CRAL

CiberRato Agent Language

Linguagem para definição e manipulação de robots

Linguagens Formais e Autómatos

Eurico Dias, Pedro Valério, Daniel Correia,
Rodrigo Rosmaninho, Rita Amante

CRAL

CiberRato Agent Language

Linguagem para definição e manipulação de robots

Linguagens Formais e Autómatos

Departamento de Eletrónica, Telecomunicações e Informática

Eurico Dias, Pedro Valério, Daniel Correia, Rodrigo Rosmaninho, Rita Amante

(72783) dias.eurico@ua.pt, (88734) pedrovalerio@ua.pt, (88753) dcorreia@ua.pt, (88802) r.rosmaninho@ua.pt, (89264) rita.amante@ua.pt

Julho 2019

Conteúdo

Capítulo 1

Objetivos

O principal objetivo deste projeto é desenvolver uma linguagem que facilite a programação de robots, com especial foco na programação de agentes virtuais no ambiente de simulação do *Ciber Rato*.

O *Ciber Rato* é uma Modalidade de Simulação do Concurso Micro-Rato da Universidade de Aveiro e do seu IEEE Student Branch, que também é utilizada como atividade na Academia de Verão para estudantes do ensino secundário. Esta modalidade é suportada por um ambiente de simulação de robots e labirintos.

Aos concorrentes é proposto o desafio de construir uma simulação de um robot móvel e autónomo que resolva um labirinto desconhecido, no sentido de localizar e alcançar um farol nele colocado e imobilizar-se numa "zona de chegada". Uma vez alcançado o primeiro objetivo o robot deverá regressar ao seu ponto de partida. Durante o percurso os robots encontram obstáculos fixos - paredes - e obstáculos móveis - outros robots - com os quais não podem colidir, sob pena de serem penalizados.

São disponibilizadas bibliotecas para interagir com a simulação e controlar os robots em **Python**, **Java**, e **C++**. No entanto, os participantes que frequentam o ensino secundário mostram alguma dificuldade a implementar os algoritmos devido ao facto de ser, para muitos, o primeiro contacto com programação em geral.

Como tal, surge a necessidade de uma nova linguagem que simplifique o processo. Os objetivos principais são:

- Simplificar a sintaxe em geral;
- Permitir a utilização de intervalos, por exemplo "[3, 5]"ou "3 < x < 7";
- Apresentar mensagens de erro muito informativas;
- Disponibilizar funcionalidades específicas à robótica;
- Permitir a configuração de vários aspetos do compilador de forma fácil para acomodar alterações ao concurso.

Capítulo 2

Instruções de Utilização

Para executar o processo de compilação, é necessário correr os seguintes comandos a partir da raiz do repositório:

```
javac lib/*.java language/*.java config/*.java
java -ea language/cralMain $nome_do_ficheiro_cral $nome_do_ficheiro_config

# Exemplo com ficheiros reais
java -ea language/cralMain tests/a0.cral config/example.config
```

O ficheiro compilado irá ser criado na raiz do repositório com o nome **Output.cpp**

No diretório **tests** existem alguns ficheiros de teste das funcionalidades base da linguagem e o ficheiro **a0.cral**, que consiste numa implementação em *CRAL* do ficheiro **a0.cpp** que foi fornecido com as bibliotecas de interação com o simulador e, como tal, tem o mesmo comportamento e pode ser testado com o simulador.

No diretório **config** existem dois ficheiros exemplo de configuração que possibilitam a interação com a biblioteca **CiberAV**. Desses, o **example.config** é indicado para testes que interajam com o simulador, e o **example_no_init.config** para restantes testes de funcionalidade básica da linguagem.

Capítulo 3

Linguagem

CRAL é uma linguagem *domain-specific* compilada cuja linguagem destino é **C++**, de forma a tirar partido das bibliotecas existentes para interagir com a simulação do *Ciber Rato*. A análise sintática e semântica está implementada em **ANTLR4**.

Exemplo de um programa muito simples implementado em *CRAL*:

```
cond left_of_start = call startAngle() < -5
cond right_of_start = call startAngle() > 5
cond at_start = call startDistance() < 50

behav gotoStart() until at_start:
    # going to target, adjusting orientation if necessary
    when:
        left_of_start: call motors(left=80, right=70)
        right_of_start: call motors(left=70, right=80)
        other: call motors(left=80, right=80)
    end
end

main():
    # connecting to server
    name = "Grimmy Bear"

    set goToStart

    # ending the program
    call print("Bye!")
end
```

3.1 Instruções

Em *CRAL* existem vários tipos de instruções.

3.1.1 Instruções de uma linha

Algumas instruções, como, por exemplo, declarações e atribuições, ocupam apenas uma linha.

```
bool verdadeiro          # Declaração
verdadeiro = true;       # Atribuição - ponto e virgula (';') é opcional
num numero = 1999        # Declaração e Atribuição simultaneamente
```

No entanto, é possível condensar várias instruções numa única linha, desde que seja usado o separador '**;**'

```
bool verdadeiro; verdadeiro = true; num numero = 1999
```

É permitido a utilização de '**;**' para marcar o fim de uma instrução, embora não seja necessário se a instrução for a única na linha onde foi escrita.

3.1.2 Instruções que podem gerar um bloco

Certas instruções, como as de controlo de fluxo, podem gerar um bloco de instruções.

No corpo dos blocos pode estar qualquer instrução exceto a definição de uma função ou behaviour. Podem também existir blocos adicionais dentro de um bloco.

Um bloco é iniciado pelo caráter '**:**' e terminado por '**end**'.

```
call print("fora do bloco")
while true:
    call print("dentro do bloco")
    until x > 3:
        call print("bloco ao quadrado :D")
    end
end
```

3.1.3 Comentários

O código pode ser comentado para efeitos de documentação. Os comentários não são executados e, como tal, é permitida a escrita livre.

Para iniciar um comentário de uma linha utiliza-se o caráter '#', após o qual todo o texto até à próxima linha é ignorado pelo processo de compilação

Para escrever um comentário multi-linha delimita-se o texto pretendido pelos carateres '#*****' e '*****#', respetivamente.

```
# Comentário de uma linha
call print("olá")
call print("hello") # Outro Comentário de uma linha
call print("bonjour")

#*
linha 1
linha 2
etc
*#

call print("adeus") #* ou apenas uma *#
```

3.1.4 Calls

Calls são funções disponíveis para utilizar e definidas no ficheiro de configuração, que permitem ao programador de *CRAL* interagir com as funções da(s) biblioteca(s) de interação com a simulação/robots.

O compilador fornecido aos programadores possui listado no ficheiro de configuração as bibliotecas a importar e as funções que estão disponíveis para utilizar, tal como o seu tipo de retorno e argumentos, de modo a poder gerar erros semânticos se estas forem usadas de forma errada. Este ficheiro de configuração é alterável. É possível consultar mais informações sobre este tópico no *Capítulo 4*.

Do ponto de vista do programador as calls funcionam exatamente como funções normais, mas chamá-las requer a keyword **call**.

```
call print("hi")
call motors(left=70, right=50)
num angle = call beaconAngle()
```

3.2 Palavras Reservadas

Listagem das palavras reservadas de *CRAL*. Estas não podem ser usadas como nomes de variáveis/funções/behaviours.

when	while	for	until
local	behav	set	end
string	num	cond	bool
break	return	other	call
true	false		

Para além destas, não é permitido definir funções com nomes idênticos a **calls** definidas no ficheiro de configuração [4].

3.3 Tipos

Estão disponíveis os seguintes tipos:

num (number)

Número real inteiro ou decimal. Utilizado para representar quantidades reais. Permite valores positivos e negativos.

string

Utilizado para representar texto, equivalente a String em **C++**. O valor literal representa-se entre aspas '"'.

Permite concatenação de expressões de qualquer tipo de forma simples. Utilizando o token '**$**' como no exemplo seguinte:

```
num fd = 10
num ba = 90

num sum(num a, num b):
    return a + b
end

call print("distances: $fd$, $3+2$, $sum(5,10)$; angle: $ba$")
# deverá ser impressa a string: distances: 10, 5, 15; angle: 90
```

Se se pretender escrever o caracter literal '**$**' na string é necessário escapá-lo utilizando '**$**'.

```
num x = 1
string y = "Writing \$x\$ in this string results in: $x$"
# Output: "Writing $x$ in this string results in: 1"
```

cond

Condição cujo valor booleano (**true** ou **false**) é re-calculado cada vez que se pretende aceder a ele. Equivalente a uma função booleana mas com sintaxe mais simples.

Este tipo de variável é *final*, ou seja, uma vez configurada, não se pode modificar novamente, funcionando, assim, como uma constante.

Contextualização: Nos programas desenvolvidos pelos participantes da Academia de Verão em **C++** é frequente a utilização de estruturas **if-else if-else** de grandes dimensões em que algumas das condições são reutilizadas em outras partes do programa.

Por exemplo, verificar se a distância a um obstáculo está dentro de um certo intervalo. Se for necessário alterar o intervalo o aluno terá de o fazer em diversos locais no código e tende a confundir-se.

Dado este problema a equipa decidiu disponibilizar o tipo **cond**, que permite atribuir um label a uma condição booleana e utilizá-lo no resto do código. Assim, se for necessária uma alteração da condição, esta faz-se apenas na declaração da variável **cond**. Para além disso, a leitura do código torna-se mais fácil.

bool

Valor booleano normal (valor permanece estático). As expressões são calculadas e o valor final pode ser **true** ou **false**.

3.4 Variáveis

Podem ser declaradas variáveis de qualquer tipo suportado pela linguagem, sendo que este tem de ser explicitamente indicado e não pode ser alterado durante o tempo de vida da variável.

Por omissão todas as variáveis declaradas pertencem ao contexto global, exceto se for utilizada a keyword **local**. Desta forma alivia-se a confusão sentida pelos participantes que têm dificuldade em perceber programação por contextos.

```
# Exemplos de declarações
num answer
string nome
# Declarações com assign
cond at_start = startDistance() < 50
local bool verdadeiro = false   # Variável local
# Assign de variaveis já declaradas
answer = 42
verdadeiro = true
```

3.5 Expressões e Operadores

Listagem dos operadores disponíveis para usar em expressões:

Operador	Função	Tipos suportados
+	soma aritmética	num
-	subtração aritmética | sinal negativo	num
*	multiplicação aritmética	num
/	divisão aritmética	num
%	resto da divisão inteira	num
^	potência	num
++	incremento por 1	num
--	decremento por 1	num
+=	operador de atribuição da adição	num
-=	operador de atribuição da subtração	num
*=	operador de atribuição da multiplicação	num
/=	operador de atribuição da divisão	num
%=	operador de atribuição do resto	num
^=	operador de atribuição da potência	num
is	igualdade	num, string, bool, cond
==	igualdade (símbolo alternativo)	num, string, bool, cond
!=	desigualdade	num, string, bool, cond
>	maior	num
<	menor	num
>=	maior ou igual	num
<=	menor ou igual	num
and	conjunção lógica	bool, cond
or	disjunção lógica	bool, cond
not	negação lógica	bool, cond
in [x,y]	pertença a um intervalo	num

Notação de intervalo	Tradução
x in [-1,2]	-1 <= x <= 2
x in]-1,2]	-1 < x <= 2
x in [-1,2[	-1 <= x < 2
x in]-1,2[	-1 < x < 2

É também permitida a utilização de expressões booleanas ternárias de verificação de pertença a um intervalo:

```
-1 < x <= 2  # Equivale a: x > -1 and x <= 2

# Exemplos de expressões
num expr = 3 + 4
bool outra_expr = 30 > 4 and false
expr += 2   # expr = 9
```

```
while expr in [9, 15[ :
    expr++
end
# expr = 15
```

3.6 Controlo de Fluxo

3.6.1 Escolha - when blocks

Contextualização: Nos programas desenvolvidos pelos participantes da Academia de Verão em **C++** é frequente a utilização de estruturas **if-else if-else** de grandes dimensões e em que cada condicional executa um número reduzido de instruções. Pelo que seria conveniente simplificar a sintaxe.
Dado este problema a equipa decidiu tentar emular a sintaxe de um bloco **switch-case**.

Para iniciar um **when-block** utiliza-se a expressão **when** seguida de '**:**', e **end** para terminar. Dentro do bloco apenas é permitido escrever condições. A 'cabeça' da condição é uma expressão ou variável do tipo **bool** ou **cond**. O corpo é uma sequência de instruções que podem ser in-line (com uso de '**;**' caso seja mais do que uma) ou multi-line.

Qualquer condição tem prioridade sobre as condições escritas posteriormente e **other** é uma palavra reservada que equivale a **else**. Pelo que:

```
when:
    x > 1: call print("if")
    bool_variable: call print("else if")
    other: call print("else")
end
```

Equivale, em **C**, a:

```
if(x > 1) {
    print("if"); // printf(......)
}
else if(bool_variable){
    print("else if");
}
else{
    print("else");
}
```

```
# Exemplo de when-block
when:
    # Equivalente, em C, a: if(cond_variable) { num = 42; }
    cond_variable: num = 42
    bool_variable: call print("else if")
    x in [3,8]: motors(30,50)
    x > 50:
        call print("else if")
        while true:
            call print("infinite loop")
        end
    other: call print("else")
end
```

3.6.2 Ciclos

Em *CRAL* são disponibilizados 3 tipos de ciclos distintos:

- **while**

  Este tipo de ciclo é equivalente a um **while loop** em **C++**. O ciclo acontece enquanto a condição for verdadeira. Aceita qualquer condição booleana (**bool** ou **cond**).

  ```
  # Exemplo de ciclos while
  while true:
      while 3 > 6:
          call print("unreachable code")
      end
  end

  num soma = 0
  while bool_or_cond_variable:
      soma = soma + 2
  end
  ```

- **for**

  Este tipo de ciclo é equivalente a um **for loop** em **C++**. Pode ser utilizado de 2 formas diferentes.

  1. **for** de 3 expressões (inicialização, condição, ação posterior)

     As três expressões são separadas por '**;**' e significam, respetivamente, ações que são executadas antes do início do ciclo, condição que é verificada a cada iteração, e ações que são efetuadas no fim de cada

iteração.

```
for num i = 0; i < 4; i = i + 1:
    call print(i) # É executado 3 vezes
end
```

2. **for** num intervalo

```
for num i in [0, 3]:
    call print(i) # É executado 3 vezes
end
```

Ambos os ciclos **for** apresentados têm o mesmo resultado/output.

- **until**

  Este tipo de ciclo é equivalente a um **while loop** em **C++** com a sua condição negada. Isto é, o ciclo acontece até que a condição se torne verdadeira. Aceita qualquer condição booleana (**bool** ou **cond**).

```
# Exemplo de ciclos until
num soma = 0
until soma > 10:
    soma = soma + 1
end
call print(soma) # soma = 11
```

3.6.3 Funções

Um programa em *CRAL*, à semelhança de outras linguagens como o **C**, pode ser organizado em funções. As funções possuem obrigatoriamente um nome e um bloco de código que é executado no caso de ser chamada. Podem adicionalmente ter uma lista de argumentos e um tipo de retorno. Uma função sem tipo de retorno é equivalente a uma função **void** em **C** e linguagens semelhantes.

Para tornar o código mas legível para os participantes da Academia de Verão, a lista de argumentos requer que o nome do argumento seja identificado.

As funções declaradas podem ser chamadas com a sintaxe apresentada no exemplo que se segue.

```
sayHi():
    call print("Hi")
end

num soma(num op1, num op2):
    return op1 + op2                # Retornar num
end
```

```
string getNome():
    return "\textit{CRAL}"            # Retornar String
end

main():
    #* Chamar Funções *#
    # Utilização sem valor de retorno ou argumentos
    sayHi()
    # Utilização do valor de retorno
    call print(getNome())
    # Utilização do valor de retorno e argumentos
    num resposta = soma(op1=20, op2=21) + 1
    # A ordem em que os argumentos aparecem não é importante
    num resposta2 = soma(op2=99, op1=1) + 1
end
```

3.6.4 Behaviours

Contextualização: Nos programas desenvolvidos pelos participantes da Academia de Verão em **C++** é frequente a utilização de um elevado número de ciclos de grandes dimensões para descrever comportamentos do robot enquanto a condição do ciclo se verificar. Esta estrutura tem efeitos adversos na legibilidade do código.

Um **behaviour** é essencialmente uma função que executa o bloco em ciclo até uma condição se verificar. A cada ciclo de um behaviour é chamada a call apply que garante que o behaviour está sincronizado com simulador. Isto é, corresponde a um determinado comportamento do robot.

Desta forma, um programa pode ser abstraído como uma sequência de comportamentos.

Para iniciar um comportamento é utilizada a keyword **set**.

```
at_beacon = call groundType() is target
left_of_beacon  = call beaconAngle(target) < -5
right_of_beacon = call beaconAngle(target) > 5

behav gotoBeaconAvoidingObstacles until at_beacon:

    # going to target, adjusting orientation if necessary
    # get sensor data and beacon angle
    num fd = call sensorData(FRONT)
    num ld = call sensorData(LEFT)
    num rd = call sensorData(RIGHT)
    num ba = call beaconAngle(target)
    call print("distances: $fd$, $ld$, $rd$; angle: $ba$")
```

```
    when:
        fd < 80:
            # nested when-block
            when:
                ld < 80: call motors(left=50,right=-50)
                other: call motors(left=-50, right=50)
            end
        ld < 80: call motors(left=80, right=40)
        rd < 80: call motors(left=40, right=80)
        left_of_beacon: call motors(left=80, right=40)
        right_of_beacon: call motors(left=40, right=80)
        other: call motors(left=80, right=80)
    end
end

main():
    num target = 1
    set gotoBeaconAvoidingObstacles
    # após o primeiro behav ter terminado, iniciar outro
    set anotherBehav
end
```

Capítulo 4

Configuração do Compilador

4.1 Introdução

É permitida a configuração de certos aspetos da compilação através da escrita de um ficheiro **.config**, que é interpretado apenas uma vez no início do processo de compilação.

Contextualização: Visto que o objetivo da linguagem é a programação de robots reais e, principalmente, simulados, os programas feitos nesta necessitam de um conjunto de funções que lhe permitam interagir com o robot/simulador. Estas funções provêm essencialmente de bibliotecas externas em C++ que devem ser importadas no ficheiro output (também **C++**).

Portanto, é evidente a necessidade de haver um ficheiro de configuração onde os imports que o ficheiro output deve ter sejam especificados. No entanto, essa configuração não é suficiente, já que o compilador de *CRAL* desconheceria quais as funções disponíveis nas bibliotecas importadas e os seus argumentos e tipos de retorno. O que impossibilita a indicação de erros, caso existam.

O ficheiro inclui 3 componentes: **header**, **define**, e **calls**, por essa ordem. No entanto, apenas **calls** é obrigatória.

```
# Exemplo
header :
#include <stdio.h>
#include "CiberAV.h"
end
define:
    FRONT : int
    LEFT : int
    RIGHT : int
    REAR : int
end
```

```
#* multi-line
   comment *#
calls:
    # inline comment
    call "init":
        init
        vars:
            "name" : string [1:] "Name can not be empty"
            "pos"  : int [:] ""
            "host" : string [:] ""
        end
        methods:
            [name, pos] : "init(<name>,<pos>)"
            [name, pos, host] : "init2(<name>,<pos>,<host>)"
        end
    end
    call "north":
        critical    # another comment
        return: int end
        methods:
            [] : "northAngle()"
        end
    end
end
```

Tal como em *CRAL*, os blocos iniciam-se com o caráter '**:**' e terminam com a keyword **end**.

4.2 Header

A componente **header** permite escrever código em **C++** que é injetado no topo do ficheiro output da compilação. É especialmente útil para importar bibliotecas relevantes à programação do robot, interação com o simulador, entre outros. Esta componente, ao contrário da componente **calls**, é opcional.

```
# Exemplo
header :
#include <stdio.h>
#include "CiberAV.h"
// comentário em C++ aparece também no topo do ficheiro output
int func(int arg){
    return arg + 1;
}
end
```

De notar que quaisquer funções definidas no **header** terão de constar em **calls** para que o compilador tome conhecimento delas e funcionem corretamente.

4.3 Define

A componente **define** permite listar as variáveis e constantes que se pretendem disponibilizar aos alunos existentes nas bibliotecas importadas ou no **header**. Desta forma, o compilador de *CRAL* conhecerá quais as variáveis e constantes disponíveis nas bibliotecas importadas e os seus respetivos tipos.

```
define:
    "FRONTSENSOR" : int
    "NAME" : string
    "val"  : double
    "is_finished" : bool
end
```

4.4 Calls

Após inserir bibliotecas, é necessário definir todas as funções que o aluno poderá usar. Para tal, a componente **calls** apresenta uma lista das funções disponíveis, tal como o seu tipo de retorno e lista de argumentos, que permite posteriormente averiguar e indicar a ocorrência de erros, caso a função não tenha sido corretamente utilizada.

Assim, a componente *calls* contém uma lista de blocos **call** que representam funções, sendo que cada uma pode apresentar vários blocos como: *init*, *apply*, *critical*, *return*, *vars* e *methods*. Estes blocos têm que estar definidos por esta ordem, no entanto, só o bloco *methods* é obrigatório.

4.4.1 Init

O bloco *init* permite sinalizar a **call** que deve ser chamada em primeiro lugar na função *main*, caso exista.

Este bloco não pode estar em simultâneo com o bloco *apply*.

Contextualização: Na biblioteca da Academia de Verão (CiberAV), que foi disponibilizada como referência para a realização deste projeto, existe uma função *init* que é crucial ser chamada como primeira instrução da função *main*. Por esse motivo, é importante o compilador tomar o conhecimento qual das funções definidas do bloco **calls** é esta função prioritária.

```
# Exemplo
call "example":
    init
    # (.....)
end
```

4.4.2 Apply

O bloco *apply* tem como função identificar qual das calls é a função de atualização do simulador. (ver *Subseção 3.6.4*)

```
# Exemplo
call "example":
    # (.....)
    apply
    # (.....)
end
```

4.4.3 Critical

O bloco *critical* significa qual a função que deve ser utilizada num contexto atualizado, como um **behaviour**. Permitindo assim apresentar um *warning* ao programador se a função for utilizada em outros contextos.

```
# Exemplo
call "example":
    # (.....)
    critical
    # (.....)
end
```

4.4.4 Return

O bloco *return* especifica o tipo de retorno da call, podendo tomar os valores: **int**, **double**, **bool**, e **string**.

```
# Exemplo
call "example":
    # (.....)
    return: int end
    # (.....)
end
```

4.4.5 Vars

O bloco *vars* especifica as variáveis que podem ser passadas na chamada de uma call e as suas respetivas restrições, nomeadamente o seu tipo e, opcionalmente, valores permitidos e mensagem de erro apresentada caso as restrições não sejam respeitadas.

```
# Exemplo
call "example":
    # (.....)
```

```
    vars:
        "name" : string [1:] "Name can not be empty"
        "pos"  : int
        "left"  : int [-150:150] "power out of range"
        "sensor" : int [0:3] "sensor does not exist"
        "var" : int [0:3] "sensor does not exist"
    end
    # (.....)
end
```

4.4.6 Methods

O bloco *methods* contém as combinações das variáveis permitidas e a respetiva função a que correspondem no ficheiro importado no bloco **header**, em que os argumentos estão definidos no formato StringTemplate para facilitar o *parsing* do ficheiro de configuração.

```
# Exemplo
call "example":
    # (.....)
    methods:
        [name, pos] : "init(<name>,<pos>)"
        [name, pos, host] : "init2(<name>,<pos>,<host>)"
    end
end
```

Capítulo 5

Contribuições dos autores

- **Eurico Dias** - 20%
  - Compilador da linguagem *CRAL*
  - Ajuda no Ficheiro String Template Group para **C++**
  - Classes da package lib
  - Planeamento e conceptualização
- **Pedro Valério** - 20%
  - Gramática da linguagem *CRAL*
  - Análise Semântica da linguagem *CRAL*
  - Classes da package lib
  - Planeamento e conceptualização
- **Daniel Correia** - 20%
  - Gramática do Ficheiro de Configuração
  - Interpretador do Ficheiro de Configuração
  - Classes da package lib
  - Planeamento e conceptualização
- **Rodrigo Rosmaninho** - 20%
  - Ficheiro String Template Group para **C++**
  - Relatório / Documentação
  - Ajuda no Compilador da linguagem *CRAL*
  - Planeamento e conceptualização
- **Rita Amante** - 20%
  - Error Handling
  - Relatório / Documentação
  - Planeamento e conceptualização