Robot-L: Compilador para Simulação de Robô Móvel

Matheus Sandes Moreira[1], Pierre Lucas Magalhães Melhor[1]

[1]Departamento de Ciência da Computação – Universidade Federal da Bahia (UFBA)
Salvador – BA – Brasil

matheus_sandes@hotmail.com, pierremmelhor@gmail.com

Abstract. *This paper describes an implementation of the compiler made on C++ language for a high-level programming language to control a robot, made at "Compiladores" in order to control a mobile robot.*

Resumo. *Este artigo descreve uma implementação de um compilador feito na linguagem C++ para uma linguagem de alto nível, feito na matéria "Compiladores" com o intuito de controlar um robô móvel.*

1. Compilador

Um compilador é um programa de computador (ou um grupo de programas) que, a partir de um código fonte escrito em uma linguagem compilada, cria um programa equivalente em outra linguagem, código objeto. Definição essa feita pelo livro "Compiladores: Princípios, Técnicas e Ferramentas", utilizado como base para o desenvolvimento do compilador. Segundo o livro, o processo de compilação é dividido em análise e síntese.

2. Análise

O processo de análise é dividido em 3 etapas, são elas: análise léxica, análise sintática e análise semântica, seguindo exatamente essa ordem. Após essas 3 etapas, é gerado uma representação intermediária, que vai ser usada pela síntese.

2.1 Análise Léxica

A análise léxica consiste no processo de analisar a entrada de linhas de caracteres e produzir uma sequência de símbolos léxicos (tokens) que posteriormente serão utilizados pelo parser (presente na análise sintática).

2.2 Análise Sintática

A análise sintática é um processo do compilador, onde se analisa uma sequência que foi dada de entrada, utiliza os tokens da análise léxica como sua entrada, para verificar sua estrutura gramatical segundo uma determinada gramática.

2.3 Análise Semântica

É um processo de um compilador onde são verificados os erros semânticos (como por exemplo, divisão por zero, operações entre tipos de atributos diferentes). A análise

semântica trata a entrada da sintática e transforma em uma representação mais simples e, mais adaptada para a geração de código (etapa seguinte, e que faz parte da síntese).

3. Síntese

Segundo Aho, a síntese é dividida em 3 etapas, são elas: geração de código, otimização de código e geração de código final. Contudo, nesse trabalho abordaremos apenas a geração de código.

3.1 Geração de código

A geração de código consiste em utilizar a saída da análise semântica e transformar em uma representação de código, ou seja, uma sequência de instruções de máquina, normalmente em um programa assembly ou em uma outra linguagem de mesmo nível.

4. Implementação

Nesse tópico serão explicadas e demonstradas algumas escolhas do projeto do compilador, tais como: forma de implementação do compilador e estruturas utilizadas em cada parte da análise.

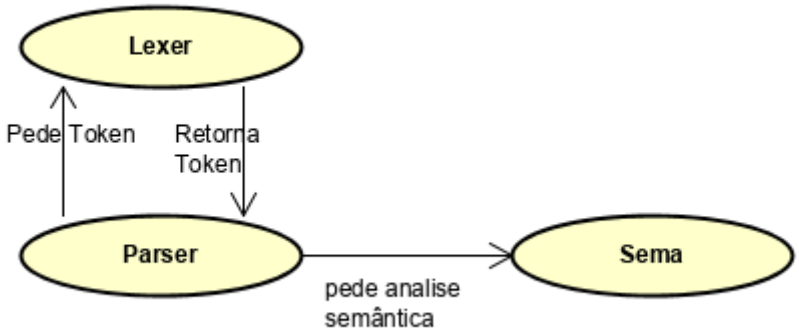

Figura 1. Apresentação da arquitetura geral do compilador

4.1. Léxico

Na implementação do analisador léxico utilizamos algumas estruturas auxiliares da linguagem C++, são elas “Map” e “Set”, ambos para guardar as palavras reservadas, o map para as compostas e o set para as palavras “simples”. O analisador funciona da seguinte forma, existe uma função “proximoToken()” que retorna o próximo token obtido da entrada, para o seu funcionamento é verificado o próximo caractere da entrada:

1- Se for um início válido de comentário (i.e. '#' em início de linha) ele irá chamar uma função que tokeniza comentário, esse token será ignorado nas etapas seguintes;

2- Caso seja um digito será chamada uma função que tokeniza número e dentro dessa função ele verifica os próximos caracteres, onde é verificado se excede ou não o tamanho máximo de um inteiro, se exceder é informado um erro léxico;

3- Se for uma letra será chamada a função tokenizaPalavra(), onde será tratado o caso da palavra composta e da palavra "simples". Para o tratamento da tokenização de palavras é feito um uma verificação se a primeira palavra lida está no set de palavras simples, caso esteja a palavra é tokenizada, senão utilizamos o map de palavras compostas para verificar se a palavra é um início válido, se for, tenta-se achar as próximas palavras para formar um token composto, caso todas as palavras sejam encontradas, cria-se um token composto. Caso a palavra lida não seja igual a nenhuma palavra reservada (palavra presente no set ou map) ela é tokenizada como id. Ainda sobre a tokenização de palavras, é importante constar que não existem distinções entre caracteres maiúsculos ou minúsculos, pois foi utilizada a função "toupper" para que todos fossem tratadas como caracteres em maiúsculos;

4- E caso seja lido algum caractere invalido, é informado um erro léxico e o caractere é ignorado.

5- Se não houver mais caracteres para serem lidos, um fim de texto é tokenizado, ele é usado posteriormente na análise sintática;

```
// Cria e retorna o próximo Token
Token* Lexer::proximoToken() {
    while(1) {
        char p = olhaProximoCaractere();
        if (espaco(p)) {    //se for espaçamento, vai pro próximo caractere
            obterCaractere();
        }
        else if (fimDeLinha(p)) {   //se for fim de linha, vai pro próximo caractere, mudando linha e coluna
            obterCaractere();
            linha++;
            coluna = 1;
        }
        else if (inicioDeComentario()) {    //se for inicio de comentário, tokeniza o comentário e retorna o token
            return tokenizaComentario();
        }
        else if (digito(p)) {   //se for dígito, tokeniza número e retorna o token
            return tokenizaNumero();
        }
        else if (letra(p)) {    //se for letra, tokeniza palavra e retorna o token
            return tokenizaPalavra();
        }
        else if (fimDeTexto(p)) {   //se for fim de texto, tokeniza fim de texto e retorna o token
            return tokenizaFim();
        }
        else {  //se for caractere invalido, informa erro léxico e retorna nulo
            erro(0, linha, coluna);
            obterCaractere();
            return nullptr;
        }
    }
}
```

Figura 2. Trecho do analisador léxico, que mostra a função proximoToken.

4.1. Sintático

No analisador sintático foi utilizado o Parser LL(1), devido a sua compatibilidade com a gramatica (não existência de conflitos) e por seu funcionamento e implementação estarem exemplificados de forma clara no livro "Compiladores: Princípios, técnicas e ferramentas".

A implementação do analisador sintático conta com uma tabela com todas as regras da gramática, presentes em um "map" de "set" e "vector" (estruturas auxiliares presentes em C++) funciona da seguinte forma:

1- A pilha é inicializada, são empilhados "$" e "PROGRAMA", e é solicitado o primeiro token;

2- Enquanto o topo da pilha não for vazio, será verificado se o topo da pilha é um terminal;

3- Se for um terminal, é verificado se o terminal é igual ao token atual (oriundo do léxico);

3.1-Se for igual, topo da pilha é desempilhado e a análise semântica (que irá ser abordada no próximo tópico) é chamada para o token atual e o próximo token é solicitado (exceto se o token atual for um fim de texto);

3.2-Se não for igual, gera um erro sintático. Se for, o topo da pilha é desempilhado, a análise semântica (que irá ser abordada no próximo tópico) é chamada para o token atual e o próximo token é solicitado (exceto se o token atual for um fim de texto, nesse caso a análise acaba, já que a pilha fica vazia);

4- Se não for um terminal, temos um não-terminal, logo verificamos na tabela se aquele não-terminal gera uma produção;

4.1-Se gerar, é chamada a função “obterInstrucao()” que coloca as produções na pilha, de forma que a primeira produção fique no topo;

4.2-Se não gerar, é sinalizado um erro sintático.

```
void Parser::analisa() {
    proximoToken();     //pega token inicial
    pilha.push("$");    //$ marca o fim da pilha
    pilha.push("PROGRAMA"); //regra inicial da gramatica

    while(not(pilha.empty())) { //enquanto houver instrução na pilha

        if (terminal(pilha.top())) {    //se o topo da pilha for terminal...
            if (pilha.top() == tokenAtual->obterLexema()) { //e o token atual for igual...
                if (pilha.top() != "$") {
                    sema->analisa(*tokenAtual, *tokenAnterior); //chama analise semantica pro token atual e...

                    proximoToken(); //pede o proximo token
                }
                pilha.pop();    //tira o terminal da pilha
            }
            else {  //caso o token atual não corresponda ao terminal no topo da pilha...
                erro(2, tokenAnterior->obterLinha(), tokenAnterior->obterColuna());   //informa erro sintático
                break;
            }
        }

        else {  //se o topo da pilha não for um terminal, então é uma regra
            if (instrucaoValida()) {
                obterInstrucao();   //coloca as instruções da regra na pilha
            }
            else {  //se a instrução não estiver na tabela...
                erro(2, tokenAnterior->obterLinha(), tokenAnterior->obterColuna());   //informa erro
                break;
            }
        }
    }

}
```

Figura 3. Trecho do analisador sintático.

5. Semântico

Na implementação do analisador semântico é usado apenas um “vector” que é usado para guardar os ids que já foram usados em declarações, as funções para o tratamento de cada caso de erro semântico são:

1- Conflito de sentidos, erro que acontece quando existe um “Vire para Direita” sucedido por “Vire para a Esquerda”, para o tratamento dele é utilizada a função “verificaConflitoSentido()”, onde é utilizada a flag “flagSentido”. A função utiliza a flag para verificar se anteriormente foi usado um comando de "Vire para" e qual o sentido ele teve. A partir disso é possível identificar um possível erro semântico;

2- Instrução “MOVA” N [PASSOS] não sucedida por “AGUARDE ATE ROBO PRONTO”, o tratamento também é feito de forma similar, com o uso de uma função que utiliza a flag “flagMova” para verificar se tal erro acontece ou não;

3- Conflito de ids (por exemplo, a declaração de um id já declarado anteriormente), o tratamento desse caso também é feito utilizando uma função, “verificaConflitoId()”, e uma flag, “flagDefinaInstrucao”, para determinar o conflito.

Lembrando que, a análise semântica é chamada pela sintática para analisar somente um token por vez.

5. Geração de código

A geração de código não foi feita por problemas com o uso do emulador. A geração seria feita caso o código de entrada não contiver erros léxicos, sintáticos e semânticos. O código de entrada seria transformado por um correspondente em assembly, utilizando detalhes da aplicação robot, do emu8086**.**

6. Exemplificação

Para exemplificar o funcionamento do compilador, utilizaremos o exemplo contido na especificação do projeto:

```
1  PROGRAMAINICIO
2      DEFINAINSTRUCAO Trilha COMO
3      INICIO
4          Mova 3 Passos
5          Aguarde Até Robô Pronto
6          Vire Para ESQUERDA
7          Apagar Lâmpada
8          Vire Para DIREITA
9          Mova 1 Passo
10         Aguarde Até Robô Pronto
11     FIM
12     EXECUCAOINICIO
13         Repita 3 VEZES Trilha
14         Vire Para DIREITA
15     FIMEXECUCAO
16 FIMPROGRAMA
```

Figura 4. Código da especificação

Nesse código existem alguns erros:

1- Acentos não são aceitos. Linhas 5, 7 e 10.

2- Má formação de EXECUCAOINICIO, onde deveria haver um bloco, delimitado por INICIO e FIM, para suportar mais de um comando.

Corrigindo esses erros, temos o código:

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

Figura 5. Código da especificação corrigido.

No próximo exemplo temos um código com mais erros que serão detectados:

```
1  PROGRAMAINICIO
2      DEFINAINSTRUCAO Trilha COMO
3      INICIO
4          Mova 346487897145 Passos
5          Aguarde Ate Robo PARADO
6          Vire Para ESQUERDA
7          Apague Lampada
8          Vire Para direita
9          Mova 1 Passo
10         Aguarde Ate Robo Pronto
11     DEFINAINSTRUCAO Trilha COMO
12     INICIO
13         Mova 2 Passos
14         Aguarde Ate Robo PARADO
15         Vire Para ESQUERDA
16         Apague Lampada
17         Vire Para direita
18     FIM
19     EXECUCAOINICIO
20         INICIO
21             Repita 5 VEZES Trilha FIMREPITA
22             Vire Para DIREITA
23             vire para esquerda
24         FIM
25     FIMEXECUCAO
26 FIMPROGRAMA
```

Figura 6. Código de exemplo

Nesse código os erros encontrados são:

1- Número que excede limite de int na linha 4, erro léxico.

2- Falta de FIM ao bloco, deveria estar entre as linhas 10 e 11, erro sintático.

3- MOVA N sem AGUARDE ATE ROBO PRONTO, linhas 5 e 15, erro semântico.

4- Duas instruções definidas com o mesmo nome, erro semântico.

5- VIRE PARA DIREITA seguido de VIRE PARA ESQUERDA, linhas 22 e 23, erro semântico.

Para consertar os erros, faremos:

1- Diminuir número.

2- Adicionar comando FIM.

3- Trocar AGUARDE ATE ROBO PARADO por AGUARDE ATE ROBO PRONTO.

4- Trocar nome da segunda instrução para Trilha2.

5- Adicionar um PARE entre os dois comandos.

Com isso, temos:

```
1  PROGRAMAINICIO
2      DEFINAINSTRUCAO Trilha COMO
3      INICIO
4          Mova 34648 Passos
5          Aguarde Ate Robo PRONTO
6          Vire Para ESQUERDA
7          Apague Lampada
8          Vire Para direita
9          Mova 1 Passo
10         Aguarde Ate Robo Pronto
11     FIM
12     DEFINAINSTRUCAO Trilha2 COMO
13     INICIO
14         Mova 2 Passos
15         Aguarde Ate Robo PRONTO
16         Vire Para ESQUERDA
17         Apague Lampada
18         Vire Para direita
19     FIM
20     EXECUCAOINICIO
21         INICIO
22             Repita 5 VEZES Trilha FIMREPITA
23             Vire Para DIREITA
24             PARE
25             vire para esquerda
26         FIM
27     FIMEXECUCAO
28 FIMPROGRAMA
```

Figura 7. Código de exemplo corrigido