Trabalho final - SQUEAK

Linguagens Formais e Autómatos

Universidade de Aveiro

Ariana Gonçalves, Dário Alves, Diogo Correia,
João Gonçalves, Mariana Pinto

Linguagem para definição e manipulação de robots

Departamento de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro

(89194) ariana@ua.pt, (67591) dario.alv@ua.pt,
(90327) diogo@ua.pt,
(80179) joao@ua.pt, (84792) mariana17@ua.pt

DATA DE ENTREGA: 7 de Julho de 2019

Resumo

No âmbito da disciplina de LFA - Linguagens Formais e Automátos - foi proposta a realização de um trabalho final para a consolidação de conhecimentos sobre a matéria lecionada ao longo do semestre sobre compiladores e linguagens.

De entre as opções dadas, foi escolhido o tema sobre "Linguagem para definição e manipulação de robots", com o objetivo de criar uma linguagem de programação de uso genérico, com a finalidade de facilitar a programação de robots usados pelos alunos que participam na Academia de Verão da UA no programa Micro-Rato.

Conteúdo

Capítulo 1

Introdução à linguagem

A dificuldade que está associada à programação de qualquer linguagem leva a que muitas vezes os futuros programadores tenham dificuldades em colocar os seus protótipos a funcionar. Quando, pela primeira vez, alguém se depara com algum tipo de linguagem de programação pode ser complicado entender o mecanismo por detrás das linhas de código por a leitura do mesmo não ser intuitiva.

Este projeto tentou combater um pouco o estigma inerente à utilização dos robots pela academia de Verão da UA no Micro-Rato, sendo este o objetivo de utilização final desta linguagem: auxiliar e simplificar o trabalho dos alunos que usem os robots no programa.

O nome da linguagem foi pensado exatamente em concordância com o seu objetivo, daí o nome "SQUEAK"alusivo ao guincho do rato, onde os seus ficheiros têm a terminação ".sqk". Foram criadas palavras de auxilio que retornam valores importantes (como por exemplo distância ao alvo, velocidade das rodas, etc), ciclos/loops, condições para serem verificados os valores do robot e mais funcionalidades que serão aprofundadas posteriormente neste relatório.

Está disponível um manual de utilizador para backup de utilização da linguagem SQUEAK.

É importante realçar que SQUEAK é uma linguagem case-insensitive, ou seja, não existe diferenciação entre letras uppercase e lowercase (maiúsculas e minúsculas).

Capítulo 2

A interface robótica

2.1 CiberRato Robot Simulation Environment

A linguagem que se irá construir tem por base o controle dos robots utilizados na academia de Verão da UA, usados com o "CiberRato Robot Simulation Environment"que simula o movimento do robot dentro de um labirinto. O objectivo dos robots é partir da sua posição inicial até à área do queijo, apanhá-lo e voltar para essa mesma posição inicial. Usou-se a versão v2.2.5 do programa e o ambiente usado é o Explorer 2019.

Para o CiberRato Robot Simulation Environment ser usado é necessário a biblioteca "qt4-dev-tools".

Capítulo 3

Linguagem Squeak

3.1 Tokens

Tokens são símbolos terminais (segmentos de texto) que podem ser manipulados por um analisador sintático, de forma a fornecer um significado coletivo a esse conjunto de carateres. Neste caso, são utilizáveis pela linguagem para definir funções importantes do robot, enviando e retornando do e para o robot informações da utilização e funcionamento do mesmo.

São exemplos de tokens:

- **START** - dá início ao programa e da main, indica o nome do robot e a sua posição inicial: *START "ROBOT_NAME"position 0*;
- **WRITE** - semelhante ao print de java/C, é a função de saída de texto: *WRITE(output)*;
- **END** - token que identifica o final da main: *END*;
- **PICKUP** - recolhe um objeto(por exemplo, um queijo) *PICKUP*;
- **RETURN** - retorna ao ponto inicial/partida: *RETURN*;
- **GROUND** - retorna o valor inteiro do tipo de chão no qual o robot se encontra, 0 se for chão e 1 até n se for um dos n queijos: *GROUND*;
- **WHEELS** - define os valores do poder das rodas da esquerda e da direita: *WHEELS(x y)*;

Para mais exemplos de tokens, consultar o manual de instruções do utilizador para maior detalhe.

3.2 Print vs. Write

O comando WRITE é usado para retornar o output para a consola daquilo que está a decorrer no programa. Poderá haver conflitos com concatenações

de strings dentro do comando WRITE, por isso é aconselhável que se declare a variavel de concatenaçao de strings fora e depois colocar essa mesma variável dentro do WRITE.

WRITE(output)

O output pode ser variado, desde strings, variáveis a tokens, desde que estes últimos retornem algum tipo de valor que possa ser escrito pelo token no terminal.

É um dos comandos mais importantes das linguagens de programação pois é o que permite que o programador exiba dados na consola, acompanhando a evolução do programa quando a aplicação é executada.

3.3 Main vs. Start/End

Foi optado desconstruir a main de forma a simplificar a linguagem construída. Assim, tudo o que estiver fora dos tokens FUNCTION e dentro dos pares de tokens START/END é considerado parte da função main quando comparada à linguagem java/C.

É importante realçar que não é possível ter código fora das funções (chamadas FUNCTION) ou da main, daí serem importantes o START e o END para o analisador léxico reconhecer os tokens como pertencendo a elementos distintos da linguagem e o sintático quanto ao seu posicionamento num programa.

O token START tem de ser sempre seguido por uma string, de forma a reconhecer o nome do robot que se quer usar, e é possível começar o robot apartir de uma certa posição no mapa. Este feature é possível através do token POSITION logo após à string com o valor preferível para o robot iniciar o seu movimento.

Um exemplo de como utilizar este excerto de código pode ser:

```
START"robot"POSITION 6
    WRITE("Going to target")
    PICKUP
END
```

No caso de não ser usado o token POSITION após a string, a linguagem supõe que POSITION tem o valor 0.

3.4 Funções vs. Function

Tudo o que não esteja entre START e END tem de estar dentro de funções. Estes blocos são construídos fora do bloco START/END mas são chamados no bloco START/END (main) através do nome que o utilizador lhe dá.

A sua estrutura caraterística é:

FUNCTION name(input)=>output

Sabe-se que uma função chamada "xpto"retorna algum valor quando tem a forma:

FUNCTION xpto(x)=>y

O retorno desta será y e a assinatura (parâmetros de entrada) será x.

Quando for uma função do tipo:

FUNCTION xpto(x)

Não retorna valor algum (tipo void quando comparado a JAVA) e se for apenas:

FUNCTION xpto()

Não tem parâmetros de entrada ou retorna valores.

Duas funçoes podem ter o mesmo nome desde que tenham assinaturas diferentes, ou seja, desde que tenham parâmetros de entrada diferentes. Se tiverem o mesmo nome e assinatura a linguagem não aceita a função.

3.5 Atribuições/Declarações

O uso de variáveis é uma das bases fundamentais da programação por serem capazes de armazenar dados ou informações de um programa por um determinado espaço de tempo.

A declaração destas entidades é importante nesta linguagem, caso contrário, seria impossível sinalizar certos erros semânticos, essenciais ao programador.

A sua inicialização é dada através de:

name <- value

Para que as variáveis não tivessem de ser inicializadas teria-se que ter um valor default para cada uma destas, o que poderia gerar vários conflitos para o compilador. Considerando uma variável xpto que é criada e depois é igualada a um token que retornará um valor que inicializará o mesmo.

xpto <- GROUND

Neste caso, "xpto"terá o valor que o token ground retorna, pois está-se a igualar "xpto"a um inteiro (mesmo que indiretamente), ou seja, estamos a inicializar a variável: imaginando que GROUND retorna o inteiro 1, "xpto"terá igualmente o valor 1.

Mas seguindo o mesmo caso de raciocínio, se fosse necessário incrementar o valor de xpto e por alguma razão trocássemos os carateres do mesmo, sendo que teria-se um valor default para variáveis nao inicializadas (como por exemplo 0):

xpto <- GROUND
WRITE(xpto)
xpto <- xpot + 1;

Assim, iria ser criada uma nova variável (xpot) em vez de retornar um erro a avisar o programador que a variável "xpot"era mal usada e não inicializada. Nesse caso "xpot"era igual a 0 e "xpto"continuaria a ter o valor 1 em vez de ser incrementado.

Concluindo, todas as variáveis têm de ser inicializadas antes de serem utilizadas.

As palavras reservadas presentes neste documento, como START, END, FUNCTION, LOOP, actions, etc, não poderão ser usadas como variáveis nestes

casos. Os vários tokens que retornam valores poderão ser usados como valores para igualarem às mesmas.

3.6 When vs. Loop

Os blocos de código LOOP vão servir para adotar estruturas de repetições de instruções (ou conjuntos de instruções) enquanto uma dada condição for satisfeita. Será equivalente ao WHEN da linguagem JAVA, utilizada anteriormente no Micro-Rato.

Assim, o nome LOOP tem a seguinte estrutura:

LOOP
    [CONDIÇÃO] => TOKEN
ENDLOOP

As condições são formadas por expressões matemáticas, usando os tokens que irão ser comparados aos valores condicionais em questão. No exemplo a seguir, é mostrado um excerto de código de uma condição:

[GROUND=1] => FOWARD 5

Neste caso, enquanto o chão for do tipo 1 anda 5 robots para a frente. Quando não se verificar, o ciclo acaba.

Não podem ser declaradas variáveis dentro de blocos LOOP.

3.7 If vs. Actions

Uma das maiores dificuldades apontadas pelos professores na Academia de Verão era a importância e a dificuldade que os alunos tinham de entender o conceito dos if's e do programa reproduzido pelos mesmos.

Mais uma vez, usou-se condições matemáticas para resolver/simplificar esta barreira que tornava confuso, à primeira vista, alguém de fora entender o que estava a ser programado.

Todas as condições que estiverem dentro de actions serão lidas linha a linha até uma delas se verificar. Quando isso acontecer, todas as seguintes são descartadas e a expressão à direita do sinal "=>"é posta em ação, daí o termo "actions".

A sua estrutura consiste em:

ACTIONS
    [CONDIÇÃO1] => TOKEN
    [CONDIÇÃO2] => TOKEN
    TOKEN
ENDACTIONS

Neste exemplo, se a primeira condição for verdadeira, o código descarta todas as linhas até encontrar o token ENDACTIONS. Por outro lado, se a primeira condição for falsa, passa para a segunda e se esta for falsa para a terceira e assim sucessivamente até encontrar uma que seja verdadeira. Se nenhuma das condições for verdadeira é então realizado o token que se encontra isolado no fim

de todas as condiçoes, antes da ENDactions. // perguntar se um token estiver no meio das condições é realizado.

Agora vejamos outro caso: se quisermos fazer uma ação dentro de uma ação (if dentro de if).

ACTIONS
[CONDIÇÃO1] => ***TOKEN***
[CONDIÇÃO2] =>
ACTIONS
[CONDIÇÃO21] => ***TOKEN***
[CONDIÇÃO22] => ***TOKEN***
ENDACTIONS
[CONDIÇÃO3] => ***TOKEN***
TOKEN
ENDACTIONS

Se se chegar à condição1 e esta for verdadeira, são descartadas as seguintes. Se for falsa, passa para a condição 2 e se esta for falsa descarta a ação que segue o lado direito da seta até encontrar ENDACTIONS. Se for verdadeira a ACTIONS que está à direita da seta é realizada e o procedimento é o mesmo, ou seja, irá verificar as condições 21 e 22 até encontrar uma verdadeira. Quando encontrar o token ENDACTIONS dessa ação todas as linhas de código serão descartadas até haver uma ENDACTIONS pois a condição2 como é verdadeira faz com que a condição3 nao tenha de ser analisada.

Não podem ser inicializadas variáveis dentro do ACTIONS.

3.8 Comentários

O uso de comentários é importante em todas as linguagens de programação para o programador adicionar notas/ideias que acha importante mas não quer que sejam analisadas pelo compilador. Daí estarem disponíveis 2 tipos de comentarios:

- **Comentários em bloco** - tal como em JAVA são implementados com:

***/ multi**

*** line**

*** comment**

***/**

- **Comentários por linha** - igual a JAVA, são inicializados com:

// **single line comment**

Capítulo 4

Implementação / Arquitetura

4.1 Criação Linguagem

A criação da nossa linguagem teve como base o ficheiro exemplo de manipulação de robots fornecido no pacote do ambiente de simulação. A análise do mesmo levou a que uma série de regras fossem implementadas para melhor gestão do código produzido, de forma a que não fossem produzidas situações gramaticais desnecessárias com o intuito de gestão de tempo e recursos.

4.2 Gramáticas

As gramáticas são responsáveis pelo reconhecimento dos tokens, bem como a análise lexical e sintática da nossa linguagem. O funcionamento e interpretação da linguagem foi feito conforme descrito anteriormente.

O ficheiro "MouseBotLexer.g4"é responsável pela criação dos tokens que interagem com o robot, em que são criadas todos as palavras que enviam/recebem valores do mesmo.

o "OperationsLexer.g4"é responsável pelas várias operações matemáticas/lógicas possíveis que podem ocorrer na linguagem, bem como o operador de atribuição nomeado anteriormente.

As operações para comparação possíveis são:

1. Diferente: *!=*
2. Igual: =
3. Maior/igual: >=
4. Menor/igual: <=
5. Maior: >
6. Menor: <

As operações matemáticas são:

1. Soma: *+*
2. Substração: -
3. Multiplicação: *
4. Divisão: */*
5. Resto da divisão: *%*
6. Expoente: ^

E as operações lógicas são:

1. AND: *AND*
2. OR: *OR*
3. NOT: *NOT*

"DecisionLexer.g4"das estruturas de repetição que o utilizador pode utilizar no programa (LOOP e ACTIONS);

"CaseInsensitiveLexer.g4"é responsável pela linguagem ser case-insensitive, conferindo a igualdade de letras maiúsculas a minúsculas;

"OtherLexer.g4"inclui as regras gerais da linguagem, como carateres primitidos, skip de espaços e indentações e novas linhas;

"PrimitivesLexer.g4"é responsável por declarar todos os tipos primitivos que são utilizados. Pela análise deste último ficheiro, consegue-se entender que não existem distinções entre valores inteiros e decimais, sendo todos lidos como "NUMBER"pelo programa. Podem ser utilizados todos os números formados pelos algarismos de 0 a 9 e as variáveis podem ser formadas por todas as letras do abcedário e números, desde que o primeiro carater seja uma letra.

"MouseBotParser.g4"cria as regras da biblioteca de forma a que sejam respeitadas todas as regras padrão definidas anteriormente. Decidimos que todos os programas construídos tenham de começar com START/END (como se tratasse de uma main) e dar a opção de terem (ou não) funções. A main terá de ser sempre criada no início do programa seguida de todas as funções, aos o token END referente ao START.

Não podem ser inicializadas variáveis dentro de loops e actions e loops e actions não podem estar sem, pelo menos, uma condição dentro dos mesmos.

4.3 Tabela de Símbolos

A pasta "symbolTable"contem os programas necessários para a tabela de símbolos ser construída. O "Symbol.java"guarda os diferentes tipos de simbolos que o analisador semantico recolhe, pois todos têm algo em como: o nome, que é o que a classe abstrata armazena.

O "FuncSymbol.java"é uma extensão do "Symbol.java"em que guarda todos os simbolos/nomes de uma função. Tem uma lista LINKEDLIST para estes ficarem organizados consoante o seu aparecimento, em que a lista retorna os parâmetros da funçao à medida que estes aparecem no programa. É importante pois é aqui que se consegue verificar, através da função "equals(Object obj)", se duas funções com o mesmo nome têm a mesma assinatura, dando erro se tal acontecer.

O "VariableSymbol.java"é também uma extensão do "Symbol.java"mas que guarda todos os símbolos/nomes de variáveis e o seu tipo, através do enum "VarType.java".

4.4 Compilador

"Synthesis.java"é o compilador da linguagem, que atráves do "MouseBotMain.java"é a main usada para compilar os programas SQUEAK (.sqk), utilizando a ferramenta String Template, do mesmo criador do ANTLR4, Terrence Parr. Se o ficheiro compilar sem erros, é imprimido no terminal o código final traduzido para C++, de forma a que a nossa linguagem possa ser lida pelo ambiente de simulação do robot.

Uma das razões para a qual o String Template tornou-se bastante importante foi para ultrapassar o problema descrito pelo professor Artur aquando do apply usado pelos alunos na academia de Verão. Sempre que wheels é usado, o ficheiro "c.stg"chama o apply sem ter de ser chamado pelo utilizador.

4.5 Análise Semântica

O analisador semântico é responsável por verificar os erros semânticos e em caso de detecção dos mesmos, faz a impressão do tipo de erro e a sua localização, e a análise do nome das variáveis tendo de analisar se estas são nomes de funçoes ou de variaveis para poderem ser adicionadas à tabela de símbolos. O ficheiro responsável pela análise semantica é o "AnalisadorSemantico.java"

A pasta "semanticUtils"também é muito importante para este capítulo, em que o ficheiro "Error.java"associa um símbolo (função ou variável) a um tipo de erro. Os tipos de erros possíveis podem ser encontrados no ficheiro enumerate "ErrorType.java".

O "ErrorHandler.java"tem duas estruturas de dados essenciais para a geração dos erros no terminal: um Hashmap e um TreeMap.

O primeiro retorna uma lista de funçoes à espera de serem definidas quando a main (START/END) é lida. Assim, é possível reunir as funções que precisam de ser definidas depois da main. Por cada linha da main que tenha uma função não definida, a função awaitFunctionDefinition adiciona ao HashMap essa função.

Caso uma função seja definida, percorremos o HashMap criado e se alguma das entradas deste corresponder com a nova função, é apagada deste, usando a

função define(FuncSymbol funcao) para a função endFile() não retornar a lista das funções que foram utilizadas na main e precisavam de ser definidas.

A segunda associa a posição a um erro, de forma a que quando seja necessário imprimir os erros seja mais fácil. O erro é adicionado através das funções addError e existem duas para tal pois as duas formas são válidas no analisador semântico.

O "Position.java"alinha o erro obtido com a sua posiçao no código. Este tinha de ser particular e separado dos outros pois utilizou-se um comparable para ter a certeza que os primeiros erros apareciam por ordem de aparecimento do código no terminal.

4.6 Tratamento de Erros

O tratamento de erros está ao cargo dos ficheiros da pasta "sintaticUtils"através dos ficheiros "MouseBotErrorStrategy.java", o "MouseBotErrorListener.java"e "UnwantedToken.java". O primeiro apenas muda os prints do ErrorStrategy original do ANTLR de forma a tornar os erros mais ligivéis pelo utilizador. O segundo ficheiro tem todos os outros erros possiveis que possam surgir na linguagem, analisa o erro e substitui pelo equivalente em português.

O "UnwantedToken.java"retorna os erros referentes a tokens e possíveis complicações que possam surgir próprio dos tokens.

4.7 Ficheiros Teste

Nos ficheiros testes estão presentes erros que pudessem surgir na linguagem e que confirmam o funcionamento do compilador. Estão presentes na pasta "testProg"em que o "TestActions.sqk"testa as Actions, "TestCaseInse.sqk"testa o facto da linguagem ser case insensitive, "TestFuncoes.sqk"testa funções e como estas podem ser definidas, "TestLoop.sqk"os Loops, "TestMain.sqk"os erros que podem surgir com START/END, "TestPrimitives.sqk"que testa os tipos de valores primitivos, desde a atribuições a operações e atribuições, "TestTokens.sqk"que testam os tokens presentes na linguagem que interagem com o robot e "A0squeak.sqk"contém um ficheiro que prova que a linguagem é programável no contexto de robots, pois consegue enviar o mesmo para o queijo e voltar.

Capítulo 5

Conclusão

Conclui-se que este trabalho foi um dos mais desafiadores até agora e que conseguimos cumprir com os objetivos pedidos. Criou-se uma linguagem de raiz, usando 3 tipos de linguagens diferentes para a construir (Java, Antlr e C++)

Uma das maiores dificuldades foi chegar ao protótipo inicial de SQUEAK com a possibilidade de sermos criativos na criação da linguagem, algo que demorou a ser realizado, sensivelmente 3/4 semanas. Outra dificuldade prendeu-se com o facto de não se ter conseguido acabar o trabalho a tempo da primeira entrega. Não querendo ser prejudicados pelo trabalho inicial, optou-se por entregar em recurso de forma a que o trabalho ficasse funcional e bem estruturado. O facto de elementos do grupo terem desistido nas últimas duas semanas de entrega e não terem contribuído como se esperava, ajudou a que o trabalho se atrasasse.

Assim, este projeto permitiu adquirir conhecimentos acerca da tecnologia inerente à compilação e análise léxica, sintática e semântica das linguagens de programação, aliado aos objetivos da unidade curricular.

5.1 Contribuição dos autores

Definiu-se uma contribuição dos autores de :
Ariana Gonçalves - 25%
Dário Alves - 30%
Diogo Correia - 5% (desistiu)
João Gonçalves - 15%
Mariana Pinto - 25%

Capítulo 6

Bibliografia

- Elearning.ua.pt - página da disciplina
- https://github.com/antlr/stringtemplate4/blob/master/doc/index.md (acedido em 05/07/2019)
- Documentação CiberRato tools v2.2.5
- http://microrato.ua.pt/ (acedido em 12/06/2019)