Projeto de Processamento de Linguagens 2025

Construção de um Compilador para Pascal Standard

Relatório Técnico

Grupo 12 - Equipa Bugbusters 🪲🚫

Ana Sá Oliveira
(a104437)

Inês Silva Marques
(a104263)

José Rafael de
Oliveira Vilas Boas
(a76350)

1 de junho de 2025

Resumo

Este projeto teve como objetivo o desenvolvimento de um compilador para a linguagem Pascal Standard. O compilador foi construído em várias etapas, começando pela análise léxica, utilizando a biblioteca ply.lex para transformar o código-fonte em uma sequência de tokens. A seguir, foi implementada a análise sintática com ply.yacc, validando a estrutura do programa com base na gramática da linguagem. A partir do código reconhecido, foi construída uma árvore de sintaxe abstrata (AST), que representa de forma estruturada os elementos do programa. Com base nessa AST, foi realizada a análise semântica, incluindo verificação de tipos de dados, declarações de variáveis e coerência do código. Na etapa de geração de código, a AST foi transformada diretamente em código para a máquina virtual do projeto. Para cada etapa, foram realizados testes automáticos com os exemplos de código Pascal fornecidos no enunciado.

Conteúdo

Capítulo 1

Introdução

O objetivo deste projeto foi desenvolver um compilador para a linguagem Pascal standard. Mais concretamente, pretendeu-se implementar um compilador capaz de analisar, interpretar e traduzir código Pascal para um formato intermediário e deste para código máquina ou diretamente para código máquina, neste caso da VM disponibilizada aos alunos. A equipa optou por primeiro passar o código Pascal para um formato intermediário e só depois passar deste para código máquina. O compilador deve ser capaz de processar programas Pascal standard, incluindo declaração de variáveis, expressões aritméticas e comandos de controle de fluxo (if, while, for), e, opcionalmente, subprogramas (procedure e function).

Para construir este compilador, tivemos várias etapas:

1. Análise léxica;
2. Análise sintática;
3. Árvore de sintaxe abstrata (AST);
4. Análise semântica;
5. Geração de código.

Para além dessas etapas, foi também necessário realizar testes, os quais foram conduzidos de forma contínua ao longo do desenvolvimento.

Capítulo 2

Análise Léxica

A análise léxica consistiu em implementar um analisador léxico (lexer) para converter código Pascal numa lista de tokens. Para implementar este lexer, usamos a ferramenta ply.lex, já conhecida das aulas.

2.1 Palavras Reservadas

O primeiro passo foi identificar as palavras reservadas da linguagem Pascal standart.

and	array	begin	case
const	div	do	downto
else	end	file	for
function	goto	if	in
label	mod	nil	not
of	or	packed	procedure
program	record	repeat	set
then	to	type	until
var	while	with	

Cada palavra reservada resulta num token que definimos.

Por exemplo, o token BEGIN (palavra reservada begin) definimos da seguinte forma:

```
def t_BEGIN(t):
    r'[bB][eE][gG][iI][nN]\b'
    return t
```

Nesta expressão regular, cada letra aparece entre colchetes com as duas versões maiúscula e minúscula, o que permite que o reconhecimento seja *case insensitive*, aceitando `begin`, `Begin`, `BEGIN`, etc.
Temos ainda uma *word boundary* (limite de palavra), garantindo que o lexer reconheça apenas a palavra completa `begin` e não uma parte de outra palavra maior, como `beginning`

2.2 Literals

O segundo passo foi identificar os simbolos únicos, os literals, compostos apenas por um caracter, desta linguagem.

+	-	*	/
=	<	>	[
]	.	,	:
;	^	(	)

Cada um destes simbolos é um token.

2.3 Restantes tokens

O terceiro passo foi identificar os restantes tokens:

NOT_EQUAL	LESS_EQUAL	GREATER_EQUAL	ASSIGNMENT
RANGE	LPA	ARP	LPP
PRP	BOOL	DATATYPE	ID
INT	REAL	STRING	COMMENT
CHAR			

Os tokens do analisador léxico incluem operadores relacionais como o `NOT_EQUAL`, que representa o símbolo de diferente (`<>`), além do `LESS_EQUAL` e `GREATER_EQUAL`, correspondentes aos operadores "menor ou igual" (`<=`) e "maior ou igual" (`>=`), respectivamente. O token `ASSIGNMENT` identifica o operador de atribuição usado em Pascal, que é representado por `:=`. Para representar intervalos, utiliza-se o token `RANGE`, que corresponde ao símbolo `...`
O token `BOOL` é usado para valores booleanos, como `true` e `false`, e o `DATATYPE` identifica os tipos de dados básicos da linguagem, incluindo `integer`, `real`, `char`, `boolean` e `string`. Para nomes de variáveis, funções e procedimentos, o token `ID` é responsável por reconhecer os identificadores.

Números inteiros e reais são capturados pelos tokens `INT` e `REAL`, respectivamente, enquanto cadeias de caracteres delimitadas por aspas são reconhecidas pelo token `STRING`. Apenas um caracter entre aspas simples será reconhecido pelo token `CHAR`. Por fim, o token `COMMENT` é utilizado para identificar os comentários no código, que são ignorados durante a compilação.
Quanto a símbolos delimitadores, temos os tokens `LPA` e `ARP`, que correspondem a`(*` e `*)`, respectivamente, enquanto `LPP` e `PRP` representam `(.` e `.)`.
Assim, o analisador léxico, o `pascal_analex.py`, transforma o código pascal numa sequência de tokens. Se existir algum caracter ilegal, o lexer irá mostrar onde ocorreu o erro.

Se quisermos testar o lexer com o input do terminal:

```
python3 pascal_analex.py
```

Se quisermos testar o lexer com um ficheiro:

```
python3 pascal_analex.py < ../examples/exemplo1.pas
```

Se quisermos testar com um exemplo concreto do enunciado:

```
python3 pascal_analex.py 1
```

Capítulo 3

Análise Sintática

A análise sintática consistiu em construir um analisador sintático (parser) para validar a estrutura gramatical do código. Para implementar o parser usamos a ferramenta ply.yacc.

3.1 Gramática

O primeiro passo consiste em definir uma gramática da linguagem:

p0: S' → program
p1: program → PROGRAM ID ; declarations code_block .
p2: program → declarations code_block .
p3: declarations → declarations declaration
p4: declarations → ϵ
p5: declaration → variables_declaration
p6: declaration → function
p7: declaration → procedure
p8: variables_declaration → VAR variables_list
p9: variables_list → variables_list same_type_variables
p10: variables_list → same_type_variables
p11: same_type_variables → id_list : DATATYPE ;
p12: same_type_variables → id_list : ARRAY [INT RANGE INT] OF DATATYPE ;
p13: id_list → id_list , ID
p14: id_list → ID
p15: var_or_not → variables_declaration
p16: var_or_not → ϵ
p17: function → FUNCTION ID (parameters) : DATATYPE ; var_or_not code_block ;
p18: procedure → PROCEDURE ID (parameters) ; var_or_not code_block ;
p19: parameters → parameter_list

p20: parameters → ϵ
p21: parameter_list → parameter_list ; parameter
p22: parameter_list → parameter
p23: parameter → VAR_opt id_list : DATATYPE
p24: VAR_opt → VAR
p25: VAR_opt → ϵ
p26: code_block → BEGIN algorithm END
p27: algorithm → algorithm ; statement
p28: algorithm → statement
p29: statement → assignment
p30: statement → func_call
p31: statement → loop
p32: statement → code_block
p33: statement → if
p34: statement → else
p35: statement → ϵ
p36: if → IF cond THEN statement
p37: else → IF cond THEN statement ELSE statement
p38: assignment → ID ASSIGNMENT cond
p39: assignment → ID [INT] ASSIGNMENT cond
p40: assignment → ID [ID] ASSIGNMENT cond
p41: loop → for
p42: loop → while
p43: for → FOR for_cond DO statement
p44: for_cond → assignment TO cond
p45: for_cond → assignment DOWNTO cond
p46: while → WHILE cond DO statement
p47: cond → expr
p48: cond → expr op_rel expr
p49: op_rel → =
p50: op_rel → NOT_EQUAL
p51: op_rel → ¡
p52: op_rel → LESS_EQUAL
p53: op_rel → ¿
p54: op_rel → GREATER_EQUAL
p55: expr → termo
p56: expr → expr op_ad termo
p57: termo → fator
p58: termo → termo op_mul fator
p59: op_ad → +
p60: op_ad → -

p61: op_ad → OR
p62: op_mul → *
p63: op_mul → /
p64: op_mul → AND
p65: op_mul → MOD
p66: op_mul → DIV
p67: fator → value
p68: fator → (cond)
p69: fator → NOT fator
p70: value → ID
p71: value → INT
p72: value → REAL
p73: value → STRING
p74: value → BOOL
p75: value → ID [INT]
p76: value → ID [ID]
p77: value → func_call
p78: value → CHAR
p79: func_call → ID (args)
p80: args → elems
p81: args → ϵ
p82: elems → elems , cond
p83: elems → cond

3.2 Analisador Sintático

Definida a gramática a utilizar, implementamos o programa `pascal_anasin.py` que define funções para reconhecer as várias produções da gramática, utilizando o ply.yacc. Assim, para cada símbolo não terminal definido, temos um método que define as produções que derivam nesse símbolo e efetua as ações semânticas necessárias. Neste caso, como estamos a criar uma AST, a única ação semântica a realizar é ir construíndo a árvore, conforme se reconhecem símbolos essenciais para a gramática concreta que definimos, explicada melhor na secção seguinte. Algo a ressaltar, por diferir um pouco do que fizemos nas aulas é a definição de precedências no caso do `if` e `else`, para evitar um conflito de shift/reduce.

Se quisermos testar o parser com o input do terminal:

```
python3 pascal_anasin.py
```

Se quisermos testar o parser com um ficheiro:

```
python3 pascal_anasin.py < ../examples/exemplo1.pas
```

Se quisermos testar com um exemplo concreto do enunciado:

```
python3 pascal_anasin.py 1
```

Capítulo 4

Árvore de Sintaxe Abstrata (AST)

Durante a fase de análise sintática, à medida que regras de produção são reconhecidas, é possível produzir ações semânticas, uma destas ações consiste em criar uma representação intermédia da estrutura de um programa, podendo abstrair detalhes desnecessários da sintaxe, ficando apenas com os elementos essenciais à estrutura e semântica do programa, facilitando assim etapas posteriores de análise semântica e geração de código.
Como o analisador sintático é bottom-up do tipo LALR(1), começa o reconhecimento das folhas para a raíz, facilmente podemos criar uma árvore de sintaxe abstrata à medida que é feito o reconhecimento, pois em cada função correspondente a uma regra de produção, o reconhecimento dos seus filhos já foi feito.
Para tal, a partir das regras de produção da gramática, foram identificados os símbolos não terminais essenciais para a definição da estrutura hierárquica em árvore e semântica do programa, passando a criar uma classe para cada um destes símbolos identificados, que foram os seguintes:

`Program`, constituído por:

- Declarações, classe `Declaration`;
- Código, classe `CodeBlock`.

`Variable`, classe com informação de uma variável.

`Declaration`, classe abstrata contendo:

- `Variables`, constituída por:

  - Lista de classes `Variable`.

- `Function`, com informação relevante da função e constituída por:
  - Variáveis locais, classe `Variables`;
  - Variáveis de parâmetros, classe `Variables`;
  - Código da função, classe `CodeBlock`.
- `Procedure`, com informação relevante do procedimento e constituída por:
  - Variáveis locais, classe `Variables`;
  - Variáveis de parâmetros, classe `Variables`;
  - Código do procedimento, classe `CodeBlock`.

`Algorithm`, classe constituída por:

- Lista de classes `Statement`.

`Statement`, classe abstrata contendo:

- `Assignment`, com informação relevante de uma atribuição, constituída por:
  - Expressão, classe `Expression`.
- `Loop`, com informação relevante de cada tipo de ciclo, constituída por:
  - Condição, classe `Expression`;
  - Lista de classe `Statement`;
  - Atribuição, no caso de um ciclo `for`, classe `Assignment`.
- `If`, com informação relevante de cada tipo de condicional, constituída por:
  - Condição, classe `Expression`;
  - Para a condição verdadeira, lista de classes `Statement`;
  - Para a condição falsa, no caso de um `else`, lista de classes `Statement`.
- `CodeBlock`, constituída por:

– Algoritmo, classe `Algorithm`.

`Expression`, classe abstrata contendo:

- `BinaryOp`, operações binárias, para além do operador, constituída por:
  - Lado esquerdo da operação, classe `Expression`;
  - Lado direito da operação, classe `Expression`.
- `UnaryOp`, operações unárias, para além do operador, constituída por:
  - Expressão, classe `Expression`.
- `Value`, com informação relevante de um valor, como um número ou variável.
- `FunctionCall`, com informação relevante de uma chamada de função, constituída por:
  - Lista de argumentos, classes `Value` ou `FunctionCall`.

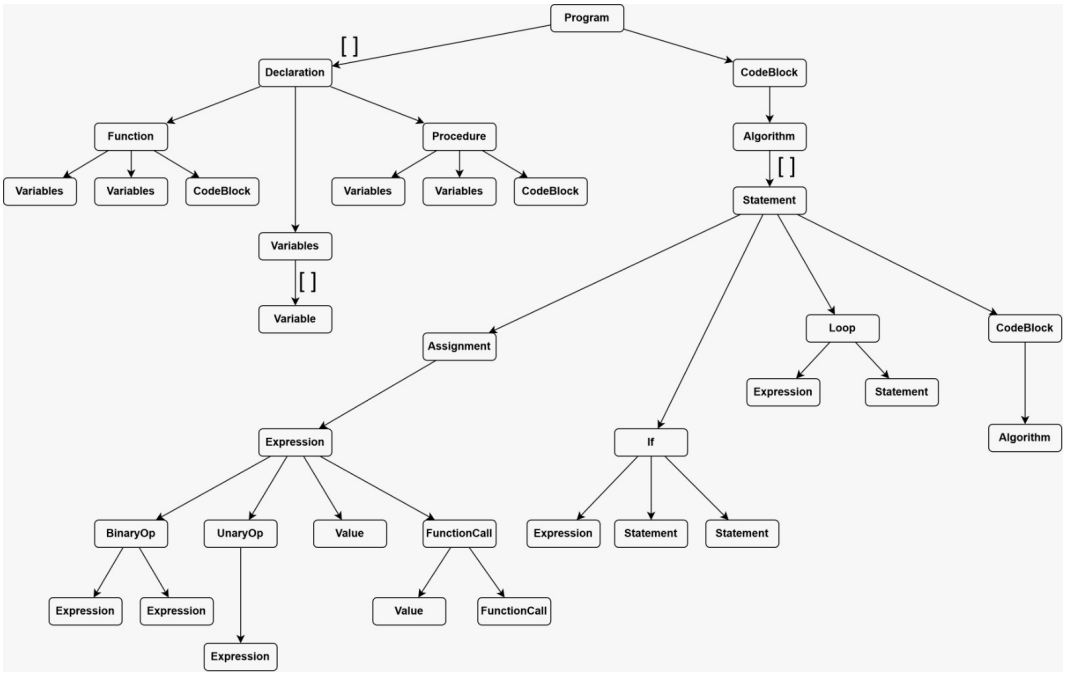

Figura 4.1: Árvore de Sintaxe Abstrata

Capítulo 5

Análise Semântica

A análise semântica consistiu em verificar tipos de dados, declaração de variáveis e coerência do código. Para o fazer, percorremos a AST e fomos guardando algumas informações e verificando outras.
Em concreto, na análise semântica:

1. não permitimos que se use variáveis que não tenham sido declaradas antes (dá erro);

2. verificamos, quando se atribui uma variável a uma expressão, que ambas teem o mesmo tipo;

3. verificamos o tipo dos operadores numa operação e vemos se são os adequados, por exemplo, não é adequado comparar uma string e um integer;

4. verificamos se as expressões usadas em comandos de controlo de fluxo `if` e `while` são booleanas;

5. não permitimos ter procedures `writeln`, `write` ou `readln` no meio de expressões aritméticas ou lógicas, uma vez que estes não retornam valor;

6. verificamos se no `for`, depois do `TO` ou do `DOWNTO`, temos um valor do tipo inteiro, ou uma expressão aritmética.

Se quisermos testar a análise semântica com o input do terminal:

```
python3 pascal_anasem.py
```

Se quisermos testar com um ficheiro:

```
python3 pascal_anasem.py < ../examples/exemplo1.pas
```

Se quisermos testar com um exemplo concreto do enunciado:

```
python3 pascal_anasem.py 1
```

Capítulo 6

Geração de Código

Após feita a análise semântica do programa, podemos passar à fase final deste pipeline: a geração de código vm a partir da definição estrutural e lógica criada do programa. Para tal foi necessário perceber como funciona o ambiente virtual de código máquina, para o qual teríamos de converter a estrutura de representação abstrata criada, ou seja, quais as operações suportadas pela vm, como funciona o seu o modelo de memória, de forma a compreender como fazer o acesso e armazenamento de variáveis e também como é feito o mecanismo de controlo para que possamos implementar estruturas de controlo como ciclos e condicionais.
Após esta análise foram, progressivamente, implementadas e testadas as conversões de cada classe para a geração de código, de forma a garantir o seu correto funcionamento, começando por classes estruturais como os blocos estruturais de um programa, por exemplo as declarações de variáveis, passando assim a poder definir atribuições simples, progressivamente integrando com operações binárias, dando assim possibilidade de passar para classes mais complexas com estruturas de controlo condicional, até chegar à conversão de todas as classes estruturais definidas.

Se quisermos testar o compilador com o input do terminal:

```
python3 pascal_compiler.py
```

Se quisermos testar com um ficheiro:

```
python3 pascal_compiler.py < ../examples/exemplo1.pas
```

Se quisermos ter um ficheiro de input com o código Pascal e um ficheiro de output com código da máquina virtual (o ficheiro vai para a pasta out):

```
python3 pascal_compiler.py ../examples/exemplo1.pas exemplo1.vm
```

Se quisermos testar com um exemplo concreto do enunciado:

```
python3 pascal_compiler.py 1
```

Para compilar o código diretamente para a vm, para quem tem Chrome:

```
python3 pascal_compiler.py ../examples/exemplo1.pas -vm https://ewvm.epl.di.umin
```

Ou então:

```
python3 pascal_compiler.py 1 -vm https://ewvm.epl.di.uminho.pt
```

Se tiver a vm localmente, para compilar o código diretamente para a vm (porta 27018), para quem tem Chrome:

```
python3 pascal_compiler.py ../examples/exemplo1.pas -vm
```

Ou então:

```
python3 pascal_compiler.py 1 -vm
```

Capítulo 7

Testes

Ao longo do desenvolvimento do projeto foram realizados vários testes.
Tanto o analisador léxico, como o analisador sintático, o semântico e o compilador teem os seus próprios programas. Assim, podemos testar cada fase individualmente.
Fizemos ainda o `pascal_test.py` que faz a analise léxica, sintática, semântica e gera código, isto tudo sobre os programas em Pascal usados como exemplo no enunciado do projeto.
Havia 7 programas em Pascal no enunciado. Todos os programas passam os testes de analise léxica, sintática e semântica porque são programas corretos, sem erros léxicos, sintáticos ou semânticos. É gerado código máquina para os 7 exemplos.
Para testar se o código máquina gerado é executável na máquina virtual e dá o resultado correto, temos de testar manualmente.

Para correr o teste geral com os 7 exemplos do enunciado fazemos:

```
python3 pascal_test.py
```

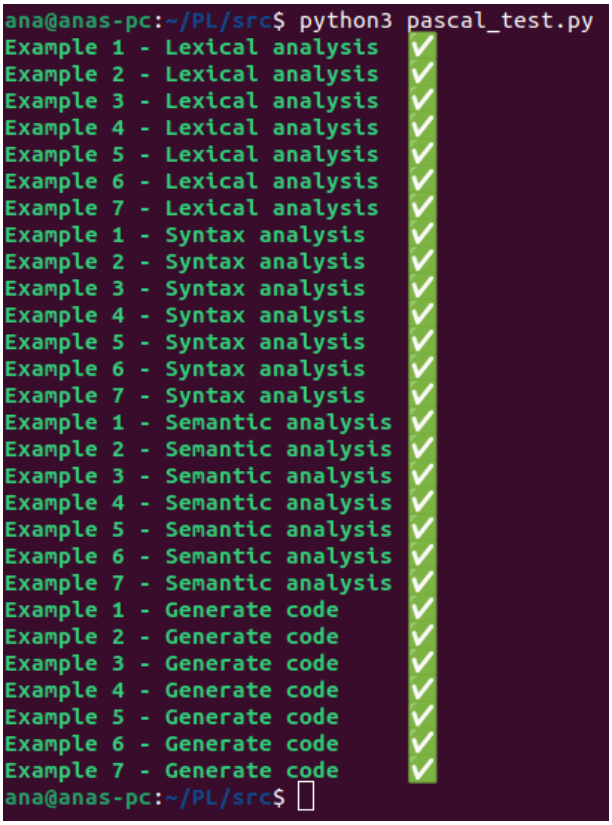

Figura 7.1: Testes automáticos.

Capítulo 8

Conclusão

Concluindo, conseguimos alcançar o objetivo principal deste projeto: construir um compilador para a linguagem Pascal Standard, que transforme código Pascal em código da máquina virtual. O nosso compilador é capaz de processar programas Pascal standart com declarações de variáveis, expressões aritméticas, comandos de controle de fluxo (if, while, for) e funções. Para além disto, seguimos as etapas do projeto: construímos um analisador léxico, um analisador sintático, fizemos uma árvore de sintaxe abstrata, fizemos um analisador semântico, geramos o código desejado e ainda fizemos testes de tudo isto, para garantir a correção de todos estes programas. Também testamos manualmente se os programas corriam como previsto na máquina virtual.

Ficaram por fazer alguns extras como otimizações do código máquina e suporte a procedures (que seria bastante parecido com as funções que fizemos). Os procedures estão suportados até ao nosso analisador sintático inclusive, mas depois não os implementamos nas fases seguintes.

Assim, concluímos que conseguimos cumprir os principais objetivos do projeto.