

Robot-L: Compilador para Simulação de Robô Móvel

João Pedro A. Virgili¹, Fabiano Fraga¹ Alexandre Cury¹

¹Instituto de Matemática – Universidade Federal da Bahia (UFBA)
– Salvador – BA – Brasil

{sr.joaovirgili, fragafernanes1}@gmail.com,
alexandrecurylima@hotmail.com

Abstract. *Robot-L is a high-level language focused on mobile robot control. This language has been specially designed for a very specific scenario where an autonomous mobile robot in an environment where it needs to dodge obstacles and control lighting by turning it on and off. We will discuss in detail the main steps of building a compiler: lexical analysis, syntatic analysis and semantic analysis.*

Resumo. *Robot-L é uma linguagem de alto nível focada no controle de robô móvel. Esta linguagem foi especialmente projetada para um cenário muito específico, em que um robô móvel atua em um ambiente no qual precisa desviar de obstáculos e controlar a iluminação acendendo ou apagando lâmpadas. Discutiremos detalhadamente pelas principais etapas de construção de um compilador: análise léxica, análise sintática e análise semântica.*

1. Introdução

Um compilador é um programa capaz de analisar e traduzir um código fonte de uma linguagem para o código objeto. A compilação é feita após diversas etapas de análise do código de entrada, verificando se o mesmo é válido para a linguagem do compilador. Neste artigo, a linguagem em questão é a Robot-L e temos como objetivo detalhar o processo de compilação, construção do compilador e as decisões tomadas em cada etapa. Para a construção do compilador, utilizamos a linguagem Python. A escolha foi feita por unanimidade na equipe, visto que temos mais domínio dessa linguagem e, por isso, julgamos ser a mais adequada.

Token	Expressão Regular
Programa	::= "programainicio" Declaração* "execucaoinicio" Comando "fimexecucao" "fimprograma"
Declaração	::= "definainstrucao" identificador "como" Comando
Bloco	::= "inicio" Comando* "fim"
Comando	::= Bloco Iteracao Laco Condicional Instrução
Iteracao	::= "repita" Numero "vezes" Comando "fimrepita"
Laco	::= "enquanto" Condicao "faca" Comando "fimpara"
Condicional	::= "se" Condicao "entao" Comando "fimse" ["senao" Comando "fimsenao"]
Instrucao	::= "mova" Numero* ["passos"] "Vire Para" Sentido Identificador "Pare" "Finalize" "Apague Lampada" "Acenda Lampada" "Aguarde Ate" Condição
Condicao	::= "Robo Pronto" "Robo Ocupado" "Robo Parado" "Robo Movimentando" "Frente Robo Bloqueada" "Direita Robo Bloqueada" "Esquerda Robo Bloqueada" "Lampada Acessa a Frente" "Lampada Apagada a Frente" "Lampada Acessa A Esquerda" "Lampada Apagada A Esquerda" "Lampada Acessa A Direita" "Lampada Apagada A Direita"
Identificador	::= Letra(Letra Digito)*
Numero	::= Digito*
Letra	::= "A" "a" "B" "b" ... "z"
Digito	::= "0" ... "9"
Sentido	::= "esquerda" "direita"

Figura 1. Definição da linguagem Robot-L

A estrutura do compilador é definida por duas partes principais: análise e síntese. A parte de análise, também chamada de *front*, é a parte que será abordada no decorrer deste artigo. Esta parte executa o tratamento de erro e é composta por três etapas: léxico, sintático e semântico, onde cada etapa é responsável por um tipo de erro diferente.



Figura 2. Estrutura Geral de um Compilador

A parte de síntese, também chamada de *backend*, é onde é feito a tradução do código. Após todo o código ter sido validado na parte de análise, a síntese compila o código de entrada para o código de mais baixo nível, realiza a otimização do código e então gera o código objeto a ser executado.

2. Analisador léxico

A análise léxica é a primeira etapa do compilador, onde é feito a leitura do código de entrada, caractere a caractere, o agrupamento dos caracteres em lexemas reconhecidos por um padrão e produção de uma sequência de símbolos que chamaremos de tokens. Os tokens são reunidos em uma estrutura chamada de tabela de símbolos, que poderá ser acessada e alterada durante todas as etapas do compilador. Neste caso, a tabela de símbolos inicia vazia e os tokens são inseridos à medida em que são reconhecidos. A estrutura da tabela é formada pelo token, o tipo do token e a linha em que ele foi encontrado.

De acordo com a definição da linguagem na Figura 1, reunimos todas as palavras reservadas que devem ser reconhecidas pelo analisador. Definimos que toda palavra reservada terá o tipo do token como seu próprio token, representado na Figura 3. Por exemplo, “<vire, vire>”, “<abc, identificador>” e “<123, numero>”. A função principal “lexico()” obtém a entrada passada e realiza um loop de caractere em caractere. Criamos funções auxiliares para tratar cada tipo de caractere lido, por exemplo, se for número a função “trataNumero()” é chamada. Caso o caractere não seja reconhecido é retornado erro.

```
PALAVRAS_RESERVADAS = [  
    ("programainicio", "programainicio"),  
    ("execucaoinicio", "execucaoinicio"),  
    ("fimexecucao", "fimexecucao"),  
    ("fimprograma", "fimprograma"),  
    ("definainstrucao", "definainstrucao"),  
    ("como", "como"),  
    ("inicio", "inicio"),  
    ("fim", "fim"),  
    ("repita", "repita"),  
    ("vezes", "vezes"),  
    ("fimrepita", "fimrepita"),  
    ("enquanto", "enquanto"),  
    ("faca", "faca"),  
    ("fimpara", "fimpara"),  
    ("se", "se"),  
    ("entao", "entao"),  
    ("fimse", "fimse"),  
    ("senao", "senao"),  
    ("fimsenao", "fimsenao"),  
    ("mova", "mova"),  
    ("vire", "vire"),  
    ("pare", "pare"),  
    ("finalize", "finalize"),  
    ("apague", "apague"),  
    ("passo", "passo"),  
    ("passos", "passos"),  
    ("acenda", "acenda"),  
    ("aguarde", "aguarde"),  
    ("ate", "ate"),  
    ("a", "a"),  
    ("pronto", "pronto"),  
    ("ocupado", "ocupado"),  
    ("parado", "parado"),  
    ("movimentando", "movimentando"),  
    ("bloqueada", "bloqueada"),  
    ("apagada", "apagada"),  
    ("acessa", "acessa"),  
    ("esquerda", "esquerda"),  
    ("direita", "direita"),  
    ("frente", "frente"),  
    ("robo", "robo"),  
    ("lampada", "lampada"),  
    ("pronto", "pronto"),  
    ("acesa", "acesa"),  
    ("para", "para"),  
    ("para", "para"),  
]
```

Figura 3. Lista de palavras reservadas

Ao ler um número, a função “trataNumero()” é chamada para identificar o padrão de constante numérica. O próximo caractere lido passa por tratamento de verificação de tipo:

- Se for número, o caractere é concatenado ao token atual e a função passa a tratar o próximo.
- Se for um caractere separador (espaço, tab ou quebra de linha) este token é finalizado.
- Se for letra, erro é retornado, pois o padrão de constante numérica não aceita caractere do tipo letra.
- Se for comentário (“#”), erro é retornado, pois o comentário só pode ser utilizado no começo da linha.
- Se for caractere desconhecido, retorna erro pois o mesmo não pertence à linguagem.

```
ERRO(4:14): Numérico com caractere Letra
Mova 2a Passos
```

Figura 4. Erro de constante numérica

Ao ler uma letra, a função “trataIdentificador()” é chamada. Neste caso, a função trata tanto identificador como palavras reservadas pois são os únicos tokens que começam com letra. O próximo caractere lido passa por tratamento de verificação de tipo:

- Se for letra ou número, o caractere é concatenado ao token atual e a função passa a tratar o próximo.
- Se for um caractere separador este token é finalizado. Após isso, verifica se este token formado é uma palavra reservada, se não é identificador.
- Se for comentário (“#”), erro é retornado, pois o comentário só pode ser utilizado no começo da linha.
- Se for caractere desconhecido, retorna erro pois o mesmo não pertence à linguagem.

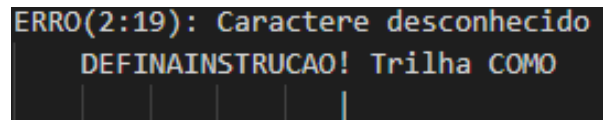
Ao ler um caractere de comentário (“#”), erro é gerado caso não seja início de linha, respeitando a regra da linguagem. Caso contrário, a função “trataComentario()” é chamada, onde todos os caracteres são ignorados até ser lido uma quebra de linha.

```
ERRO(2:20): Comentário ('#') somente em início de linha
DEFINAINSTRUCAO # Trilha COMO
```

Figura 5. Erro de comentário no meio da linha

Ao ler um caractere separador, nada é feito.

Caso o caractere lido não seja aceito em nenhum dos casos anteriores, isso quer dizer que o mesmo é desconhecido pela linguagem, retornando este erro.

A screenshot of a terminal window with a black background and white text. The text displays a compiler error: 'ERRO(2:19): Caractere desconhecido' on the first line, and 'DEFINAINSTRUCAO! Trilha COMO' on the second line. Below the text, there is a row of five empty square boxes, with a vertical cursor line positioned under the second box from the left.

```
ERRO(2:19): Caractere desconhecido
DEFINAINSTRUCAO! Trilha COMO
```

Figura 6. Erro de caractere desconhecido

3. Análise Sintática

A análise sintática é a segunda etapa do compilador, onde sua tarefa é determinar se o programa de entrada possui as sentenças válidas para a linguagem em questão. Escolhemos o analisador SLR(1) entre as opções pois sentimos mais facilidade com analisador ascendente e estudamos mais este do que o LALR(1). Nesta etapa, construímos uma gramática para especificar a sintaxe da linguagem.

Após a construção da gramática, para a construção da tabela LR, primeiro criamos a tabela First/Follow e em seguida o autômato de itens. Para auxílio de correção, utilizamos o site [jsmachine] que nos permitiu testar diversas entradas para a gramática. A tabela criada se encontra na pasta do projeto no caminho “sintatico/rl_tableFINAL.json”.

```

PROG -> programainicio DECL execucaoinicio COMANDO fimexecucao fimprograma

DECL -> definainstrucao ID como COMANDO DECL
DECL -> ''

COMANDO -> BLOCO
COMANDO -> ITER
COMANDO -> LACO
COMANDO -> IFELSE
COMANDO -> INSTR

COMANDOS -> COMANDO COMANDOS
COMANDOS -> ''

BLOCO -> inicio COMANDOS fim

ITER -> repita NUM vezes COMANDO fimrepita

LACO -> enquanto COND faca COMANDO fimpara

IFELSE -> IF
IFELSE -> IF ELSE
IF -> se COND entao COMANDO fimse
ELSE -> senao COMANDO fimsenao

INSTR -> mova NUM
INSTR -> mova NUM passos
INSTR -> vire para SENTIDO
INSTR -> ID
INSTR -> pare
INSTR -> finalize
INSTR -> apague lampada
INSTR -> acenda lampada
INSTR -> aguarde ate COND

COND -> robo pronto
COND -> robo ocupado
COND -> robo parado
COND -> robo movimentando
COND -> frente robo bloqueada
COND -> direita robo bloqueada
COND -> esquerda robo bloqueada
COND -> lampada acesa a frente
COND -> lampada apagada a frente
COND -> lampada acesa a esquerda
COND -> lampada apagada a esquerda
COND -> lampada acesa a direita
COND -> lampada apagada a direita

ID -> identificador

NUM -> numero
SENTIDO -> esquerda
SENTIDO -> direita

```

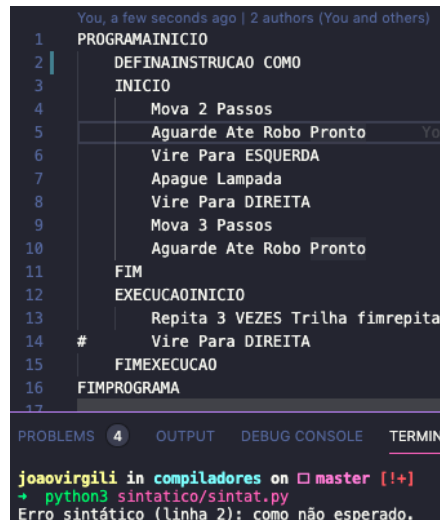
Figura 7. Gramática construída

Como vemos na Figura 1, o token “declaracao” é constituído por “**definainstrucao** identificador **como** Comando”. Caso um token “definainstrucao” não venha seguido de um “identificador”, este é um exemplo de erro sintático. Na gramática criada, temos o não terminal “DECL” que se refere a este caso.

Na implementação do analisador sintático utilizamos uma pilha referente aos tokens e o estado atual e uma cadeia de entrada. Para a implementação, temos as principais ações: *shift* e *reduce*. Começamos com a pilha apenas com o estado atual, neste caso 0. Em seguida, cada token da tabela de símbolos é lido para executar uma ação referente a ele. Após a leitura, verificamos na tabela LR que ação deve ser executada para esta entrada neste estado. O *shift* consome o token da entrada, o empilha e empilha também o estado destino. O *reduce* consome da pilha a sequência de token e estado atual e o empilhando o não-terminal referente à redução da sequência de tokens lidos. Quando, para um estado e uma entrada, não temos uma ação definida, ocorre o

erro sintático. A análise sintática encerra quando a pilha estiver vazia e a ação a ser tomada é *accept*.

Na Figura 8, vemos um exemplo do analisador sintático realizando o tratamento de erro de uma sentença de tokens que mencionamos acima. A produção “DECL” definida na gramática espera um sequencia de tokens “definainstrucao ID como DECL”, porém um token “como” não é esperado após “definainstrucao” e então o erro é exibido.



```
You, a few seconds ago | 2 authors (You and others)
1 PROGRAMAINICIO
2 |   DEFINAINSTRUCAO COMO
3   INICIO
4   Mova 2 Passos
5   Aguarde Ate Robo Pronto
6   Vire Para ESQUERDA
7   Apague Lampada
8   Vire Para DIREITA
9   Mova 3 Passos
10  Aguarde Ate Robo Pronto
11  FIM
12  EXECUCAOINICIO
13  Repita 3 VEZES Trilha fimrepita
14  #   Vire Para DIREITA
15  FIMEXECUCAO
16  FIMPROGRAMA
17

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL
joaovirgili in compiladores on master [!+]
+ python3 sintatico/sintat.py
Erro sintático (linha 2): como não esperado.
```

Figura 8. Erro sintático

4. Análise Semântica

A análise semântica é a última etapa da fase de análise do compilador, onde, como, no caso do Robot-L, verifica se duas declarações possuem o mesmo nome. O objetivo é gerar uma árvore abstrata que é uma representação mais simples e adaptada da árvore sintática para a geração do código que será a próxima fase da compilação.

A implementação foi construída usando o método de tradução dirigida pela sintaxe, mas com algumas diferenças. Aproveitamos o fluxo dos shifts do analisador sintático, que fazem a produção dos terminais, para a construção da árvore semântica. Porém, na tradução dirigida pela sintaxe, as verificações feitas pelo analisador semântico é ao mesmo tempo que vai sendo construída a árvore. O que fizemos foi construir árvore semântica, transforma-lá em string lendo de forma top-down, e então verificamos se ocorre os erros semânticos de "vire para direita vire para esquerda", ou vice versa, e se há duas declarações com o mesmo nome, usando expressões regulares para verificar suas ocorrências. Optamos por esse jeito, por ter uma pequena quantidade de erros semânticos para verificar e não precisar modificar a gramática, e o modo como ela é lida no código.

Nas figuras 9 e 10 temos os exemplos de erros semânticos.

```
1 PROGRAMAINICIO
2   DEFINAINSTRUCAO abab COMO
3   INICIO
4     Mova 2 Passos
5     Aguarde Ate Robo Pronto
6     Vire Para ESQUERDA
7     Vire Para DIREITA
8     Apague Lampada
9     Mova 3 Passos
10    Aguarde Ate Robo Pronto
11  FIM
12  EXECUCAOINICIO
13    Repita 3 VEZES Trilha fimrepita
14  # Vire Para DIREITA
15  FIMEXECUCAO
16  FIMPROGRAMA
```

PROBLEMS 4 TERMINAL ... 2: Python Debug Cons

```
joaovirgili in compiladores on master [!+]
+ python3 sintatico/sintat.py
erro semantico: ha ocorrencia de vire para em sentidos opostos seguidas
```

Figura 9. Erro “Vire para esquerda” seguido de “Vire para a direita”

```
1 PROGRAMAINICIO
2   DEFINAINSTRUCAO abab COMO
3   INICIO
4     Mova 2 Passos
5     Aguarde Ate Robo Pronto
6     Vire Para ESQUERDA
7     Vire Para DIREITA
8     Apague Lampada
9     Mova 3 Passos
10    Aguarde Ate Robo Pronto
11  FIM
12  DEFINAINSTRUCAO abab COMO You, a few seconds ago
13  INICIO
14    Mova 3 Passos
15  FIM
16  EXECUCAOINICIO
17    Repita 3 VEZES Trilha fimrepita
18  # Vire Para DIREITA
19  FIMEXECUCAO
20  FIMPROGRAMA
```

PROBLEMS 4 TERMINAL ... 2: Python Debug Cons

```
joaovirgili in compiladores on master [!+]
+ python3 sintatico/sintat.py
erro semantico: ha duas declaracoes de instrucoes com mesmo nome {'defina
instrucao abab'}
```

Figura 10. Erro de duas declarações com o mesmo nome

Referências

Jsmachine. <http://jsmachines.sourceforge.net/machines/slr.html>

Livro do "Dragao": Aho, Lam, Sethi, and Ullman. Compiladores: princípios, técnicas e ferramentas. 2a edição, Addison-Wesley, 2008.