

Universidade da Beira Interior

Departamento de Informática



**Departamento de
Informática**

Compilador de Pico Crust

Elaborado por:

Dário Santos nº 10929
Raquel Guerra nº 10625

Docente:

Professor Doutor Simão Melo de Sousa

20 de janeiro de 2021

Conteúdo

Conteúdo	i
Lista de Figuras	v
Lista de Tabelas	vii
1 Introdução	1
1.1 Enquadramento	1
1.2 Objetivos	2
1.3 Organização do Documento	2
2 Linguagem Pico-Rust	3
2.1 Introdução	3
2.2 Ponto de Partida	3
2.3 Saída	4
2.4 Variáveis	4
2.4.1 Mutabilidade	5
2.4.2 <i>Shadowing</i>	6
2.4.3 Operações Aritméticas com Inteiros	6
2.5 Comentários	7
2.6 Expressões Booleanas	8
2.6.1 Operadores Relacionais	8
2.6.2 Instrução If	8
2.6.3 Instrução If, Else If e Else	9
2.6.4 Operadores Lógicos	10
2.7 Ciclos	11
2.7.1 Instruções <i>Break</i> e <i>Continue</i>	12
2.8 Funções	13
2.8.1 Funções Recursivas	15
2.9 Estruturas de Dados	15
2.9.1 Structs	16
2.9.2 Vetores	17
2.9.2.1 Vetores e Ciclos	18

2.9.2.2	Funções Especiais de Vetores	18
2.10	<i>Ownership</i>	19
2.11	Empréstimos Mutáveis	20
2.12	Conclusões	21
3	Implementação do Compilador	23
3.1	Introdução	23
3.2	Estrutura do Compilador de Pico-Rust	23
3.3	Analizador Léxico	24
3.3.1	Símbolos e Regras	24
3.3.2	Erros do Analizador Léxico	25
3.4	Analizador de Sintaxe	26
3.4.1	Regras do Analizador de Sintaxe	26
3.4.2	Árvore de Sintaxe Abstrata	27
3.4.3	Erros do Analizador Sintático	27
3.5	Tipagem	27
3.5.1	Árvore de Sintaxe Abstrata Tipada	28
3.5.2	Erros da Análise de Tipos	28
3.6	Analizador de <i>Ownership</i>	28
3.6.1	Erros da Análise de <i>Ownership</i>	29
3.7	Pré-Compilação	29
3.7.1	Tabelas Relacionais	30
3.7.1.1	Estruturas	30
3.7.1.2	Vetores	31
3.7.1.3	Funções	32
3.7.2	Árvore de Sintaxe Abstrata Pré-Compilada	33
3.8	Compilação	33
3.8.1	Expressões	33
3.8.1.1	Constantes	34
3.8.1.2	Declaração de Tipos Complexos	34
3.8.1.3	Chamada de Funções	36
3.9	Conclusões	37
4	Conclusões e Trabalho Futuro	39
4.1	Conclusões Principais	39
4.2	Trabalho Futuro	39
4.2.1	Retorno de Tipos Complexos	40
4.2.2	Passagem por Referência	40
4.2.3	Retorno de Referências	40
A	Palavras Reservadas	41

B	Associatividade e Precedência dos Operadores	43
C	Tipos de Dados	45
D	Gramática <i>BNF</i>	47
D.1	Instruções Globais	47
D.2	Tipos	48
D.3	Instruções Globais	48
D.4	Expressões	49
D.5	Operadores	50
E	Árvores de Tipos	51
E.1	Expressões	51
E.1.1	Constantes	51
E.1.2	Variáveis	51
E.1.3	Expressões	51
E.1.4	Atribuição	51
E.2	Operações	52
E.2.1	Operadores Binários	52
E.2.2	Operadores Unários	52
E.3	Instruções	52
E.3.1	Retorno	52
E.3.2	Vetores	52
E.3.2.1	Declaração	52
E.3.2.2	Tamanho	53
E.3.2.3	Acesso	53
E.3.3	Estruturas	53
E.3.3.1	Declaração	53
E.3.3.2	Acesso	53
E.3.4	if else	53
E.3.5	while	53
E.3.6	Bloco de Instruções	53
E.4	Instruções Globais	54
E.4.1	Declaração de Funções	54
E.4.2	Auto-Dereferenciação	54
E.5	Mutabilidade	54
E.6	Declaração de uma Estrutura	54
E.7	Declaração de uma Função	55
E.8	Ficheiros	55
	Bibliografia	57

Lista de Figuras

3.1	Ciclo de vida do compilador de Pico-Rust.	24
3.2	Erro provocado por símbolo não reconhecido pelo analisador léxico.	26
3.3	Erro provocado por uma derivação não reconhecida pelo analisador sintático.	27
3.4	Erro provocado pela utilização inválida dos tipos.	28
3.5	Erro provocado pelo analisador de <i>ownership</i> do Pico-Rust.	29
3.6	Tabela de suporte das posições relativas de uma estrutura na fase de pré-compilação.	31
3.7	Posições relativas dos elementos de um vetor face à posição inicial em Pico-Rust.	32
3.8	Tabela de suporte das posições relativas dos parâmetros de uma função em Pico-Rust.	33
3.9	Estado da pilha após a execução do código gerado para as constantes.	34
3.10	Estado da pilha após a declaração de uma estrutura.	35
3.11	Estado da pilha após a declaração de um vetor.	36
3.12	Estado da pilha após a chamada de uma função.	36

Lista de Tabelas

2.1	Operadores aritméticos do Pico-Rust.	7
2.2	Operadores relacionais do Pico-Rust.	8
2.3	Operadores lógicos do Pico-Rust.	10
3.1	Símbolos suportados pelo analisador léxico do Pico-Rust.	25
A.1	Palavras reservadas do Pico-Rust.	41
B.1	Associatividade e precedência dos operadores do Pico-Rust.	43
C.1	Tipos de dados suportados pelo Pico-Rust.	45
C.2	Tamanhos dos tipos de dados suportados pelo Pico-Rust.	45

Lista de Excertos de Código

2.1	Exemplo de um programa incompleto em Pico-Rust.	3
2.2	Exemplo de um programa completo em Pico-Rust.	4
2.3	Exemplo da utilização das funções de saída <code>print!</code> e <code>println!</code> . .	4
2.4	Estrutura da declaração de variáveis em Pico-Rust. e <code>println!</code> . . .	5
2.5	Exemplo de declaração inválida em Pico-Rust.	5
2.6	Exemplo de declaração de uma variável mutável em Pico-Rust. . .	6
2.7	Exemplo do efeito de <i>shadowing</i> em Pico-Rust.	6
2.8	Utilização de comentários em Pico-Rust.	7
2.9	Utilização de comentários num programa de Pico-Rust.	8
2.10	Estrutura de uma instrução <i>if</i> em Pico-Rust.	9
2.11	Estrutura de uma instrução <i>if</i> com ramo <i>else</i> em Pico-Rust.	9
2.12	Estrutura de uma instrução <i>if</i> com ramificações <i>else if</i> em Pico-Rust.	10
2.13	Exemplo de um programa em Pico-Rust com a utilização de operadores lógicos.	11
2.14	Estrutura do ciclo <i>while</i> em Pico-Rust.	11
2.15	Exemplo de um programa com recurso ao ciclo <i>while</i> em Pico-Rust.	12
2.16	Exemplo de um programa com recurso ao ciclo <i>while</i> com instruções <i>continue</i> e <i>break</i> em Pico-Rust.	13
2.17	Estrutura de uma declaração de uma função em Pico-Rust.	14
2.18	Exemplo de uma função que soma dois valores e retorna a soma soma em Pico-Rust.	14
2.19	Exemplo de um programa sem a especificação do tipo de retorno em Pico-Rust.	14
2.20	Exemplo de um programa sem a especificação do tipo de retorno omitindo-o em Pico-Rust.	15
2.21	Exemplo de um programa com uma função recursiva em Pico-Rust.	15
2.22	Estrutura de uma declaração de uma <i>struct</i> em Pico-Rust.	16
2.23	Exemplo de um programa com uma declaração de uma <i>struct</i> em Pico-Rust.	16
2.24	Exemplo de um programa com uma instanciação de uma <i>struct</i> e a sua utilização em Pico-Rust.	17
2.25	Estrutura da declaração de um tipo de dados do tipo vetor em Pico-Rust.	17

2.26	Exemplo de um programa com um declaração de um vetor e a sua utilização em Pico-Rust.	17
2.27	Exemplo de um programa com recurso a um ciclo para se iterar um vetor em Pico-Rust.	18
2.28	Exemplo de um programa com utilização da função especial de vetores <code>len</code> em Pico-Rust.	19
2.29	Exemplo de um programa com a utilização do operador de empréstimo em Pico-Rust.	19
2.30	Exemplo de um programa com empréstimos em diferentes contextos em Pico-Rust.	20
2.31	Exemplo de um programa com a declaração de empréstimos mutáveis em Pico-Rust.	20
2.32	Exemplo de um programa com a declaração de empréstimos mutáveis e desreferenciações em Pico-Rust.	21
3.1	Declaração de uma estrutura <code>Point3D</code> em Pico-Rust.	30
3.2	Declaração de um vetor de valores numéricos em Pico-Rust.	31
3.3	Declaração de uma função com três parâmetros numéricos em Pico-Rust.	32
3.4	Código gerado no acesso a um elemento do vetor.	35
D.1	Gramática para definição de instruções globais.	47
D.2	Gramática para definição de tipos.	48
D.3	Gramática para instruções.	48
D.4	Gramática de expressões.	49
D.5	Gramática de operadores.	50

Acrónimos

LR *Left Right*

AST *Abstract Syntax Tree*

BNF *Backus–Naur Form*

UBI Universidade da Beira Interior

DLPC Desenho de Linguagens de Programação e de Compiladores

PAST *Pre-compiled Abstract Syntax Tree*

TAST *Typed Abstract Syntax Tree*

Capítulo

1

Introdução

1.1 Enquadramento

Com os avanços na área da teoria da computação a que se assistiu durante a segunda guerra mundial, rapidamente se denotou a necessidade de produzir programas para os computadores de forma mais intuitiva sem a implementação direta de binário.

Porquanto o conceito foi desenvolvido nos anos 40, apenas nos anos 50 foram criados os primeiros compiladores modernos. Em particular, Grace Hopper em 1952 com um compilador rudimentar da linguagem A-0 [13], e John Backus em 1957 com aquele que é considerado o primeiro compilador completo, destinado à linguagem FORTRAN [10].

Na sequência de inúmeros avanços na área da teoria da computação e da compilação de linguagens de programação, em 2010 foi lançada a linguagem Rust, originalmente desenvolvida em 2006 por Graydon Hoare quando trabalhava na Mozilla [3]. Sendo a primeira versão do compilador foi desenvolvido em OCaml[4].

Com o objetivo de ser uma linguagem que permite criar *software* bastante seguro e robusto para sistemas concorrentes e críticos, várias funcionalidades de verificação formal do código estão implementadas ao nível da compilação, permitindo detetar uma família de problemas que apenas são detetáveis em *runtime* nas linguagens de programação mais comuns [7].

Por conseguinte, o Professor Jean-Christophe Filliâtre, que se inspirou nos trabalhos de Jacques-Henri Jourdan, propôs originalmente um exercício cujo alvo é a criação de um compilador de Rust.

No seguimento deste exercício, foi proposto no âmbito da Unidade Curricular de Linguagens de Programação a implementação de um compilador de

um fragmento de Rust usando OCaml e as bibliotecas Menhir e OCamllex.

Este exercício enquadra-se, portanto, nas origens da linguagem Rust, cujo primeiro compilador foi feito precisamente em OCaml.

1.2 Objetivos

O presente projeto tem como objetivo primário implementar um compilador capaz de produzir código x86-64 para um subconjunto da linguagem de programação Rust, designado de Pico-Rust.

Por seu turno, este compilador, para além das funcionalidades base de um compilador como as instruções *if*, ciclos e operações aritméticas, deve também:

- Suportar tipos de dados simples, como inteiros e booleanos;
- Suportar tipos de dados complexos, nomeadamente vetores e estruturas;
- Suportar o conceito de empréstimos e empréstimos mutáveis, *ownership*;
- Ser compatível com a linguagem de programação original Rust.

1.3 Organização do Documento

De modo a refletir o trabalho que foi feito, este documento encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – apresenta o tema para este trabalho e dá uma contextualização da técnica explorada, apresenta os principais objetivos com a realização do mesmo e mostra a organização deste documento;
2. O segundo capítulo – **Linguagem Pico-Rust** – explora a linguagem Pico-Rust e as suas diversas funcionalidades, tal como uma exemplificação de cada nuance da mesma.
3. O terceiro capítulo – **Implementação do Compilador** – contém a solução do grupo para a implementação de um compilador de Pico-Rust, e estruturação do mesmo e algumas das decisões tomadas.
4. O quarto capítulo – **Conclusões e Trabalho Futuro** – encerra a discussão do trabalho feito, apresentando conclusões relevantes e ideias para trabalhos futuros.

Capítulo

2

Linguagem Pico-Rust

2.1 Introdução

Neste capítulo será abordada a linguagem de programação implementada, Pico-Rust, com recurso a demonstrações em código de cada uma das funcionalidades implementadas, tais como funções, ciclos, recursividade e referências mutáveis.

2.2 Ponto de Partida

Existem linguagens de programação como C e C++ que necessitam de um ponto de partida comum nos programas desenvolvidos, e outras que nada requerem como Python e OCaml. A linguagem de programação Rust, e por consequência Pico-Rust, depende de um ponto de partida a função *main*.

A sua existência é obrigatória em qualquer programa que seja desenvolvido nesta linguagem de programação. Programas como 2.1 não executariam com sucesso enquanto que programas como 2.2 sim.

```
println!(42);
```

Excerto de Código 2.1: Exemplo de um programa incompleto em Pico-Rust.

```
fn main()
{
    println!(42);

    return;
}
```

Excerto de Código 2.2: Exemplo de um programa completo em Pico-Rust.

2.3 Saída

O Pico-Rust suporta duas funções de escrita para o terminal. Estas funções são o `print!` e o `println!`. Estas funções recebem um único argumento, um valor numérico, do tipo `i32`, ou um valor booleano.

Enquanto que a função `print!` imprime o valor sem qualquer adição de caracteres especiais, a função `println!` (**print** com **nova linha**) imprime o valor com a adição de um caracter de quebra de linha, o famoso

n.

O excerto de código 2.3 demonstra a utilização de ambas as funções para imprimir valores numéricos e booleanos.

```
print!(-4);
println!(true);

// Saída esperada:
// -4true
```

Excerto de Código 2.3: Exemplo da utilização das funções de saída `print!` e `println!`.

2.4 Variáveis

A linguagem Pico-Rust suporta variáveis inteiros, `i32`, booleanas, vetores, estruturas, entre outros. O apêndice C demonstra todos os tipos de dados suportados pelo Pico-Rust.

Por definição todas as variáveis declaradas em Pico-Rust são imutáveis, isto significa que o seu valor não poderá ser alterado após a sua declaração.

Caso seja requerida uma variável mutável, esta deverá ser marcada explicitamente como tal. A sintaxe para a declaração de variáveis segue a sintaxe 2.4, a qual mais detalhada no apêndice D.

```
let <mut>? <ident> : <type> = <expr>;
```

Excerto de Código 2.4: Estrutura da declaração de variáveis em Pico-Rust. e println!.

Algo explícito na sintaxe *Backus–Naur Form* (BNF) porém suportado por vezes por outras linguagens de programação, tal como C, C# ou Java, mas que não é suportado pelo Pico-Rust é a inicialização no ato da declaração de variáveis. Em Pico-Rust uma variável necessita obrigatoriamente de ser inicializada no ato da sua declaração. Isto significa que o programa 2.5 não iria ser compilado, resultando num erro na análise semântica do programa.

```
let x : i32;
```

Excerto de Código 2.5: Exemplo de declaração inválida em Pico-Rust.

2.4.1 Mutabilidade

Tal como referido anteriormente, esta linguagem suporta o conceito de mutabilidade. Quando uma variável é marcada como mutável o seu valor pode ser alterado após a sua declaração. Para se marcar uma variável como mutável é apenas necessário recorrer ao uso da palavra reservada **mut** no ato da sua declaração. O excerto de código 2.6 demonstra a declaração e utilização de uma variável mutável.

```
let mut x : i32 = -3;

println!(x); // -3

x = 109;

println!(x); // 109
```

Excerto de Código 2.6: Exemplo de declaração de uma variável mutável em Pico-Rust.

2.4.2 *Shadowing*

O *shadowing* é um conceito comum no paradigma de programação funcional, no qual o Rust se influencia e retira boas práticas. *Shadowing* é o conceito de se esconder uma variável com outra do mesmo nome. O excerto de código 2.7 ilustra um exemplo prático, em que a variável `x`, tipo `i32` é escondida quando a variável `x`, tipo `bool` é declarada. É importante realçar que os tipos em nada influenciam, servindo neste exemplo para simplificar a exposição do conceito.

```
let x : i32 = 65;

println!(x);          // 65

let x : bool = true;

println!(x);          // true
```

Excerto de Código 2.7: Exemplo do efeito de *shadowing* em Pico-Rust.

2.4.3 Operações Aritméticas com Inteiros

Como é normal em qualquer linguagem de programação, é possível efetuar operações numéricas com inteiros, tais como a adição, subtração e divisões. O apêndice B debate a ordem de precedência e a associatividade destes operadores.

A lista completa de operadores suportados pelo Pico-Rust está listada na tabela 2.4.3.

Descrição	Operador
Adição	+
Subtração	−
Divisão	/
Módulo	%
Multiplicação	*

Tabela 2.1: Operadores aritméticos do Pico-Rust.

2.5 Comentários

O Pico-Rust suporta os dois tipos de comentários presentes em Rust. Os famosos comentários de uma linha e de múltiplas linhas. Comentários, por mais simples que pareçam, são cruciais e importantes no ato da programação pois são completamente ignorados pelo compilador, sendo que albergam o propósito de transmitir informação útil para o programador, quer seja a documentação de um excerto de código complexo ou de um ficheiro.

Os excertos de código 2.8 e 2.9 demonstram casos úteis da utilização e do poder dos comentários. É importante notar que não existem regras de como se utilizar comentários em código, sendo um gosto moldado pela experiência.

```
// Um comentario de uma linha

/* Comentario
com
múltiplas
linhas
*/
```

Excerto de Código 2.8: Utilização de comentários em Pico-Rust.

```
let x :i32 = 55;
let y :i32 = 33;

// Divisao de dois numeros
// Atencao: y nao pode ser 0!!!
println!(x / y);
```

Excerto de Código 2.9: Utilização de comentários num programa de Pico-Rust.

2.6 Expressões Booleanas

Para além dos operadores aritméticos descritos na seção 2.4.3, o Pico-Rust suporta operadores relacionais e de lógica. Os quais descritos nas seções seguintes.

2.6.1 Operadores Relacionais

Os operadores relacionais do Pico-Rust, são utilizados para se relacionarem dois valores numéricos do tipo `i32`. A tabela 2.6.1 contém os operadores relacionais suportados pelo Pico-Rust.

Descrição	Operador	Utilização
Igualdade	<code>==</code>	<code>i32 == i32</code>
Desigualdade	<code>!=</code>	<code>i32 != i32</code>
Menor que	<code><</code>	<code>i32 < i32</code>
Menor ou igual que	<code><=</code>	<code>i32 <= i32</code>
Maior que	<code>></code>	<code>i32 > i32</code>
Maior ou igual que	<code>>=</code>	<code>i32 >= i32</code>

Tabela 2.2: Operadores relacionais do Pico-Rust.

2.6.2 Instrução If

As instruções `if` desbloqueiam a habilidade de escolha durante a escrita de um programa por parte do programador, através da verificação de uma condição. Utilizando os operadores relacionais expostos na seção 2.6.1 podem ser escritas as condições utilizadas pelas instruções `if`.

O excerto de código 2.10 demonstra a estrutura de uma instrução `if`. Opcionalmente, a instrução `if` pode ser utilizada em conjunto com a ramificação `else` cuja dinâmica permite a execução de excertos de código diferentes quando determinada condição é avaliada para verde ou falso, este dinamismo é exemplificado no excerto 2.11.

```
if <condicao>
{
    // Bloco de instrucoes a ser
    // executado se a condicao for verdade
}
```

Excerto de Código 2.10: Estrutura de uma instrução *if* em Pico-Rust.

```
if <condicao>
{
    // Bloco de instrucoes a ser
    // executado se a condicao for verdade
}
else
{
    // Bloco de instrucoes a ser
    // executado se a condicao for falsa
}
```

Excerto de Código 2.11: Estrutura de uma instrução *if* com ramo *else* em Pico-Rust.

2.6.3 Instrução If, Else If e Else

Por vezes embatemos em problemas que se subdividem em múltiplas instruções `if` encadeadas, nestes casos é aconselhada a utilização de ramificações `else if`. O Pico-Rust suporta esta noção, a de ramificações encadeadas que possuem uma condições próprias. O excerto de código 2.12 exemplifica um programa com recurso a esta noção.

A utilização de ramificações `else if`, ao contrário de simples instruções `if` encadeadas, resulta em programas mais otimizados. Pois o computador

necessita apenas de avaliar as condições até que uma destas avalie para verdade.

```
let x : i32 = 3;

if x == 0
{
    println!(0);
}
else if x == 1
{
    println!(10);
}
else if x == 2
{
    println!(20);
}
else
{
    println!(100);
}
```

Excerto de Código 2.12: Estrutura de uma instrução *if* com ramificações *else if* em Pico-Rust.

2.6.4 Operadores Lógicos

É possível, construir condições complexas, em que são testadas mais do que uma condição simples, através da utilização de operadores lógicos. Pico-Rust suporta os operadores lógicos 2.6.4, exemplificados no exerto de código 2.13.

Descrição	Operador	Utilização
Igualdade	<code>==</code>	<code>bool && bool</code>
Desigualdade	<code>!=</code>	<code>bool bool</code>
Negação	<code>!</code>	<code>! bool</code>

Tabela 2.3: Operadores lógicos do Pico-Rust.

```
let mut y : i32 = 6;

if 1 > 7 && 2 == 3
{
    y = y + 1;
}
```

Excerto de Código 2.13: Exemplo de um programa em Pico-Rust com a utilização de operadores lógicos.

2.7 Ciclos

Tal como o nome indica, os ciclos são utilizados quando existe a necessidade de repetição de excertos de código. A linguagem Pico-Rust suporta ciclos *while* que podem ser utilizados em conjunção às instruções *break* e *continue*. Estes que permitem um controlo mais rico do fluxo do programa.

Ciclos *while* executam código enquanto uma determinada condição for avaliada a verdade. A sua estrutura simples está exemplificada no excerto de código 2.14.

```
while <condicao>
{
    /*
        Bloco a ser executado
        enquanto a condicao
        for verdade
    */
}
```

Excerto de Código 2.14: Estrutura do ciclo *while* em Pico-Rust.

O excerto de código 2.15 demonstra a aplicação de um ciclo *while* num programa.

```
let mut i : i32= 0;

while i < 5
{
    println!(i);
    i = i + 1;
}

// Saida esperada:
// 0
// 1
// 2
// 3
// 4
```

Excerto de Código 2.15: Exemplo de um programa com recurso ao ciclo *while* em Pico-Rust.

2.7.1 Instruções *Break* e *Continue*

Como referenciado 2.7, as instruções *break* e *continue* estão disponíveis para o programador. Estas permitem um controlo rico do fluxo. Enquanto que o *break* desbloqueia a função de termino de um ciclo, o *continue* permite o salto para a próxima iteração de um ciclo. O excerto 2.16 ilustra uma utilização destas instruções.

```
let x : i32 = 0;

while x < 10
{
    if x == 5
    {
        break;
    }

    if(x % 2 == 0)
    {
        continue;
    }

    println!(x);

    x := x + 1;
}
// Expected Output:
// 1
// 3
```

Excerto de Código 2.16: Exemplo de um programa com recurso ao ciclo *while* com instruções *continue* e *break* em Pico-Rust.

2.8 Funções

O Pico-Rust suporta o conceito de funções, um bloco de instruções com determinadas entradas e uma saída. As funções do Pico-Rust suportam recursividade e chamada por valor.

O excerto de código 2.17 demonstra a declaração de uma função com n argumentos, estrutura presente no apêndice D.

```
fn <id>(<arg1 : type1>, ..., <argn : typen>) -> <return type>
{
    // Corpo da funcao
}
```

Excerto de Código 2.17: Estrutura de uma declaração de uma função em Pico-Rust.

O excerto 2.18 ilustra uma função, de exemplo, que adiciona dois valores numéricos e retorna a sua soma. Esta função recebe dois valores numéricos como parâmetro, *x* e *y*, e retorna um valor numérico.

```
fn add(x : i32, y : i32) -> i32
{
    return x + y;
}
```

Excerto de Código 2.18: Exemplo de uma função que soma dois valores e retorna a soma em Pico-Rust.

Funções que possuam `unit` como tipo de retorno não necessitam de o explicitar, podendo omitir o tipo de dados de retorno. Os excertos de código 2.19 e 2.20 são então sinónimos.

```
fn no_return() -> ()
{
    return;
}
```

Excerto de Código 2.19: Exemplo de um programa sem a especificação do tipo de retorno em Pico-Rust.

```
fn no_return()
{
    return;
}
```

Excerto de Código 2.20: Exemplo de um programa sem a especificação do tipo de retorno omitindo-o em Pico-Rust.

2.8.1 Funções Recursivas

Na área das linguagens de programação, funções recursivas são assim chamadas pois invocam-se a si mesmas, ou seja recursivamente. Pico-Rust não é um estranho neste conceito suportando-o. A declaração de uma função recursiva é idêntica a qualquer outra, sendo a única diferença o fato desta se invocar. O excerto de código 2.21 exemplifica um caso em que se recorre à recursividade para se somar todos os valores positivos no intervalo $[0, n]$.

```
fn sum(n : i32) -> i32
{
    if n <= 0
    {
        return 0;
    }

    return n + sum(n - 1);
}
```

Excerto de Código 2.21: Exemplo de um programa com uma função recursiva em Pico-Rust.

2.9 Estruturas de Dados

Para além dos tipos primitivos, `i32` e `bool`, o Pico-Rust suporta ele também duas estruturas de dados. As `structs` e os `vectors`. Estes serão abordados nas subseções seguintes.

2.9.1 Structs

Estruturas, structs, são coleções de dados que possuem uma razão para serem agrupadas, estes dados podem possuir o mesmo, ou diferentes tipos de dados. Atualmente as estruturas implementas em Pico-Rust estão limitadas ao tipos primitivos C.

A declaração de uma estrutura segue a sintaxe 2.22, exemplificada em 2.23, e pode ser posteriormente utilizada para se declararem instâncias da mesma 2.24. Uma instância de uma estrutura consegue aceder aos seus campos através da utilização do operador `.` na forma `<id da instancia>.<id do campo>`, tal como já exemplificado.

```
struct <ident>{  
  <ident_1>:<type_1>,  
  ...  
  <ident_n>:<type_n>  
}
```

Excerto de Código 2.22: Estrutura de uma declaração de uma struct em Pico-Rust.

```
struct Point3D{  
  x:i32,  
  y:i32,  
  z:i32  
}
```

Excerto de Código 2.23: Exemplo de um programa com uma declaração de uma struct em Pico-Rust.

```
struct Point2D{
    x:i32,
    y:i32

fn add(p: Point2D) -> i32
{
    return p.x + p.y;
}

fn main(){
    let p:Point2D = Point2D{x:55, y:-7};

    println!(add(p));
}
```

Excerto de Código 2.24: Exemplo de um programa com uma instanciação de uma struct e a sua utilização em Pico-Rust.

2.9.2 Vetores

Vetores são uma segunda estrutura de dados que permite a agregação de vários dados de um mesmo tipo. A declaração de uma variável do tipo vetor segue a estrutura 2.25. Ao contrário das structs 2.9.1 o acesso aos elementos desta estrutura não é realizada através de identificadores mas sim de um índice 2.26.

```
let <ident> :Vec< <tipo> > = vec![<expr_1>, ... , <expr_n>];
```

Excerto de Código 2.25: Estrutura da declaração de um tipo de dados do tipo vetor em Pico-Rust.

```
let v :Vec<i32> = vec![5, 6, 99, 105];

println!(v[2]); // 99
```

Excerto de Código 2.26: Exemplo de um programa com um declaração de um vetor e a sua utilização em Pico-Rust.

2.9.2.1 Vetores e Ciclos

Devido ao fato dos elementos de um vetor ser acessado através de um índice, é normal se utilizarem ciclos que se iteram sobre todos, ou parte dos elementos de um vetor. O excerto 2.27 demonstra um programa em que este é o fato.

```
let v :Vec<i32> = vec![5, 6, 99, 105];

let mut i : i32 = 0;

while i < v.len()
{
    println!(v[i]);
    i = i + 1;
}

// Expected Output:
// 5
// 6
// 99
// 105
```

Excerto de Código 2.27: Exemplo de um programa com recurso a um ciclo para se iterar um vetor em Pico-Rust.

2.9.2.2 Funções Especiais de Vetores

A linguagem Pico-Rust possui uma função especial para estas estruturas de dados, a função `len`. Esta função tem como retorno um valor numérico, `i32`, e calcula o tamanho de um vetor. O excerto 2.28 demonstra uma utilização desta função

```
let v :Vec<i32> = vec![5, 6, 99, 105];

let size : i32 = v.len();
println!(size);

// Expected Output:
// 4
```

Excerto de Código 2.28: Exemplo de um programa com utilização da função especial de vetores `len` em Pico-Rust.

2.10 Ownership

O *ownership* é um conceito único e especial do Rust. *Ownership* é o conceito de endereços de memória possuírem um único dono, em C é possível que múltiplos apontadores referenciem o mesmo endereço de memória, em Rust apenas pode haver um único dono de cada vez. Existem dois conceitos importantes no *ownership*, "como é que se deixa de ser dono" e "até quando"?

A utilização do operador de empréstimo `&` retira o título de dono à variável a que este aplicado, como exemplificado no excerto 2.29. Quando uma variável deixa de ser dona esta não pode ser utilizada até que o volte a ser.

```
let x:i32 = 55;

let novo_dono:&i32 = &x; // x nao pode mais ser utilizado
                        // enquanto o novo_dono existir
                        // no seu contexto
```

Excerto de Código 2.29: Exemplo de um programa com a utilização do operador de empréstimo em Pico-Rust.

O *ownership* é verificado por contextos, o que significa que se uma variável perder o título de dono para uma segunda, e esta segunda sair de contexto, a primeira variável voltará a ser a dona. Isto é melhor exemplificado num excerto de código 2.30.

```
let x:i32 = 55;

{
    // x perde o titulo de dono
    let novo dono:&i32 = &x;
}
// x reganha o título de dono pois a variavel novo_dono sai de contexto

println!(x);

// Saida Experada:
// 55
```

Excerto de Código 2.30: Exemplo de um programa com empréstimos em diferentes contextos em Pico-Rust.

2.11 Empréstimos Mutáveis

As referências mutáveis são o equivalente em Rust aos ponteiros do C e do C++. Estes são afetados pelas regras do *ownership* da mesma forma que os empréstimos (&). A declaração de uma referência mutável é semelhante a do empréstimo sendo para isto utilizado o operador &mut. A sintaxe de uma declaração de um empréstimo mutável segue a do excerto 2.31.

```
let mut x:i32 = 55;

let pointer :&mut i32 = &mut x;
```

Excerto de Código 2.31: Exemplo de um programa com a declaração de empréstimos mutáveis em Pico-Rust.

Devido ao fato de uma referência mutável ser um ponteiro para um endereço de memória, é necessário dereferencia-lo, para que se possa aceder ao valor nesse endereço armazenado. O operador de dereferenciação, *, é utilizado nestes casos para este mesmo fim. O excerto de código 2.32 exemplifica uma situação em que tal se sucede.

```
let x:i32 = 55;

let pointer :&mut i32 = &mut x;
*pointer = 44;

println!(*pointer);

// Expected Output.
// 44
```

Excerto de Código 2.32: Exemplo de um programa com a declaração de empréstimos mutáveis e desreferenciações em Pico-Rust.

2.12 Conclusões

Este capítulo abordou de forma detalhada a linguagem Pico-Rust, quer em conceitos mais simples como ciclos e declarações de variáveis como em recursividades e empréstimos. No próximo capítulo será discutida a nossa abordagem e implementação do compilador de Pico-Rust.

Capítulo

3

Implementação do Compilador

3.1 Introdução

Neste capítulo é discutida a implementação do compilador de Pico-Rust, a sua estruturação, funcionalidades e algumas das decisões tomadas durante o seu desenvolvimento.

3.2 Estrutura do Compilador de Pico-Rust

O compilador de Pico-Rust está organizado em seis etapas distintas, que vão desde a tokenização do ficheiro até à geração de código. Estas etapas são:

1. Lexer – Responsável pela tokenização do ficheiro a ser compilado;
2. Parser – Responsável pela geração da *Abstract Syntax Tree* (AST) através dos tokens lidos;
3. Tipagem – Responsável por verificações de tipagem, constrói a *Typed Abstract Syntax Tree* (TAST);
4. Ownership – Verificação das regras de ownership;
5. Pré Compilação – Responsável pelo cálculo dos endereços de memória de cada variável, função e estrutura que culminam na construção da *Pre-compiled Abstract Syntax Tree* (PAST);
6. Compilação – Responsável pela geração de código em *assembly* x86-64.

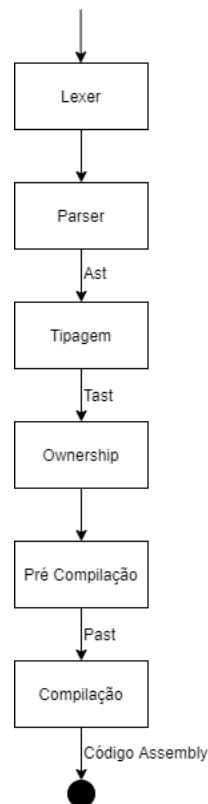


Figura 3.1: Ciclo de vida do compilador de Pico-Rust.

3.3 Analisador Léxico

O analisador léxico deteta os *tokens* que pertencem à nossa linguagem. O analisador léxico utiliza um conjunto de expressões regulares para detetar os diferentes *tokens*, sendo uma expressão regular normalmente cunhada de *rule*.

O analisador léxico utilizado foi o *Ocamllex* que quando confrontado com uma escolha entre duas expressões regulares válidas para um mesmo *token* recorre a duas regras, a expressão regular escolhida é a que contiver o maior *match* e em caso de dúvida é escolhida a que ocorre primeiro.

3.3.1 Símbolos e Regras

As regras que compõe o Pico-Rust vão desde simples símbolos a expressões regulares complexas, estas estão ilustradas nas tabelas 3.3.1 e A e nas seguintes enumerações.

//	/*	*/	espaço em branco
\n	\t	()
{	}	[]
+	-	*	/
%	&		^
<<	>>	<	<=
>	>=	==	!=
	&&	!	.
->	;	,	i32
<i>bool</i>	()		

Tabela 3.1: Símbolos suportados pelo analisador léxico do Pico-Rust.

- HEX_DIGIT -> (0 .. 9) | (a .. f) | (A .. F);
- DEC_DIGIT -> (0 .. 9);
- OCT_DIGIT -> (0 .. 7);
- BIN_DIGIT -> 0 | 1;
- HEX_LITERAL -> (0x | 0X).HEX_DIGIT.(HEX_DIGIT|_)*;
- OCT_LITERAL -> (0o | 0O).OCT_DIGIT.(OCT_DIGIT|_)*;
- BIN_LITERAL -> (0b | 0B).BIN_DIGIT.(BIN_DIGIT|_)*;
- DEC_LITERAL -> DEC_DIGIT.(DEC_DIGIT|_)*;
- INTEGER_LITERAL -> (DEC_LITERAL|BIN_LITERAL|OCT_LITERAL|HEX_LITERAL);
- BOOLEAN_LITERAL -> true | false;
- ALPHA -> (a .. z | A .. Z)
- ID -> ALPHA((_|ALPHA|DEC_DIGIT)*)(!)?.

3.3.2 Erros do Analisador Léxico

O analisador léxico possui apenas um ambiente em que pode ocorrer em erro, quando um símbolo não é aceite por nenhuma das regras deste. Nestes casos o analisador léxico lança uma exceção que irá culminar numa mensagem de erro para informar o utilizador do acontecido 3.2.

```
File "./tests/test.rs", line 10, characters 2-3:  
  
error:  
  
Lexical error: invalid symbol: $.
```

Figura 3.2: Erro provocado por símbolo não reconhecido pelo analisador léxico.

3.4 Analisador de Sintaxe

Como descrito na seção 3.2 o analisador de sintaxe é o segundo passo do processo de compilação do Pico-Rust, este recebe uma lista de *tokens* devolvida pelo analisador léxico e tem a função de verificar se estes constituem produções válidas. Para isto o *parser* utiliza um conjunto de derivações definidas.

O analisador de sintaxe utilizado foi o *Menhir*, um analisador de sintaxe com suporte para a linguagem de programação OCaml do tipo *Left Right* (LR) (*Left to right, Rightmost derivation*), com um *token* de avanço. O *Menhir* é apenas um de muitos *parsers* com suporte a OCaml, sendo outra alternativa bastante popular o *Ocamlyacc*, porém o *Menhir* acabou por ser escolhido devido ao seu amplo suporte pela comunidade e por ser o mais recente de ambos.

3.4.1 Regras do Analisador de Sintaxe

O *parser* necessita de um conjunto de regras definidas pelo programador para conseguir avaliar uma lista de *tokens* como válida ou inválida. O *parser* desenvolvido possui um conjunto de derivações agrupadas em diferentes regras. Estas regras estão listadas abaixo e as suas derivações seguem a gramática BNF presente no apêndice D.

- Instrução Global – Funções e estruturas;
- Instruções – Qualquer instrução que pode ocorrer dentro do contexto de uma função;
- Expressão – Qualquer expressão que possa ser utilizada pelas instruções.

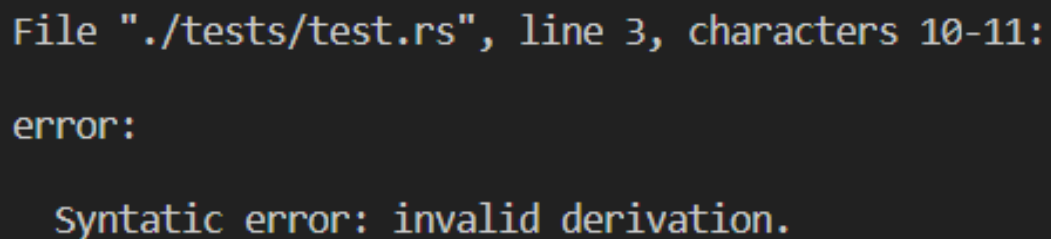
Para além de regras de derivação, são necessárias regras de associatividade e precedência para que o analisador de sintaxe tome a decisão esperada quando confrontado com múltiplas derivações válidas. O apêndice B possui as associatividades e precedências dos operadores escolhidas.

3.4.2 Árvore de Sintaxe Abstrata

O retorno desta etapa do processo de compilação do Pico-Rust é da forma de árvore sendo cada nodo correspondente a uma derivação do *parser*. Este formato de representação do nosso programa não é exclusiva ao Pico-Rust sendo uma prática comum a modificação e geração de árvores com informações diferentes durante o processo de compilação. A AST é a primeira de três estruturas em árvore utilizadas durante o processo de compilação, esta contém as informações da estrutura, ordem e linha em que cada nodo ocorre no programa.

3.4.3 Erros do Analisador Sintático

Tal como o analisador léxico o *parser* também possui mensagens de erro informativas. O *parser* desenvolvido possui uma única mensagem de erro, quando uma sequência de *tokens* não é aceite por nenhuma regra de derivação definida. Neste caso o analisador sintático lança uma exceção que será posteriormente tratada e resultar numa mensagem de erro com as informações de linha e posição para o utilizador 3.3.



```
File "./tests/test.rs", line 3, characters 10-11:  
error:  
  
Syntatic error: invalid derivation.
```

Figura 3.3: Erro provocado por uma derivação não reconhecida pelo analisador sintático.

3.5 Tipagem

Após a geração da AST representativa do programa pelo analisador sintático 3.4, é realizada uma verificação de tipos na mesma que resulta na geração de uma

AST tipada, a TAST. Por ser a primeira verificação profunda da AST do processo de compilação, a tipagem acaba por também tomar responsabilidades secundárias para além da verificação da utilização correta dos tipos. Estas responsabilidades são:

- Verificação da utilização correta dos tipos;
- Verificação do acesso itens válidos (variáveis/funções/estruturas).

A verificação da utilização correta dos tipos utiliza as árvores de tipo presentes no enunciado deste projeto [6], estas estão presentes no apêndice E.

3.5.1 Árvore de Sintaxe Abstrata Tipada

A análise de tipos extrai informações do programa que são necessárias em passos posteriores, para que não seja necessário a recalculação dos tipos de cada expressão e instrução é gerada uma TAST, uma estrutura de árvore semelhante à AST mas substituindo a informação das linhas de cada nodo pelo seu tipo.

3.5.2 Erros da Análise de Tipos

O analisador de tipos, À semelhança do analisador léxico e do analisador sintático também possui mensagens de erro informativas únicas a esta etapa. O analisador de tipos é a fase da compilação que contém a maior fatia das mensagens de erro presentes no Pico-Rust, sendo que vão desde erros informativos da utilização de variáveis inexistentes à utilização errada de tipos em todo o tipo de instruções e expressões. A figura 3.4 ilustra apenas um destes vários casos.

```
File "./tests/test.rs", line 6:  
error:  
  
Typing analysis:  
Wrong type in the declaration of variable x, was given Tbool but a Ti32 was expected.
```

Figura 3.4: Erro provocado pela utilização inválida dos tipos.

3.6 Analisador de *Ownership*

Tal como foi descrito no capítulo 2 o conceito de *ownership* é especial ao Rust e tem como ideia principal a da existência de um único dono. O Pico-Rust

possui uma fase de compilação exclusiva à verificação da boa utilização deste conceito.

Para se desenvolver um analisador deste tipo é necessário identificar em que momentos uma variável **perde** e **ganha** controlo de algo, *ownership*. Na linguagem Pico-Rust existe apenas um momento em que uma variável perde o *ownership*, quando esta é emprestada a outra, quer por empréstimo normal quer por empréstimo mutável. Uma variável reganha o *ownership* quando o atual dono do *ownership* sai de contexto 2.30.

Sendo assim, para verificarmos a correta utilização das variáveis, necessitamos apenas de realizar duas operações. A primeira é a de invalidar uma variável num contexto quando esta é emprestada. A segunda é a de guardar uma cópia do estado dos contextos ao se entrar num novo, para que possamos restaurar o estado do *ownership* ao sairmos deste novo contexto.

3.6.1 Erros da Análise de *Ownership*

Como é comum a todas as fases de compilação do Pico-Rust, a análise do *ownership* gera ela também um conjunto de mensagens de erro informativas. Quando é detetada a utilização ilegal de um identificador o *ownership* lança uma exceção, que será tratada e que irá resultar na mensagem de erro para o utilizador 3.5.

```
File "./tests/test.rs\n
error:

Ownership error:
Invalid use of the variable x, it's not the current owner.
```

Figura 3.5: Erro provocado pelo analisador de *ownership* do Pico-Rust.

3.7 Pré-Compilação

A pré-compilação é a quinta fase do processo de compilação do Pico-Rust e é responsável pela designação dos espaços de memória de cada variável e da geração das tabelas de posições das estruturas, vetores e funções. Estes cálculos irão culminar na criação de uma PAST, a última estrutura em árvore antes do processo de geração de código. A fase de pré-compilação tem como entrada a TAST construída na terceira etapa 3.5.

Para se determinar em que posição será associada uma variável de Pico-Rust é utilizada uma variável que contém a última posição livre na pilha. Esta variável é incrementada, pelo tamanho necessário para albergar todo o tipo de dados, sempre que se é realizada uma declaração. O apêndice C contém uma tabela com as dimensões de cada tipo de dados.

3.7.1 Tabelas Relacionais

Tal como descrito, um dos objetivos da pré-compilação é o da criação de tabelas de posições relacionais temporárias para suporte à função principal de calcular em que posição da pilha cada variável será armazenada. Existem três estruturas que necessitam da criação de uma tabela de posições, estruturas, vetores e funções.

3.7.1.1 Estruturas

A tabela de posições de uma estrutura armazena a distância de um campo da estrutura ao endereço base. Neste compilador de Pico-Rust o endereço base é o primeiro endereço livre. Ou seja, para a estrutura 3.1 seria gerada a tabela de posições relativas 3.6.

```
struct Point3D{  
    x:i32,  
    y:i32,  
    z:i32  
}
```

Excerto de Código 3.1: Declaração de uma estrutura Point3D em Pico-Rust.

Point3D	
x	0
y	8
z	16

Figura 3.6: Tabela de suporte das posições relativas de uma estrutura na fase de pré-compilação.

Esta tabela é então utilizada para se conhecer as posições em memória de cada campo da estrutura. Caso queiramos aceder ao campo `z` da estrutura 3.1 a sua posição em memória seria a posição em memória da variável + 16.

3.7.1.2 Vetores

Ao contrário das estruturas, não é necessária uma tabela de posições relativas auxiliar para vetores. Isto pois todos os elementos de um vetor possuem o mesmo tipo, logo a mesma dimensão. Desta forma é apenas necessário conhecermos, a posição inicial do vetor, o seu tipo e o índice do elemento a aceder. A partir destas três informações é possível calcularmos a posição de qualquer elemento de um vetor utilizando a equação 3.1.

$$\text{posElemento} = \text{pos} + \text{índice} * \text{SizeOf}(\text{type}) \quad (3.1)$$

De forma ilustrativa, o vetor declarado no excerto 3.2 teria os seus elementos nas posições relativas descritas no diagrama 3.7 à base.

```
let v : Vec<i32> = vec![5, 55, -99, 108];
```

Excerto de Código 3.2: Declaração de um vetor de valores numéricos em Pico-Rust.

Pos. Relativa	0	8	16	24
índice	0	1	2	3

Figura 3.7: Posições relativas dos elementos de um vetor face à posição inicial em Pico-Rust.

3.7.1.3 Funções

Devido ao fato das funções poderem possuir parâmetros de diferentes dimensões, estas possuem uma tabela auxiliar de posições relativas ao registo *rbp*. A qual é utilizada para se obter a posição na memória desse mesmo parâmetro. Algo notório é o de que as posições dos parâmetros não iniciam-se na posição 0 tal como as estruturas e os vetores, mas sim na posição 16 devido ao fato de armazenarmos a posição da *frame* e do sistema de chamada de funções do *assembly*.

Uma função com três parâmetros como a exemplificada no excerto 3.3 irá possuir a tabela de posições 3.8.

```
fn add(x:i32, y:i32, z:i32) -> i32
{
    return x + y + z;
}
```

Excerto de Código 3.3: Declaração de uma função com três parâmetros numéricos em Pico-Rust.

	add
x	16
y	24
z	32

Figura 3.8: Tabela de suporte das posições relativas dos parâmetros de uma função em Pico-Rust.

3.7.2 Árvore de Sintaxe Abstrata Pré-Compilada

Tal como referido o retorno da pré-compilação do Pico-Rust é da forma de uma árvore, cuja construção provém da análise da TAST gerada pela análise de tipos 3.5. Esta árvore, possui a informação adicional das posições da pilha de cada variável, tipo de dados complexos e funções. Esta é a última das árvores representativas do programa no processo de compilação do Pico-Rust e serve de entrada para a etapa de geração de código *assembly*.

3.8 Compilação

A fase de compilação é a última fase de uma cadeia de etapas do processo de compilação do compilador desenvolvido de Pico-Rust. Este passo é responsável pela geração do código *assembly* e utiliza a informação contida na PAST para tal.

3.8.1 Expressões

A compilação das expressões, constantes, acesso a variáveis, etc. é uma peça crucial e que influencia como um todo todas as outras partes da geração de código. Como uma expressão apenas é calculada quando o seu valor é necessário, a abordagem tomada foi a de colocar os valores obtidos de uma expressão no topo da pilha de forma a que possam ser acedidos de forma imediata.

3.8.1.1 Constantes

O exemplo mais simples é o de uma constante, uma expressão com um valor cru. Neste caso colocamos o valor no registo *rax* e apenas fazemos *push* do seu valor para a pilha. A figura 3.9 ilustra o estado da pilha após a execução do código gerado para a expressão constante 42.

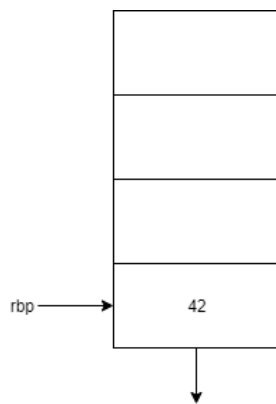


Figura 3.9: Estado da pilha após a execução do código gerado para as constantes.

3.8.1.2 Declaração de Tipos Complexos

A declaração de tipos complexos, estruturas e vetores, apresenta um problema diferente à da geração de código para uma simples constante, isto pois agora podemos possuir mais do que uma expressão para armazenar na pilha de uma vez.

Nas estruturas, os valores são dispostos em espaços de memória contíguos, isto devido ao fato de na fase de pré-compilação termos gerado tabelas de posições relativas. Após a declaração de uma instância da estrutura 3.1 a pilha fica num estado semelhante a 3.10.

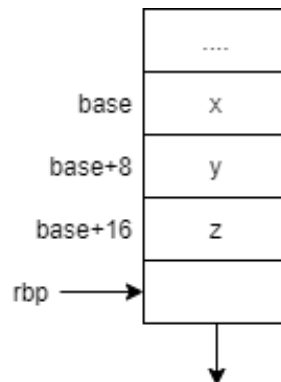


Figura 3.10: Estado da pilha após a declaração de uma estrutura.

Nos vetores, a ordem das expressões compiladas é invertida, ficando o primeiro elemento do vetor no topo. Esta ordem foi decidida pois é possível calcular uma posição relativa a um registo através de uma posição, índice e escala em *assembly* 3.4. A figura 3.11 ilustra o estado da pilha após a declaração do vetor 3.2.

```
movq -24(%rbp,%rax,8), %rax

// Posicao do primeiro elemento: -24
//   em relacao ao registo rbp
// Escala: 8
// Indice: Valor armazenado em %rax
```

Excerto de Código 3.4: Código gerado no acesso a um elemento do vetor.

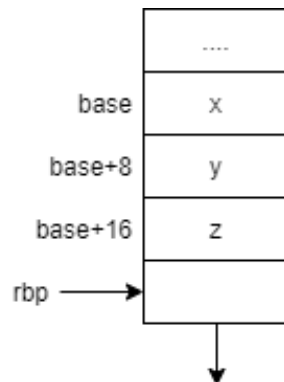


Figura 3.11: Estado da pilha após a declaração de um vetor.

3.8.1.3 Chamada de Funções

A chamada de funções pode ser separada em dois momentos, a passagem dos argumentos e a criação de uma nova *frame*. Utilizamos a convenção de chamada de funções do C [5] em que os argumentos são colocados na pilha na ordem inversa, para que o primeiro argumento fique no topo da pilha no momento imediatamente antes da chamada da função em *assembly*. Após a colocação dos argumentos é guardada a posição da *frame* anterior e é criada uma nova. Ficando a pilha num estado semelhante a 3.12.

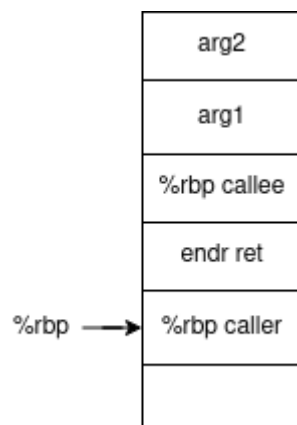


Figura 3.12: Estado da pilha após a chamada de uma função.

3.9 Conclusões

Este capítulo abordou de forma detalhada a nossa implementação do compilador de Pico-Rust e as suas diferentes fases. No próximo capítulo serão realizadas as conclusões finais do projeto tal como possível trabalho futuro.

Capítulo

4

Conclusões e Trabalho Futuro

4.1 Conclusões Principais

O desenvolvimento de um compilador é um processo demoroso e exige um planeamento mais rigoroso do que outros tipos de *software*, pois decisões tomadas em qualquer uma das fases de compilação poderão limitar futuras fases caso mal estruturadas.

O trabalho desenvolvido, mesmo que não contendo todas as funcionalidades que queríamos implementar, é uma reflexão natural da evolução face ao compilador desenvolvido na unidade curricular Desenho de Linguagens de Programação e de Compiladores (DLPC) da licenciatura em Engenharia Informática na Universidade da Beira Interior (UBI) [2]. A divisão do processo de compilação em múltiplas fases 3.2 e a utilização de diferentes representações em árvore do nosso programa permitiu um desenvolvimento menos penoso e mais facilitado.

Outro fator positivo, é a da utilização de normas de compilação, a chamada de funções, declaração de funções e a de vetores, seguem todos estes as normas de compilação do C.

4.2 Trabalho Futuro

Infelizmente, o compilador não alberga todas as funcionalidades que queríamos para o mesmo. A nossa inexperiência em *assembly* combinada com toda a carga de trabalho do primeiro semestre nestes últimos dois meses limitou os horizontes que este compilador poderia ter alcançado. As funcionalidades por implementar culminam todas elas num mesmo problema comum, o acesso indireto a espaços de memória. Estes são:

- Retorno de tipos complexos de funções – Retornar estruturas ou vetores de funções;
- Passagem por referência – Passagem por referência nas funções;
- Tipos complexos de tipos complexos – Por exemplo, vetores de vetores ou de estruturas;
- Retorno de referências – Retornar referências de funções.

Outras funcionalidades que teriam acrescentado ao nosso trabalho seriam:

- Utilização de técnicas de otimização – Técnicas como o *graph coloring* [1];
- Interatividade com o utilizador – Funções de entrada.

Foram pensadas em possíveis soluções para estes problemas que infelizmente não foram implementados.

4.2.1 Retorno de Tipos Complexos

O retorno de um tipo primitivo é trivial pois necessita apenas da utilização do mecanismo de retorno de funções em *assembly*. Devido ao fato dos tipos complexos poderem possuir mais do que um valor a utilização direta deste mecanismo não seria suficiente para transportar todos os valores, sendo assim, uma solução seria a de retornar o espaço de memória base do tipo complexo e a partir deste extrair todos os outros valores.

4.2.2 Passagem por Referência

O problema aqui presente, é o mesmo do retorno de tipos complexos, o acesso indireto à memória. Uma solução seria a da passagem do espaço de memória em vez do valor como argumento, e posteriormente a desreferenciação para podermos aceder ao valor neste armazenado.

4.2.3 Retorno de Referências

A solução para o retorno de referências é o mesmo da passagem por referência 4.2.1, o retorno dum endereço de memória ao contrário de um simples valor.

Apêndice

A

Palavras Reservadas

Tal como é normal em qualquer linguagem de programação o Rust, e por consequência o Pico-Rust, possui uma lista de palavras reservadas. Estas palavras não podem ser utilizadas como identificadores, por exemplo na declaração de estruturas/funções/variáveis. A tabela A possui a lista de palavras reservadas do Pico-Rust.

break	continue	else	false
fn	i32	if	len
let	mut	print!	println!
return	while		

Tabela A.1: Palavras reservadas do Pico-Rust.

Apêndice

B

Associatividade e Precedência dos Operadores

Os operadores do Pico-Rust possuem uma ordem de associatividade e precedência entre si. A tabela B demonstra as regras de associatividade e os níveis de precedência de cada operador desta linguagem.

Operador	Associatividade	Precedência
=	direita	mais fraco
	esquerda	
&&	esquerda	
==, !=, <, <=, >, >=	—	
+, -	esquerda	
!, *, -, &, &mut	—	
[]	—	
.	—	mais forte

Tabela B.1: Associatividade e precedência dos operadores do Pico-Rust.

Apêndice

C

Tipos de Dados

O Pico-Rust oferece diferentes tipos de dados, quer primitivos quer complexos. As tabelas C e C ilustram os tipos de dados e os seus respectivos tamanhos.

Tipo	Exemplificação
i32	Inteiros de 32 bit
bool	Booleanos
Vec<T>	Vetores do tipo T
struct	Coleção de vários tipos primitivos
()	Unit
&	Empréstimo
&mut	Empréstimo mutável

Tabela C.1: Tipos de dados suportados pelo Pico-Rust.

Tipo	Exemplificação
i32	8
bool	8
Vec<T>	$tamanho * sizeof(T)$
struct	soma dos tamanhos dos campos
()	0
&	Tamanho do tipo a emprestar
&mut	Tamanho do tipo a emprestar

Tabela C.2: Tamanhos dos tipos de dados suportados pelo Pico-Rust.

Apêndice

D

Gramática BNF

D.1 Instruções Globais

```
<file>          ::= <decl>* EOF
<global_stmt> ::=
    | <decl_struct>
    | <decl_fun>
<decl_struct> ::= struct <ident> { (<ident> : <type>)*, }
<decl_fun>     ::= fn <ident> ( (mut? <ident> : <type>)*, ) (
    ↪ -> <type>)? <block>
```

Excerto de Código D.1: Gramática BNF do Pico-Rust para definição de instruções globais.

D.2 Tipos

```
<type> ::= <ident>
        | <ident> < <type> >
        | bool
        | i32
        | Vec < <type> >
        | & <type>
        | & mut <type>
        | mut <type>
```

Excerto de Código D.2: Gramática BNF dos tipos de Pico-Rust.

D.3 Instruções Globais

```
<block> ::= { <stmt>* }
<stmt>  ::= <block>
        | ;
        | return <expr>? ;
        | break ;
        | continue ;
        | let mut? <ident> : <type> = <expr> ;
        | <ident> = <expr> ;
        | * <ident> = <expr> ;
        | println! ( <expr> ) ;
        | print! ( <expr> ) ;
        | <expr> ;
        | <while> <expr> <stmt>
        | if <expr> <block> <elif>
<elif>  ::=
        | else if <expr> <block>
        | else <block>
```

Excerto de Código D.3: Gramática BNF de instruções Pico-Rust.

D.4 Expressões

```
<expr> ::=  
    | <integer>  
    | true  
    | false  
    | <ident>  
    | <expr> <binop> <expr>  
    | <unop> <expr>  
    | <ident> . <ident>  
    | <ident> . len ( )  
    | <ident> ( <expr>*, )  
    | <ident> [ <expr> ]  
    | vec! [ <expr>,* ]  
    | ( <expr> )  
    | & <ident>  
    | &mut <ident>  
    | * <ident>
```

Excerto de Código D.4: Gramática BNF de expressões do Pico-Rust.

D.5 Operadores

```
<unop> ::= - | !  
<binop> ::= +  
          | *  
          | /  
          | \%  
          | ==  
          | !=  
          | >  
          | <  
          | >=  
          | <=  
          | &&  
          | ||
```

Excerto de Código D.5: Gramática BNF de operadores do Pico-Rust.

Apêndice

E

Árvores de Tipos

E.1 Expressões

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{e\} : \tau} \quad (\text{E.1})$$

E.1.1 Constantes

$$\frac{\text{constante de tipo } \tau}{\Gamma \vdash c : \tau} \quad (\text{E.2})$$

E.1.2 Variáveis

$$\frac{m \ x : \tau \in \Gamma}{\Gamma \vdash_l x : \tau} \quad (\text{E.3})$$

E.1.3 Expressões

$$\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash e : \tau} \quad (\text{E.4})$$

E.1.4 Atribuição

$$\frac{\Gamma \vdash_l e_1 : \tau_1 \quad \Gamma \vdash_{\text{mut}} e_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \leq \tau_1}{\Gamma \vdash e_1 = e_2 : ()} \quad (\text{E.5})$$

E.2 Operações

E.2.1 Operadores Binários

$$\frac{\Gamma \vdash e_1 : \text{i32} \quad \Gamma \vdash e_2 : \text{i32} \quad op \in \{==, !=, <, <=, >, >=\}}{\Gamma \vdash e_1 \text{ op } e_2 : \text{bool}} \quad (\text{E.6})$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \quad op \in \{||, \&\&\}}{\Gamma \vdash e_1 \text{ op } e_2 : \text{bool}} \quad (\text{E.7})$$

E.2.2 Operadores Unários

$$\frac{\Gamma \vdash e : \text{i32}}{\Gamma \vdash -e : \text{i32}} \quad (\text{E.8})$$

$$\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash !e : \text{bool}} \quad (\text{E.9})$$

$$\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash \&e : \&\tau} \quad (\text{E.10})$$

$$\frac{\Gamma \vdash_l e : \tau \quad \Gamma \vdash_{\text{mut}} e}{\Gamma \vdash \&\text{mut } e : \&\text{mut } \tau} \quad (\text{E.11})$$

$$\frac{\Gamma \vdash e : \&m \tau}{\Gamma \vdash_l *e : \tau} \quad (\text{E.12})$$

E.3 Instruções

E.3.1 Retorno

$$\overline{\Gamma \vdash \text{return} : ()} \quad (\text{E.13})$$

$$\frac{\Gamma \vdash e : \tau_r}{\Gamma \vdash \text{return } e : ()} \quad (\text{E.14})$$

E.3.2 Vetores

E.3.2.1 Declaração

$$\frac{\forall i. \Gamma \vdash e_i : \tau}{\Gamma \vdash \text{vec}![e_1, \dots, e_n] : \text{Vec} <\tau>} \quad (\text{E.15})$$

E.3.2.2 Tamanho

$$\frac{\Gamma \vdash e : \text{Vec} \langle \tau \rangle}{\Gamma \vdash e.\text{len}() : \text{i32}} \quad (\text{E.16})$$

E.3.2.3 Acesso

$$\frac{\Gamma \vdash_l e_1 : \text{Vec} \langle \tau \rangle \quad \Gamma \vdash e_2 : \text{i32}}{\Gamma \vdash_l e_1[e_n] : \tau} \quad (\text{E.17})$$

E.3.3 Estruturas**E.3.3.1 Declaração**

$$\frac{\text{struct } S\{x_1 : \tau, \dots, x_n : \tau_n\} \in \Gamma \quad \pi \text{ uma permutação} \quad \forall i. \Gamma \vdash e_i : \tau_i}{\Gamma \vdash S\{x_1 : \tau, \dots, x_n : \tau_n\} : S} \quad (\text{E.18})$$

E.3.3.2 Acesso

$$\frac{\Gamma \vdash_l e : S \quad \text{struct } S\{x : \tau\} \in \Gamma}{\Gamma \vdash_l e.x : \tau} \quad (\text{E.19})$$

E.3.4 if else

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ } e_1 \text{ else } e_2 : \tau} \quad (\text{E.20})$$

E.3.5 while

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : ()}{\Gamma \vdash \text{while } e \text{ } e_1 : ()} \quad (\text{E.21})$$

E.3.6 Bloco de Instruções

$$\overline{\Gamma \vdash \{\} : ()} \quad (\text{E.22})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{e\} : \tau} \quad (\text{E.23})$$

$$\frac{\Gamma \vdash b : \tau}{\Gamma \vdash \{e ; b\} : \tau} \quad (\text{E.24})$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, m \ x : \tau \vdash \{b\} : \tau'}{\Gamma \vdash \{ \text{let } m \ x = e ; b \} : \tau'} \quad (\text{E.25})$$

$$\frac{e \neq \text{let} \quad \Gamma \vdash e : \tau \quad \Gamma \vdash \{b\} : \tau'}{\Gamma \vdash \{e ; b\} : \tau'} \quad (\text{E.26})$$

E.4 Instruções Globais

E.4.1 Declaração de Funções

$$\frac{f(\tau'_1, \dots, \tau'_n) \rightarrow \tau \in \Gamma \quad \forall i. \Gamma \vdash e_i : \tau_i \quad \tau_i \leq \tau'_i}{\Gamma \vdash f(e_1, \dots, e_n) : \tau} \quad (\text{E.27})$$

E.4.2 Auto-Dereferenciação

$$\frac{\Gamma \vdash e : \&m \text{Vec} \langle \tau \rangle}{\Gamma \vdash e.\text{len}() : \text{i32}} \quad (\text{E.28})$$

$$\frac{\Gamma \vdash e : \&m \text{Vec} \langle \tau \rangle \quad \Gamma \vdash e_2 : \text{i32}}{\Gamma \vdash_l e[e_2] : \tau} \quad (\text{E.29})$$

$$\frac{\Gamma \vdash e : \&m S \quad \text{struct } S\{x : \tau\} \in \Gamma}{\Gamma \vdash_l e.x : \tau} \quad (\text{E.30})$$

E.5 Mutabilidade

$$\frac{\text{mut } x : \tau \in \Gamma}{\Gamma \vdash_{\text{mut}} x} \quad (\text{E.31})$$

$$\frac{\Gamma \vdash_{\text{mut}} e}{\Gamma \vdash_{\text{mut}} e[e_2]} \quad (\text{E.32})$$

$$\frac{\Gamma \vdash_{\text{mut}} e}{\Gamma \vdash_{\text{mut}} e.x} \quad (\text{E.33})$$

$$\frac{\Gamma \vdash e : \&\text{mut } \tau}{\Gamma \vdash_{\text{mut}} *e} \quad (\text{E.34})$$

E.6 Declaração de uma Estrutura

$$\frac{\forall i. \Gamma, \text{struct } S\{x_1 : \tau_1, \dots, x_n : \tau_n\} \vdash \tau_i \quad \tau_i \text{ não contém empréstimos}}{\Gamma \vdash \text{struct } S\{x_1 : \tau_1, \dots, x_n : \tau_n\} \Rightarrow \{\text{struct } S\{x_1 : \tau_1, \dots, x_n : \tau_n\}\} \cup \Gamma} \quad (\text{E.35})$$

E.7 Declaração de uma Função

$$\begin{array}{c}
 \Gamma \vdash \tau_r \quad \tau_r \text{ não contém empréstimos} \quad \forall i. \Gamma, \vdash \tau_i \\
 \{f(\tau_1, \dots, \tau_n) \rightarrow \tau_r, x_1 : \tau_1, \dots, x_n : \tau_n\} \cup \Gamma \vdash \{b\} : \tau \\
 \tau = () \text{ e a avaliação de } b \text{ conduz sempre a um return ou então a } \tau = \tau_r \\
 \hline
 \Gamma \vdash \text{fnf}(x_1 : \tau_1, \dots, x_n : \tau_n) \rightarrow \tau_r \quad b \Rightarrow \{f(\tau_1, \dots, \tau_n) \rightarrow \tau_r\} \cup \Gamma
 \end{array}
 \quad (E.36)$$

E.8 Ficheiros

$$\overline{\Gamma \vdash_f \emptyset} \quad (E.37)$$

$$\frac{\Gamma \vdash d_1 \Rightarrow \Gamma' \quad \Gamma' \vdash_f d_2 \dots d_n}{\Gamma \vdash_f d_1 \quad d_2 \dots d_n} \quad (E.38)$$

Bibliografia

- [1] Graph coloring. [Online] https://en.wikipedia.org/wiki/Graph_coloring. Último acesso a 19 de janeiro de 2021.
- [2] Natrix Language Documentation. [Online] <https://dario-santos.github.io/2020/01/12/natrixdoc.html>. Último acesso a 19 de janeiro de 2021.
- [3] Rust-lang.org. [Online] <https://web.archive.org/web/20160609195720/https://www.rust-lang.org/faq.html#project>. Último acesso a 20 de janeiro de 2021.
- [4] Rust-lang.org. [Online] <https://web.archive.org/web/20140815054745/http://blog.mozilla.org/graydon/2010/10/02/rust-progress/>. Último acesso a 20 de janeiro de 2021.
- [5] The 64 bit x86 C Calling Convention. [Online] <https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf>. Último acesso a 17 de janeiro de 2021.
- [6] Trabalho de Compilação Pico Rust. [Online] http://www.di.ubi.pt/~desousa/DLPC/trab_dlpc_2021.pdf. Último acesso a 18 de janeiro de 2021.
- [7] Abel Avram. Interview on Rust, a Systems Programming Language Developed by Mozilla. [Online] <https://www.infoq.com/news/2012/08/Interview-Rust/>. Último acesso a 20 de janeiro de 2021.
- [8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.
- [9] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, new ed edition, July 2004.
- [10] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. *The FORTRAN Automatic Coding System*. IRE-AIEE-ACM '57

- (Western). Association for Computing Machinery, New York, NY, USA, 1957.
- [11] Randal Bryant and David O'Hallaron. x86-64 machine-level programming. April 2011.
- [12] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.
- [13] Grace Murray Hopper. The education of a computer. New York, NY, USA, 1952. Association for Computing Machinery.
- [14] S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.
- [15] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.