Relatório do Trabalho da Disciplina de Compiladores

Francisco M.L. Machado

Universidade Federal da Bahia – Salvador, BA, Brasil

franciscomontenegrolm@gmail.com

Resumo: *Este artigo visa relatar todo o processo do projeto de implementação de um compilador para a linguagem Robot-L, abordando desde as decisões de projeto até trechos de código-fonte que ilustrem tais decisões.*

1. Informações gerais do projeto

O código-fonte deste trabalho foi feito inteiramente em *C++*, na plataforma *Windows* e utilizando o compilador do *g++* como compilador padrão. Este projeto corresponde a 3 pontos na disciplina de Compiladores, ministrada pela professora Vaninha Vieira.

O repositório para todo o código-fonte discutido aqui pode ser encontrado em: https://github.com/furran/mata61.

2. A Escolha da linguagem

A escolha de *C++* como linguagem de programação para a implementação deste projeto prático teve várias razões:

i) Certa familiaridade com a linguagem;

ii) A ferramenta extremamente poderosa que são os ponteiros;

iii) Poder modularizar cada etapa do projeto seguindo uma abordagem de POO;

iv) A capacidade de programar em mais baixo nível do que em outras linguagens.

A principal razão, porém, foi que essas características da linguagem permitiam um maior aproveitamento didático do projeto. A capacidade de construir explicitamente as árvores sintáticas e poder modularizar as etapas do compilador nos permite ver mais claramente a teoria estudada em sala de aula em sua própria implementação, o que não seria necessariamente dessa forma se fosse escolhida uma linguagem de mais alto nível, como Java, por exemplo.

3. O Token

O *Token* é um classe implementada para representar os tokens a serem reconhecidos durante a análise léxica. Porém, eles também funcionam como a estrutura que guarda outros dados, como o lexema que o compõe, a linha em que foi criado, e uma *"tag"*, que é um valor numérico que representa o token de fato.

4. O Analisador Léxico

```
class Lexer{
private:
        char peek;
        Token* curToken;
        Buffer buffer;
        int line;
        std::unordered_map<std::string, Token>
reservedWords;
public:
        Lexer(std::string filename);
        int getLinesRead();
        void install(Token w);
        void deleteUntilDelimiter();
        Token* getCurrentToken();
        Token* scan();
};
```

Figura 1. Classe Lexer

Correspondente à primeira etapa do projeto, o Analisador Léxico foi implementado como uma classe de nome *Lexer*, com um par de buffers implementados na classe *Buffer* e com 42 palavras reservadas simples — e da forma como veremos à seguir.

4.1. As Palavras Reservadas e os Lexemas

Existem 42 palavras reservadas:

programainicio, fimprograma, execucaoinicio,

fimexecucao, definainstrucao, como, inicio, fim, repita,

vezes, fimrepita, enquanto, faca, fimpara, se, entao,

fimse, senao, fimsenao, mova, passos, vire, para, pare,

finalize, apague, lampada, acenda, aguarde, ate, robo,

pronto, ocupado, parado, movimentando, bloqueada, acesa,

a, frente, apagada, esquerda, direita

Todas as palavras reservadas são colocadas numa tabela hash que será utilizada para verificar se a sequência de caracteres lida pelo analisador léxico é um identificador ou uma palavra chave.

As palavras chave nunca são compostas, ficando como trabalho do analisador sintático juntar essas palavras.

Além das palavras chaves, temos dois outros possíveis tokens: **identificadores** e **números**. Identificadores começam com uma letra ou ‘_’, seguidos de números, letras ou ‘_’, note que identificadores não podem ser iguais às palavras chaves. Números podem apenas ser formados por dígitos.

Há também tokens especiais, chamados **error** e **end_of_file**, que permite que a análise sintática prossiga mesmo com erros léxicos e informa que o fim do arquivo foi reconhecido, respectivamente.

4.2. O Buffer de Caracteres

Ler diretamente do arquivo, um caractere de cada vez, tende a ser uma operação lenta e ineficiente. Por isso, ao invés de ler um caractere de vez, é muito mais eficiente ler um arquivo em blocos e salvar estes blocos em um buffer, e reenchendo o buffer toda vez que todos os seus caracteres já foram lidos.

Ter apenas um buffer, porém, pode gerar problemas. Às vezes precisamos examinar um ou mais caracteres a frente do lexema para nos certificar que temos o lexema certo. Por exemplo, quando estamos reconhecendo um identificador, só sabemos que chegamos no fim do lexema quando encontramos um caractere que não pode pertencer à um identificador — no caso deste projeto pode ser um delimitador (espaço, tab, etc) ou um caractere especial (#,@,etc).

Para evitar isso, implementa-se na classe *Buffer* um par de buffers que tratam de lookaheads grandes e se alternam em carregar o bloco de caracteres do arquivo, assim contornando este possível problema.

4.3. A Classe Lexer e o Reconhecimento do Tokens

A classe *Lexer* (*Figura 1*) tem 4 atributos: *peek*, que guarda o caractere atual da análise; *curToken* que guarda o último *Token* reconhecido pelo analisador léxico; *buffer*, que é simplesmente um par de buffers implementados em uma única classe; *line*, para dizer em que linha o erro foi encontrado; e reservedWords, que é a tabela hash que será inicializada como todas as palavras reservadas.

A função *scan()* é responsável pelo reconhecimento dos tokens, eliminando símbolos inúteis, lendo as cadeias de caracteres e gerando um token de acordo. Vejamos como são eliminados espaços, tabs, símbolos especiais e comentários:

```
//consome caracteres em branco, linhas de comentário
//,LF,CR até o próximo caractere válido.
while (true) {
   if (peek == ' ' || peek == '\t' || peek == '\r');
   else if (peek == '\n')
      line++;
   else if (peek == '#') {
      while (true) {
      peek = buffer.next();
         if (peek == '\n') {
            line++;
            break;
         } else if (peek == END_FILE) {
            curToken = new Token(END_OF_FILE, line, "");
            return curToken;
         }
      }
   } else break;
   peek = buffer.next();
}
```

Figura 2. Trecho da função Lexer::scan() que consome caracteres inúteis

O seguinte trecho de código reconhece identificadores e/ou palavras reservadas (simplificado — omitindo reconhecimento de erros,etc):

```
//tokeniza id ou keyword
if (isalpha(peek) || peek == '_') {
   std::string lex("");
   lex += peek;
   char look = buffer.lookAhead();
   while (isdigit(look) || isalpha(look) || look == '_') {
      peek = buffer.next();
      look = buffer.lookAhead();
      lex += peek;
   }
   std::unordered_map<std::string, Token>::iterator it =
reservedWords.find(lex);
   if (it != reservedWords.end()) {
       curToken = new Token(it->second.getTag(),line,it-
>second.getLexeme());
       return curToken;
   }
   curToken = new Token(ID,line ,lex);
   return curToken;
}
```

Figura 3. Reconhecimento de identificadores/palavras reservadas

Para o reconhecimento de todos os outros possíveis tokens, a lógica é praticamente a mesma.

5. A Análise Sintática

Para a análise sintática, foi escolhida uma abordagem de análise descendente preditiva não-recursiva, utilizando uma gramática LL(1) adaptada da gramática na especificação do projeto. A gramática pode ser encontrada entre os arquivos do repositório com o nome "gramatica.pdf".

```
class Parser{
public:
   Lexer lexer;
   std::unordered_map<pair,std::vector<int>,hash_pair> analysisTable;

   Parser(std::string filename);   virtual ~Parser();
   node * parse();
};
```

Figura 4. Classe Parser

A análise sintática consiste em utilizar de uma pilha de símbolos da gramática, preemptivamente contendo os símbolos *$* e *pPROGRAMA*, que servem pra indicar o fim da análise e derivar o início da análise, respectivamente.

O analisador considera o símbolo ***X*** no topo da pilha e o símbolo ***t*** corrente na entrada, e então:

1. Caso os símbolos ***X*** e ***t*** batam, então desempilha ***X*** e tira ***t*** da entrada;
2. Caso ***X*** é um terminal, e não bate com o símbolo na entrada, então encontramos um erro sintático;
3. Caso ***X*** seja uma variável, então verificamos na tabela de análise a produção mapeada para as chaves [***X,t***] na tabela, criamos um *node* (nó) com símbolo ***X***, e então empilhamos todos os símbolos do lado direito da produção ao mesmo tempo que criamos um *node* para cada símbolo e colocamos-os como filhos do nó de ***X***, gerando a árvore sintática, sem remover ***t*** da entrada;
4. Caso ***X*** seja uma variável, mas não haja produção mapeada na tabela para as chaves [***X,t***], então encontramos um erro sintático.

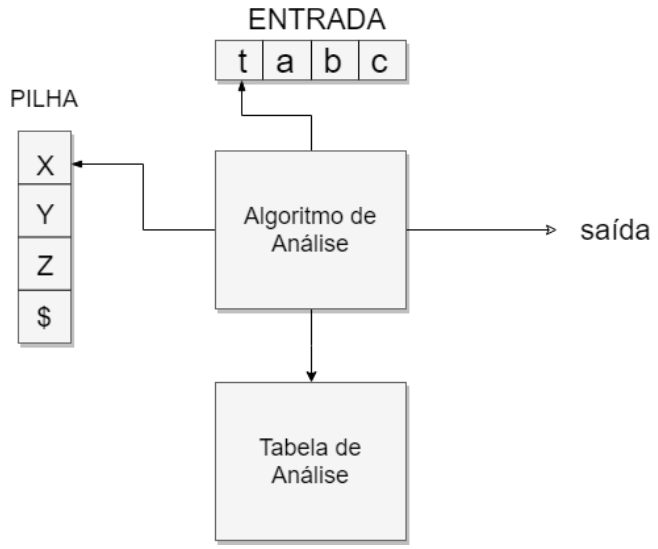

Figura 5. Ilustração do processo de análise

No final da análise, quando todos todos os itens da pilha foram desempilhados, a função *parse()* retorna um *node* correspondente a raíz da árvore sintática produzida.

Nodes são uma estrutura implementada para guardar um ponteiro para um token e um vetor de ponteiros para os seus filhos. Eles formam os nós da árvore sintática gerada.

```
while(x!=$){
t = lexer.getCurrentToken();
if(x==(*t).getTag()){ // caso 1
   s.pop();
   lexer.scan();
   cur->token = t;
}
else if(isTerminal(x)){ // caso 2
   s.pop();
   lexer.scan();
   tmp = new node;
   tmp->token = t;
   cur->children.push_back(tmp);
   printf("ERRO::LINHA:%d: Terminal nao bate.\n", (*t).getLine());
}else if (isNonTerminal(x)) { //caso 3
   pai = s.top();
   s.pop();
   it = analysisTable.find(std::make_pair(x, (*t).getTag()));
   if (it != analysisTable.end()) {
      node *arr[(it->second.size()];
      for (int i = (it->second.size() - 1; i >= 0; i--) {
         arr[i] = new node;
         (arr[i])->token = new Token((it->second,0,"");
         s.push(arr[i]);
      }
      cur->children.insert(cur->children.begin(), arr,arr + (it-
>second.size());
   } else { //caso 4
      it = analysisTable.find(std::make_pair(x, VAZIO));
      if (it != analysisTable.end()) {
         tmp = new node;
         tmp->token = new Token(VAZIO,0,"");
         cur->children.push_back(tmp);
      } else
      printf("ERRO::LINHA:%d: Producao incapaz de gerar o
token.\n",(*t).getLine());
}
   cur = s.top();
   x = s.top()->token->getTag();
}
return head; //retorna raiz da arvore
```

Figura 6. Trecho de código (simplificado) da função Parser::parse().

5. A Análise Semântica

A análise semântica ocorre ao mesmo tempo que a análise sintática, se aproveitando da tabela de análise, e das regras semânticas inseridas nela. Isto é, para certas produções mapeadas na tabela, além de produzirem os terminais e as variáveis que serão utilizadas na análise sintática, elas também produzem "regras" semânticas, que são então empilhadas junto com os outros símbolos da gramática.

Então, além dos quatro casos mencionados na análise sintática para os símbolos no topo da pilha, existe um quinto caso:

5. Caso ***X*** seja uma ***REGRA***, então realizar ação semântica compatível com a ***REGRA***.

6. A Geração de Código

A análise semântica foi até aonde eu consegui chegar, infelizmente.