Analisador Léxico FORMAT

Breno Cardoso Elaine Lima João Pinto Viviane Pinheiro

Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada

Agosto, 2015

Este é o segundo documento da série de documentos do projeto da disciplina de Compiladores – 2015.2, ministrada pelo prof. *Umberto Souza da Costa*, nesta instituição. Visa-se neste momento implementar e documentar o analisador léxico da linguagem e descrever os passos para a sua execução, bem como resumir as etapas básicas presentes na compilação de uma linguagem. Em anexo há o código desenvolvido até o momento.

1. Análise léxica e o processo de compilação

Posto de forma simples, um compilador é responsável por ler um programa escrito em uma linguagem fonte e o traduzir em um programa equivalente na linguagem alvo. O processo completo de compilação pode ser dividido basicamente em duas fases: análise e síntese. O primeiro divide o programa fonte em partes constituintes e cria uma representação intermediaria do mesmo. As operações são determinadas e registradas numa estrutura hierárquica, chamada árvore sintática. A síntese constrói o programa alvo a partir dessa representação intermediária.

A análise por sua vez consiste em 3 fases: a) **análise léxica**, linear ou esquadrinhamento (*scanning*). O fluxo de caracteres é lido e agrupado em *tokens* (sequência de caracteres com significado coletivo) b) **análise sintática**, hierárquica ou gramatical. Os *tokens* são agrupados hierarquicamente em coleções aninhadas com significado coletivo; agrupados em frases gramaticais, representadas por árvores gramaticais. c) **análise semântica**. Verificações são realizadas para assegurar que os componentes de um programa se combinam de forma significativa e para isso utiliza a estrutura hierarquia da fase anterior. Um importante componente desta fase é a verificação de tipos, onde é verificado se o operador recebe os operandos que lhe são permitidos.

O processo de compilação não é exatamente linear. E o analisador léxico pode ser visto como uma subrotina do sintático. Dessa forma, o programa escrito na linguagem original é lido, dividido em pequenas partes, e essas estruturas são armazenados em uma **tabela de símbolos** para uso posterior pelo analisador sintático e são passadas ao analisador semântico. O objetivo desta etapa é a construção do analisador léxico. O resultado de sua análise é posto em uma tabela de símbolos para que na fase posterior seja acrescentado informações a essa tabela e as devidas decisões sejam tomadas, tais como a checagem de tipos. Na implementação apresentada aqui, somente o reconhecimento dos *tokens* ocorre.

Para essa implementação utilizamos uma ferramenta de geração de analisadores léxicos, chamada Lex. Isso permite simplificar a sua construção, assim podemos relacionar a gramática criada no problema anterior com as expressões regulares que relacionam os lexemas da linguagem com os

seus padrões e assim chegar aos *tokens* que são vinculados àquele lexema. Acredita-se que a próxima etapa seja a construção do analisador sintático, que fará uso do código produzido nesta etapa. Dessa forma, as funções que estão ausente e que se relacionam com o analisador sintático serão produzidas efetivamente na próxima etapa.

Após as análises sintática e semânticas, alguns compiladores geram uma representação intermediária explícita do programa fonte. Essa representação pode ter uma variedade de formas; uma é a chamada "código de três endereços", que é semelhante a uma linguagem de montagem. Esta forma possui várias propriedades que favorecem seu uso. Por exemplo, cada instrução há no máximo um operador, além da atribuição. **Otimização de código** é uma fase posterior a **geração de código intermediário**, na qual se tenta melhorar o código intermediário, de tal forma que venha resultar um código de máquina mais rápido em tempo de execução. Existe uma grande variação na quantidade de otimizações de código de máquina que cada compilador pode executar. No entanto, algumas otimizações podem melhorar significativamente o tempo de execução sem alongar de mais o tempo de compilação. **Geração de código** é a fase final da compilação e trata-se da geração do código alvo propriamente, consistindo normalmente de código de máquina realocável ou código de montagem. O aspecto crucial nessa fase é a atribuição das variáveis aos registradores. A Figura 1 apresenta um esquema que resume o que foi discutido até o momento.

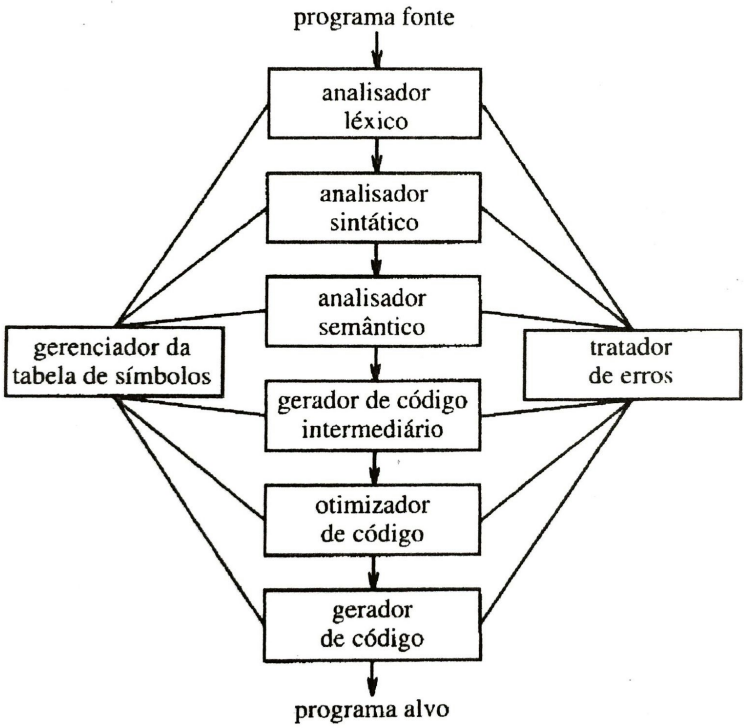

Fig. 1.9 Fases de um compilador.

Figura 1: *Fases de um compilador.*

2. Documentação

A construção do analisador léxico pode ser realizada de forma automatizada através de uma ferramenta chamada Lex ou pode ser construída do zero. Muito dificilmente se obterar um analisador mais eficiente comparando ao produzido pelo Lex. Outro benefício de utiliza-lo é a praticidade da especificação e desenvolvimento do compilador. Dessa forma, para o desenvolvimento do analisador léxico foi utilizada esta ferramenta. Seu objetivo é separar a entrada em *tokens* (unidades que possuem sentido coletivo). E para isso é fornecido um conjunto de descrições em expressões regulares bem formadas, ele produzirá uma rotina na linguagem C que reconhecerá unidades que obedeçam as expressões regulares definidas especificação lex.

Uma especificação léxica é dividida em três partes: a) na primeira seção, chamada **seção de definição**, limitada por %%, pode conter comandos iniciais em C (delimitados por "%{" e "%}"), os quais serão utilizados integralmente no arquivo C que é gerado pelo *lex:* `lex.yy.c`; b) na segunda seção, chamada **seção de regras**, delimitada por "%%" e "%%", deve conter as regras definidas em expressões regulares e as ações em código C que devem ser executadas quando uma estrutura correspondente aquela expressão é encontrada; c) na terceira e última seção, chamada **subrotinas de usuário**, pode conter qualquer tipo de código C. Pode também conter uma função `main()` com uma chamada ao procedimento `yylex()` em seu interior, pois esta é a chamada do analisador léxico.

a. Analisador léxico da linguagem Format

O analisador léxico irá reconhecer as seguintes palavras reservadas, identificadores, números e delimitadores que foram definidos no documento anterior.

i. Palavras reservadas

Palavras-chave são palavras que têm significado especial para o compilador. São reservadas e não podem ser utilizadas como identificadores comuns, uma vez que elas já possuem um significado especial na linguagem. São elas:

int	real	complex	single_real	double_real
string	enum	struct	end_struct	matrix_of
set_of	const	ref	if	else
for	while	switch	case	break
other	end_if	end_for	end_while	end_switch
return	function	end_function	procedure	end_procedure
module	end_module	null	true	false

ii. Identificadores

Identificadores são os nomes que se fornece para as variáveis, tipos, funções e rótulos em um programa. Nomes de identificador deve diferir em ortografia e caixa de quaisquer palavras-chave. Não se pode usar palavras-chave como identificadores; eles são reservados para uso especial. Se cria um identificador, especificando-o na declaração de uma variável, tipo ou função. O identificador deve seguir a seguinte regra na sua formação:

```
LETRA       [a-zA-Z]
DIGITO      [0-9]
ID          ({LETRA}|_)({LETRA}|{DIGITO}|_)*
```

iii. Literais

Uma expressão é uma construção que vai ser avaliada, para se obter um valor. O tipo mais simples de expressão é um literal, o qual possui um valor fixo de algum tipo. Aqui estão alguns exemplos típicos de literais em linguagens de programação:

```
365          3.1416          false          "O quê?"
```

Estes denotam um número inteiro, um número real, um booleano, e uma *string*, respectivamente. Merecem destaque para a forma de representação aceita pela linguagem FORMAT, os seguintes tipos de literais:

Pontos flutuantes. Os literais de ponto flutuante na linguagem FORMAT segue o padrão adotado em grande parte das linguagens de programação, sendo descritos pelas seguintes definições lexicais a seguir:

```
DIGITO      [0-9]
EXPOENTE    (e|E)[+-]?{DIGITO}+
NUM_INT     {DIGITO}+
NUM_PONTO   {NUM_INT}"."{DIGITO}+
NUM_REAL    {NUM_PONTO}|{NUM_PONTO}{EXPOENTE}
```

Imaginários. Os Literais imaginários são descritos pelas seguintes definições lexicais:

```
NUM_INT     {DIGITO}+
NUM_PONTO   {NUM_INT}"."{DIGITO}+
NUM_COM     ({NUM_PONTO}|{NUM_INT})(i|I)
```

Um literal imaginário produz um número complexo com uma parte real igual a zero. Os números complexos são representados como um par de números de ponto flutuante e têm as mesmas restrições adotada a eles. Para se criar um número complexo com uma parte real diferente de zero basta adicionar um número de ponto flutuante a ele, por exemplo, (3 + 4i). Em Fortran a representação visual se aproxima de um par ordenado. No entanto, FORMAT adota o formato mais flexível presente em MATLAB.

iv. Operadores

Há vários tipos de operadores em uma linguagem de programação: aritiméticos, lógicos, relacionais e de atribuição. As convenções adotadas em FORMAT são elencadas a seguir e seu significado/uso é igual ao adotado na grande maioria das linguagens. Quanto à precedência os operadores seguem a usual.

Operadores aritméticos	`+ - * /`
Operadores comparação	`< > <= >= <>`
Operadores lógicos	`&& || !`
Exponenciação	`^`
Atribuição	`=`
Atribuição e operação aritmética	`+= -= *= /=`

v. Delimitadores

Os *tokens* listados a seguir servem como delimitadores na gramática. Segue quando eles são utilizados e exemplos de sua utilização.

Delimitação de expressões lógicas e parametros de funções e procedimentos	`( )`	`if( n <> 0 )`
Delimitação de acessos e faixas em matrizes	`[ ]`	`matrix_of int [12] m`
Delimitação de comandos (marca seu fim)	`;`	`++k;`
Delimitação de faixa entre números	`..`	`1..4`

3. Compilação e execução

Em anexo a este documento vai um conjunto de arquivos com código exemplo para testar a implementação do analisador léxico. São eles: `eg_fatorial_func.format`, `eg_mergesort.format`, `eg_insertionsort.format`. Além desses arquivos estão presentes o arquivo `lex.l` com as especificações da linguagem FORMAT, juntamente do arquivo que representa a tabela de símbolos `y.tab.h` e o código gerado pelo Lex `lex.yy.c`.

A compilação segue o processo usual, nos sistemas baseado em UNIX procede como mostrado a seguir.

```
% lex lex.l
% cc lex.yy.c -o <executavel> -ll
```

O executável produzido é capaz de receber arquivos de programas fonte, como também em sua ausência receber comandos interativos. A execução se dá, supondo um executável `format.analizer` como segue.

```
% ./format.analizer eg_fatorial_func.format
```

O arquivo citado no comando anterior contém o código listado a seguir.

```
function int factorial ( int n ) :
   if( n <> 0 ) :
      return 1;
   end_if;
   return n * factorial( n - 1 );
end_function

int x = factorial(4);
```

Afim de ilustração, caso o analisador léxico seja executado sobre o código anterior o resultado retornado seria o listado abaixo.

```
COMMENT
KEYWORD FUNCTION
KEYWORD INT
ID
DELIMITER OPEN_PARENTHESIS
KEYWORD INT
ID
DELIMITER CLOSE_PARENTHESIS
DELIMITER COLON
KEYWORD IF
DELIMITER OPEN_PARENTHESIS
ID
OPERATOR NOT EQUAL
INTEGER NUMBER
DELIMITER CLOSE_PARENTHESIS
DELIMITER COLON
KEYWORD RETURN
INTEGER NUMBER
SEMICOLON
KEYWORD END_IF
KEYWORD RETURN
ID
MULT OPERATOR
ID
DELIMITER OPEN_PARENTHESIS
ID
SUB OPERATOR
INTEGER NUMBER
DELIMITER CLOSE_PARENTHESIS
```

```
SEMICOLON
KEYWORD END_FUNCTION
KEYWORD INT
ID
OPERATOR ASSIGN
ID
DELIMITER OPEN_PARENTHESIS
INTEGER NUMBER
DELIMITER CLOSE_PARENTHESIS
SEMICOLON
```

4. Referências bibliográficas

Appel, A. and Ginsburg, M. (1998). *Modern compiler implementation in C*. Cambridge: Cambridge University Press.

Levine, J., Mason, T. and Brown, D. (1995). *Lex and Yacc*. [s. l.]: O'Reilly.