenizar o código Pascal fornecido. Essa tokenização é essencial para a construção da **Árvore de Sintaxe Abstrata** (AST), que facilita substancialmente a interpretação e manipulação da estrutura do programa.

Nas subseções seguintes, iremos detalhar cada um dos constituintes principais deste compilador, explicando as suas responsabilidades e a forma como se interligam para permitir a compilação de programas Pascal.

2.1. GIC

Após várias tentativas e de muita remodelação, o grupo chegou a um resultado satisfatório no que toca à **GIC** desenvolvida e, tal como referido anteriormente, este é o passo fundamental para o correto desenvolvimento do analisador léxico e sintático que serão abordados nas próximas secções.

A nossa gramática apresenta uma estrutura simples e direta, cobrindo na totalidade as construções suportadas pela linguagem `Pascal`, conforme especificado no enunciado do projeto.

Relativamente à convenção de nomes adotada para os *tokens*, optámos por prefixos que indicassem claramente a sua natureza:

- `cXXX` para constantes, como `cINTEGER`, `cREAL`, `cCHAR`, `cSTRING` e `cBOO`
- `oXXX` para operadores e símbolos, como `oPLUS`, `oMUL`, `oSEMI`, `oDOT`, `oASSIGN`, etc.
- `kXXX` para palavras-chave (keywords) reservadas, como `kIF`, `kTHEN`, `kELSE`, `kVAR`, `kBEGIN`, `kEND`, entre outras
- `yNAME` para identificadores definidos pelo utilizador (variáveis, funções, etc.)

Esta convenção permitiu manter o código organizado, distinguindo de forma clara os diferentes tipos de tokens, facilitando a leitura, a manutenção e a depuração do compilador ao longo do desenvolvimento.

2.2. Analisador Léxico

Sendo o Pascal uma linguagem relativamente simples, sem muitas ambiguidades, a implementação foi bastante direta e seguimos todo o planeamento que nos pareceu mais coerente. Assim, os *tokens* definidos no nosso *lexer* são:

```
cCHAR, cINTEGER, cREAL, cBOO, cSTRING,
oLP, oRP, oLB, oRB, oPLUS, oMINUS,
oMUL, oDIV, oASSIGN, oEQUAL, oLT, oGT,
oLE, oGE, oUNEQU, oCOMMA, oSEMI,
oCOLON, oDOTDOT, oDOT,
yNAME, kDOWNTO,
kAND, kARRAY, kBEGIN, kDO, kELSE, kEND,
kFOR, kIF, kMOD, kNOT, kOF, kOR,
kPROGRAM, kTHEN, kTO, kVAR, kWHILE, kDIV,
SYS_FUNCT, SYS_PROC, SYS_TYPE
```

Cada *token* foi cuidadosamente definido com expressões regulares adequadas à linguagem Pascal, incluindo distinção entre palavras-chave e identificadores, bem como suporte a tipos numéricos, caracteres, strings e operadores diversos.Para tal, utilizámos o módulo **ply.lex**, que nos permitiu especificar regras precisas de forma clara e eficiente.

Apenas *tokens* simples foram definidos diretamente sem regras (como é o caso de caracteres unitários como parênteses, vírgulas ou pontos). Todos os restantes foram definidos com funções e regras claras, de modo a evitar ambiguidades. Não foi usada uma lista de palavras reservadas estática, pois considerámos ser uma prática desnecessária e desaconselhada no contexto desta unidade curricular.

2.3. Analisador Sintático

Após vários testes ao analisador léxico e de verificar a sua corretude com diversos programas Pascal (disponíveis na pasta dos testes), procedemos então à construção do pilar deste compilador: o **parser**.

Tal como lecionado nas aulas teóricas e práticas, demos uso às funcionalidades do módulo `yacc` do **PLY**, de modo a construir o melhor parser possível, garantindo uma ligação direta e eficiente com o nosso analisador léxico. O tipo de parsing suportado pelo `yacc` é do tipo **LALR**, bastante eficiente e amplamente utilizado na indústria para análise sintática.

Com a gramática já definida, a sua implementação revelou-se relativamente simples. No entanto, foi necessário recorrer a regras de precedência especificamente para lidar com as construções do tipo `if ... then ... else`, que geraram conflitos do tipo **shift/reduce**. Apesar de termos recorrido a uma solução técnica baseada em precedência, reconhecemos que uma abordagem alternativa que evitasse esses conflitos seria ideal num cenário de maior complexidade.

2.3.1. Árvores Sintáticas Abstratas

De modo a tornar a compreensão do resultado do analisador sintático mais simples, o grupo optou por criar uma estrutura intermédia que obedecesse às nossas necessidades e tornasse a interpretação da informação mais clara, além de poder ser reutilizada na fase de geração de código.

Assim, esta árvore funciona como o motor de interpretação da informação, sendo que armazena todos os elementos do programa numa estrutura hierárquica, composta por nós pai e seus respetivos filhos.

O nó raiz da Árvore Sintática Abstrata (AST) corresponde à produção inicial do programa, conforme exemplificado:

```
def p_program(p):
    'program : kPROGRAM yNAME oSEMI program_block oDOT'
    p[0] = ast.Program(name=p[2], block=p[4])
```

Cada produção do *parser* atribui ao `p[0]` uma instância de uma classe definida no ficheiro `ast_nodes.py`, representando um nó da *AST*. Estas classes são estruturadas de forma a conter os atributos essenciais e a permitir a geração de uma representação textual e visual da árvore.

Além disso, implementámos funcionalidades auxiliares, como o `pretty_print_ast` para impressão indentada da árvore, e `graphviz()` para a geração de um ficheiro .png com a representação visual da mesma.

Por exemplo, se o código que queremos obter código máquina for:

```
program BasicArithmetic;
var
  a, b, c: integer;
begin
  a := 3;
  b := 4;
  c := a + b * 2 - (a - b) div 2;
  writeln('Result of basic arithmetic operations: ', c);
end.
```

Então a respetiva AST corresponde a:

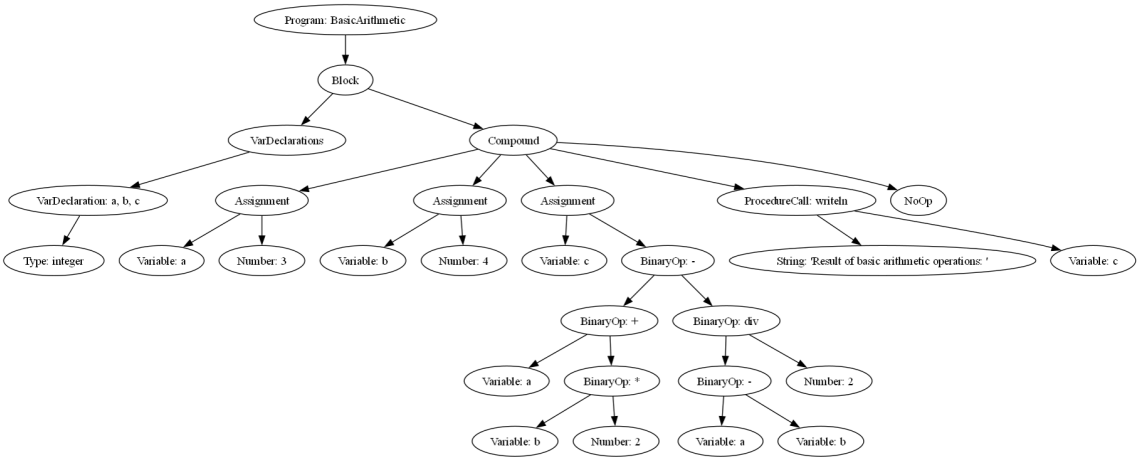


Figura 1: AST para programa de teste

Com esta representação, é nítida a organização presente, e também é possível ver como as precedências estão corretamente aplicadas no que toca a operações aritméticas.

2.4. Geração de Código

Após a correta definição das fases anteriores, a AST revelou-se essencial para a geração correta de código para a **EWVM**. A nossa abordagem assentou na separação clara de responsabilidades, criando um módulo dedicado exclusivamente à geração de código. Isto facilita a manutenção do compilador e permite facilmente realizar alterações ou melhorias futuras na forma como o código final é emitido.

O processo de geração de código é baseado numa travessia da árvore sintática abstrata. Cada nó da *AST* corresponde a uma construção da linguagem Pascal e está associado a um método específico que sabe gerar o respetivo código em linguagem de máquina para a *EWVM* e esta abordagem torna o sistema modular e extensível.

A lógica de geração foi encapsulada numa classe `CodeGenerator`, que percorre recursivamente os nós da AST utilizando o método `generate(node)`, que faz **dispatch** dinâmico com base no tipo do nó. Este design permite adicionar novos tipos de instruções facilmente, bastando criar um novo método.

Assim, após o *parser* construir a *AST*, a geração de código é invocada de forma simples e direta:

```
result = parser.parse(data, lexer=lexer)
generator = CodeGenerator()
generator.generate(result)
machine_code = generator.get_code()
```

O código gerado é então guardado num ficheiro de saída e pode ser diretamente executado na máquina virtual EWVM. Este processo garante que a semântica dos programas Pascal é corretamente preservada e traduzida para um formato interpretável pela VM.

2.4.1. Análise Semântica

No decorrer da geração do código, é feita também uma análise semântica cuidada, de forma a validar a completude e coerência do programa. Esta etapa assegura que não ocorrem erros como atribuições entre tipos incompatíveis, uso de variáveis não declaradas ou operações inválidas (por exemplo, aplicar `div` a strings).

A validação semântica está embutida nos métodos da classe `CodeGenerator`, que ao processar cada nó da AST, verifica os tipos e a validade das operações realizadas. Esta estratégia garante que apenas programas semanticamente válidos são traduzidos para código EWVM, aumentando assim a fiabilidade e robustez do compilador.

Deste modo, a estrutura modular adoptada garante uma divisão clara entre as fases de parsing e geração de código, e a AST torna-se uma representação intermédia poderosa, capaz de suportar transformações futuras e otimizações.

2.4.2. Estratégias Utilizadas

Neste capítulo, vamos apresentar as estratégias utilizadas para implementar os diversos elementos da linguagem `Pascal`, explicando como cada construção foi tratada na nossa AST e transformada em código EWVM.

2.4.2.1. Atribuições e Operações Matemáticas

As operações matemáticas são representadas por nós do tipo `BinaryOp`, que incluem operadores como `+`, `-`, `*`, `div`, `mod`, entre outros. A geração de código distingue entre inteiros e

reais, utilizando instruções como `add` ou `fadd`, consoante o tipo dos operandos. A atribuição é feita com o nó `Assignment`, que armazena o valor gerado num endereço global obtido da tabela de símbolos.

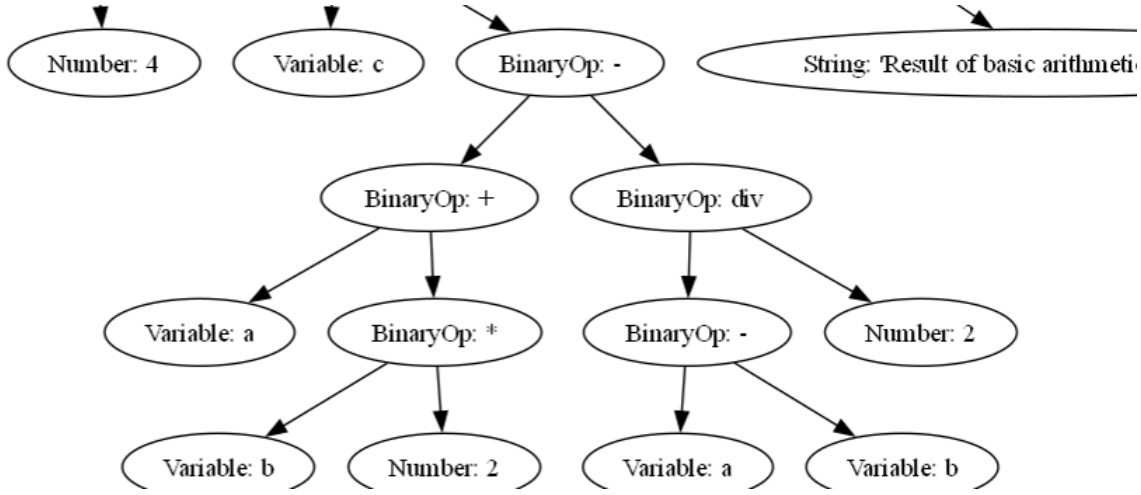


Figura 2: Operações matemáticas

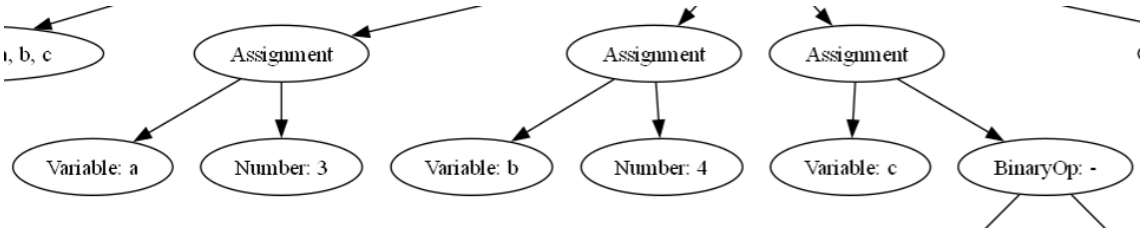


Figura 3: Atribuições

2.4.2.2. Operadores Lógicos

Operações como `and`, `or`, e `not` são tratados da mesma forma que operadores aritméticos, com conversão dos operandos e uso de instruções apropriadas da EWVM. O nó `UnaryOp` lida com `not`, enquanto `BinaryOp` trata `and` e `or`.

2.4.2.3. Variáveis

Cada variável declarada é armazenada na `symbol_table` com um offset. Na geração de código, o seu valor é acedido com `pushg offset` e atribuído com `storeg offset`. O tipo é guardado em `type_info` para validações futuras.

Um excerto de código máquina gerado com base nesta abordagem pode ser:

```
pushi 0
storeg 0   ; variável a = 0
pushi 0
storeg 1   ; variável b = 0
pushi 0
storeg 2   ; variável c = 0
```

Aqui, cada `storeg n` corresponde ao armazenamento de uma variável no *offset* `n`. Esta alocação é feita no início do programa com valores iniciais definidos (neste caso, `0`).

2.4.2.4. Constantes

Os nós `Number`, `CharLiteral`, `String`, e `Boolean` tratam valores constantes. São traduzidos diretamente para `pushi`, `pushs` ou `pushf`, consoante o tipo.

2.4.2.5. Arrays

Arrays são alocados com `allocn` e o endereço é armazenado em memória global. O acesso e atribuição usam `loadn` e `storen`, ajustando os índices com base no limite inferior do *array*, que é guardado em `array_info`.

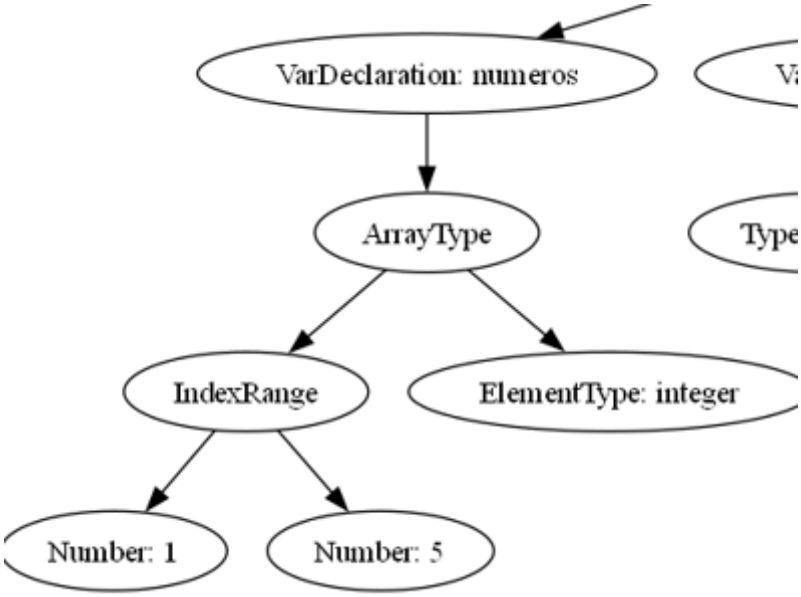


Figura 4: Declaração Array

Assim, após a geração do código máquina obtemos o seguinte resultado para declarar o *array*:

```
pushi 5
allocn
storeg 0
```

2.4.2.6. Palavras Reservadas e Funções Internas

Funções como `length`, `abs`, `succ`, `pred`, `odd`, entre outras, são tratadas diretamente no método `generate_function_call`. Cada função é traduzida para a instrução EWVM correspondente. No caso de `length`, são verificadas strings e arrays separadamente.

2.4.2.7. Condicionais (`if`)

Instruções `if` geram um label `ELSE{n}` e `ENDIF{n}`. Primeiro é avaliada a condição, seguido de salto condicional com `jz`. O bloco `else`, se existir, é adicionado após o `jump` para o fim.

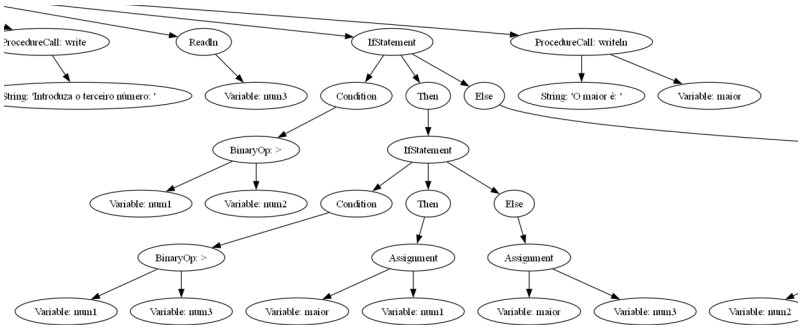
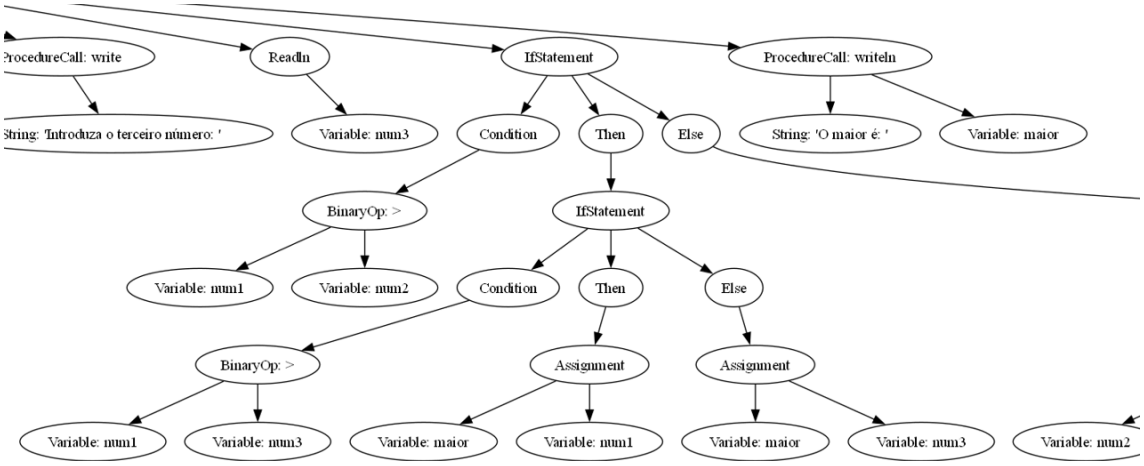


Figura 5: If



Um bloco if, em código máquina, tem a seguinte estrutura:

```
pushg 0
pushg 1
sup
jz ELSE0
; instruções se condição for verdadeira
...
jump ENDIF0
ELSE0:
; instruções se condição for falsa (else)
...
ENDIF0:
```

2.4.2.8. Ciclos (`while` e `for`)

Ciclos `while` e `for` usam labels numerados `WHILE{n}` , `ENDWHILE{n}` , `FOR{n}` , e `ENDFOR{n}` . São geradas instruções de comparação e salto (`jz` , `jump`) para garantir o controlo de fluxo correto. O ciclo `for` considera a direção (`to` ou `downto`) na lógica de comparação.

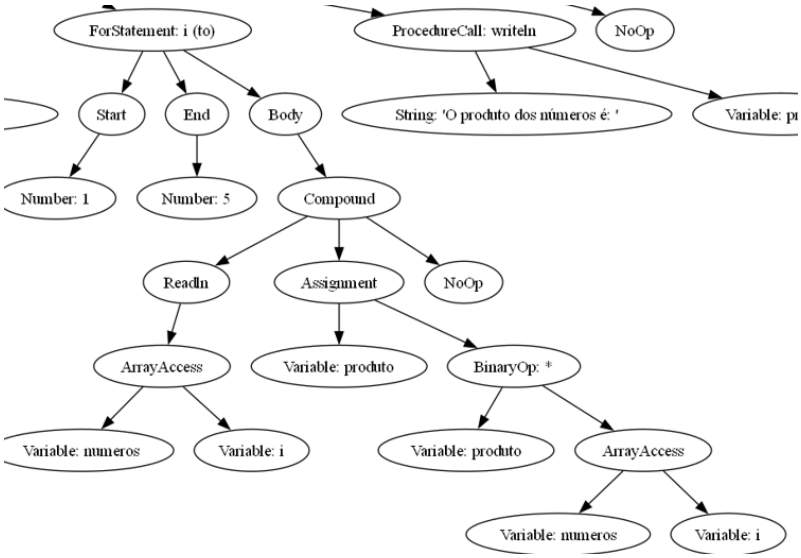


Figura 6: Ciclo For

Um ciclo for, em código máquina, tem a seguinte estrutura:

```
pushi 1       ; valor inicial
storeg 1
FOR0:
pushg 1
pushi 5       ; valor final
infeq
jz ENDFOR0
; corpo do ciclo
...
pushg 1
pushi 1
add
storeg 1
jump FOR0
ENDFOR0:
```

Já o ciclo `while` tem estrutura semelhante, com labels próprios e a verificação da condição logo no início:

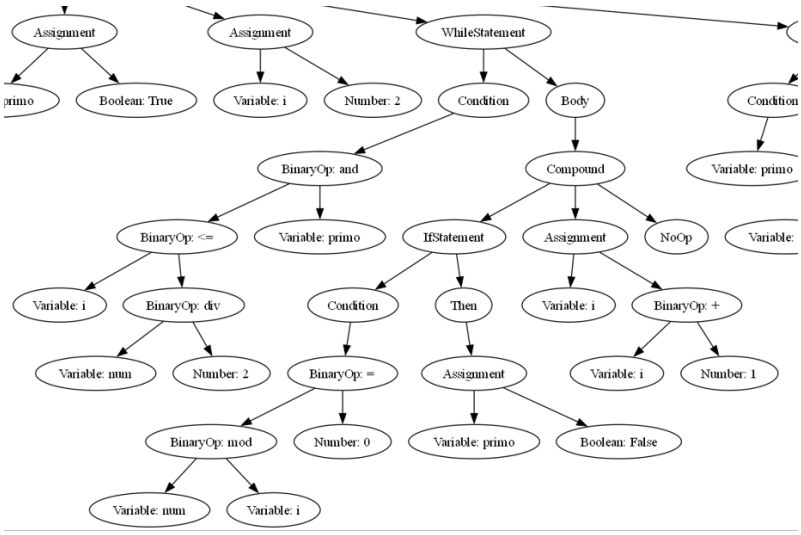


Figura 7: Ciclo While

```
WHILE0:
; condição
pushg 1
pushg 0
pushi 2
div
infeq
pushg 2
and
jz ENDWHILE0
; corpo
...
jump WHILE0
ENDWHILE0:
```

Desta forma, é garantido que os ciclos seguem corretamente a semântica de execução da linguagem Pascal, usando labels únicos e instruções de salto para controlar a repetição e saída do ciclo.

2.4.2.9. Declaração e Verificação de Variáveis

Durante a geração, é validado se as variáveis estão declaradas e se os tipos são compatíveis. A atribuição entre tipos diferentes (por exemplo, `integer` para `real`) é convertida com `itof`, enquanto incompatibilidades geram erro.

Esta verificação é feita diretamente no método `generate_Assignment` do `CodeGenerator`. Por exemplo:

```python
if not self._types_compatible(var_type, expr_type):
    raise TypeError(f"Incompatible types in assignment: {var_type} := {expr_type}")

if var_type == 'real' and expr_type == 'integer':
    self.code.append("itof")
```

Também é feita análise de coerência de tipos em operações binárias:

```python
if left_type != right_type:
    if 'real' in [left_type, right_type]:
        if left_type == 'integer':
            self.code.insert(-len(self.code_right), "itof")
        if right_type == 'integer':
            self.code.append("itof")
        result_type = 'real'
    else:
        raise TypeError(f"Type mismatch in binary operation: {left_type} {node.op}
{right_type}")
```

E em expressões condicionais, é exigido tipo `boolean`:

```python
if cond_type != 'boolean':
    raise TypeError(f"Condition expression must be boolean, got {cond_type}")
```

Por fim, funções internas como `length` validam os tipos esperados:

```python
if arg_type not in ('string', 'array'):
    raise TypeError(f"Function 'length' requires string or array, got {arg_type}")
```

Estas validações asseguram que o código gerado não só é correto em termos sintáticos como também respeita as regras semânticas da linguagem Pascal.

2.4.2.10. Labels e Geração de Rótulos

Cada instrução de controlo (`if` , `while` , `for` , `abs` , etc.) usa labels únicos como `IF{n}` , `ENDIF{n}` , `WHILE{n}` , que são gerados com um contador (`label_count`). Isto evita conflitos e permite saltos seguros dentro do código da VM.

3. Problemas, desafios e Trabalho Futuro

Com o concluir deste projeto, o grupo considera que obteve um resultado satisfatório, mas reconhece que existem áreas significativas onde o compilador de Pascal Standard pode ser melhorado.

A lacuna principal prende-se com a **ineficiência do código gerado**, que, devido a limitações temporais, não foi possível otimizar como desejado. O foco esteve na correção funcional, garantindo a tradução correta de programas Pascal para instruções da EWVM, mas não necessariamente na performance ou legibilidade do código gerado.

A dificuldade principal prendeu-se com a **simulação precisa do comportamento do Pascal na máquina virtual EWVM**, o que exigiu cuidados específicos na geração de instruções e uso rigoroso de convenções da VM. Isso acabou por resultar em algumas funções mais complexas, que poderiam ser simplificadas numa implementação que usasse estruturas de dados mais robustas ou uma abstração mais elaborada para o gerador de código.

No futuro, gostaríamos de melhorar a estrutura interna do compilador para suportar otimizações, incluir um sistema de **escopos para variáveis locais e globais**, bem como melhorar o **tratamento de erros**, adicionando mensagens mais informativas e detecção precoce de incoerências.

Além disso, seria interessante adicionar suporte a **funções definidas pelo utilizador** e manipulação de **ficheiros**, que embora estejam fora do escopo atual, são características relevantes do Pascal e enriquecem a análise de uma linguagem realista.

4. Conclusão

Com a conclusão deste projeto, o grupo compreendeu efetivamente a importância do desenvolvimento de um bom compilador e teve a oportunidade de explorar os processos envolvidos na construção dos mesmos. Ao longo do trabalho, foi possível aplicar, de forma prática, os conhecimentos adquiridos nas aulas, desde a definição de uma gramática até à geração de código para uma máquina virtual.

O projeto permitiu desenvolver competências técnicas como a implementação de um analisador léxico e sintático com o PLY, bem como a elaboração de uma Árvore Sintática Abstrata (AST) e a sua posterior interpretação para código da EWVM. Este percurso consolidou a compreensão sobre o funcionamento interno de compiladores e a importância de uma arquitetura bem definida e modular.

Apesar dos desafios encontrados, consideramos que os objetivos essenciais foram atingidos, ficando também identificadas várias oportunidades de melhoria e extensão futura do compilador desenvolvido.

Bibliografia

- [1] «PLY (Python Lex-Yacc) Documentation». David Beazley. Disponível em: https://www.dabeaz.com/ply/ply.html
- [2] «PLY: Python Lex-Yacc». PyPI. Disponível em: https://pypi.org/project/ply/
- [3] «The Python Language Reference». Python Software Foundation. Disponível em: https://docs.python.org/3/reference/index.html
- [6] «Free Pascal Language Reference Guide». Free Pascal Team. Disponível em: https://www.freepascal.org/docs-html/ref/ref.html
- [7] «LALR Parser». Wikipedia. Disponível em: https://en.wikipedia.org/wiki/LALR_parser
- [9] «Context-Free Grammar (CFG)». Geeks for Geeks. Disponível em: https://www.geeksforgeeks.org/context-free-grammar-cfg-introduction/
- [10] «Design and Implementation of a Simple Compiler». TutorialsPoint. Disponível em: https://www.tutorialspoint.com/compiler_design/index.htm

GIC

```
program : kPROGRAM yNAME oSEMI program_block oDOT

program_block : declarations compound_statement

declarations : var_declaration declarations
             | empty

var_declaration : kVAR var_declaration_list oSEMI

var_declaration_list : var_declaration_item
                     | var_declaration_list oSEMI var_declaration_item

var_declaration_item : name_list oCOLON type_spec

name_list : yNAME
          | name_list oCOMMA yNAME

type_spec : SYS_TYPE
          | yNAME
          | array_type

array_type : kARRAY oLB index_range oRB kOF type_spec

index_range : expression oDOTDOT expression

compound_statement : kBEGIN statement_list kEND

statement_list : statement
               | statement_list oSEMI statement

statement : if_statement
          | assignment_statement
          | procedure_call
          | compound_statement
          | while_statement
          | for_statement
          | empty

if_statement : kIF expression kTHEN statement %prec kTHEN
             | kIF expression kTHEN statement else_part

else_part : kELSE statement

assignment_statement : lvalue oASSIGN expression

procedure_call : yNAME
               | SYS_PROC oLP argument_list oRP
               | SYS_PROC

argument_list : expression
```

```
              | argument_list oCOMMA expression

while_statement : kWHILE expression kDO statement

for_statement : kFOR yNAME oASSIGN expression direction expression kDO statement

direction : kTO
          | kDOWNTO

expression : simple_expression
           | simple_expression relop simple_expression

relop : oEQUAL
      | oUNEQU
      | oLT
      | oLE
      | oGT
      | oGE

simple_expression : term
                  | sign term
                  | simple_expression addop term

sign : oPLUS
     | oMINUS

addop : oPLUS
      | oMINUS
      | kOR

term : factor
     | term mulop factor

mulop : oMUL
      | oDIV
      | kDIV
      | kMOD
      | kAND

lvalue : yNAME
       | yNAME oLB expression oRB

factor : lvalue
       | number
       | cBOO
       | char_literal
       | string
       | oLP expression oRP
       | kNOT factor
       | SYS_FUNCT oLP argument_list oRP
       | SYS_FUNCT

number : cINTEGER
       | cREAL

string : cSTRING

char_literal : cCHAR
```

```
empty :
```