Interfaces de Vanguarda do Compilador

Stefani Henrique Ramalho[1], Prof Mário Rubens Welerson Sott[1]

[1]DCC – Departamento de Ciência da Computação – Universidade Presidente Antônio Carlos (UNIPAC)

Barbacena – MG – Brasil

stefanidc@gmail.com, mrws@powerline.com.br

Resumo. *Este artigo tem por finalidade demonstrar de uma forma abstrata o funcionamento das fases da vanguarda do compilador e fazer um estudo comparativo entre linguagens de programação, a fim de ratificar o conceito de transportabilidade do compilador, através dos processos de análise léxica, sintática e semântica, demonstrando o funcionamento de sua representação intermediária.*

Palavras chave: *compilador, análise léxica, análise sintática e código intermediário.*

1. Introdução

A construção de compiladores estende-se devido às linguagens de programações que fazem a comunicação entre um indivíduo que deseja resolver uma determinada situação e a máquina que irá ajudá-lo a resolver esse problema. A conversão entre essas linguagens é feita através dos compiladores.

Compilação é um processo de tradução que lê um programa escrito em uma linguagem fonte, e o transforma em uma linguagem alvo equivalente, podendo ser, então, outra linguagem de programação ou uma linguagem de máquina, da família dos computadores.

A partir de uma divisão que ocorre dentro do compilador, podemos representa-las como vanguarda e retaguarda. A vanguarda irá atender as fases relacionadas com a análise do código fonte e a retaguarda com a síntese do código fonte em linguagem alvo, assim, idealizando que podemos dividir um compilador em fases independentes.

Para argumentar a idéia de que o compilador pode ser dividido em fases, sendo que a retaguarda poderá ser comum entra as vanguardas, será feito um estudo de caso entre as linguagens C e Pascal, a fim de gerar um único código intermediário entendido pela fase final do compilador.

2. Processo de Compilação

Compiladores são programas tradutores que recebem uma linguagem – Linguagem fonte – e a traduz em uma outra linguagem equivalente – linguagem alvo. Uma de suas principais atividades é informar a presença de erros [3].

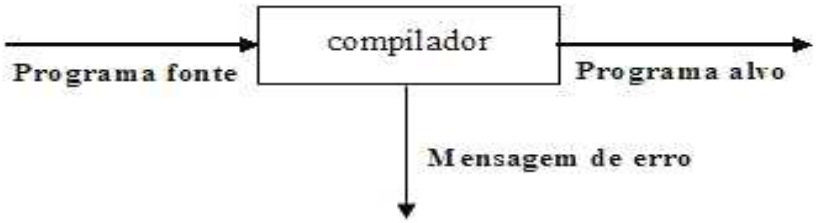

Figura 1. Compilador

Existem inúmeras linguagens fontes que vão das linguagens de programações tradicionais, como Pascal, *Fortran*, às linguagens de 4ª geração, como o SQL, *Framework* [1]. O código alvo pode ser uma linguagem de máquina ou outra linguagem de programação. Mesmo com inúmeras linguagens, a função do compilador é basicamente a mesma, por isso, conseguimos construir inúmeros códigos fontes e alvos, mas sempre utilizando as mesmas técnicas fundamentais [2].

2.1 Análise e Síntese

As fases de análise e síntese, nada mais são que uma divisão abstrata, que ocorre dentro do compilador, sendo que, na análise o compilador deve verificar se o programa é válido, por qualquer motivo enquanto a síntese é a geração do código alvo desejado [3].

A fase de análise engloba o analisador léxico, sintático e semântico, que geram uma representação intermediária do código e registra a informação obtida, em uma estrutura denominada árvore sintática, em que cada nó representa uma operação e o filho de um nó representa um argumento de uma operação.

Qualquer fase da análise que for detectado a presença de erros deve-se prosseguir com seu processo normalmente, evitando assim que a compilação seja interrompida, de modo que todo o código seja lido [2], pois pode ocorrer a presença de outros erros, posteriormente.

A síntese constrói um programa alvo, a partir da representação intermediária realizada pela análise. A síntese percorre a árvore sintática visitando todos os nós em uma ordem pré-definida, para gerar uma linguagem de máquina [1].

3. Gramáticas

Gramática nada mais é que um conjunto de regras que especificam a sintaxe de uma linguagem [1]. No processo de compilação faz-se necessário o uso de gramáticas, como meio de reconhecer se o código está seguindo as leis de formação da linguagem.

As fases de análise utilizam à gramática, como forma de garantir que código fonte atenda os padrões sintáticos definidos nas linguagens de programação, através de reconhecedores.

Para se criar compiladores, seria necessária a implementação e entendimento das gramáticas, já que elas tratam do funcionamento dos compiladores.

3.1 Reconhecedores

Os reconhecedores são mecanismos utilizados como forma de determinar se as regras de uma linguagem estão sendo seguidas, para isso existem três notações para a representação de reconhecedores, a produção de autômatos, a tabela de transições e o diagrama de estados [5].

Autômato é um reconhecedor que determina uma linguagem utilizando o conceito de estado [5].

Segue abaixo um exemplo de um autômato “M” que reconheça números inteiros e reais [3].

- M = (K, Σ, §, e0, F)
- K = (E0, E1, E2, E0)
- Σ = (d,.)
- §(E0,d) -> E1
- §(E1,d) -> E1
- §(E1,.) -> E2
- §(E2,d) -> E3
- §(E3,d) -> E3

· e0 = E0

· F = (E1 , E3)

Seguindo o exemplo utilizado, iremos representar um autômato através da tabela de transições.

	d	.
--> E0	E1	
(E1)	E1	E2
E2	E3	
(E3)	E3	

Tabela 1. Tabela de transição [3].

Por último, temos a representação do autômato através de diagramas de estados.

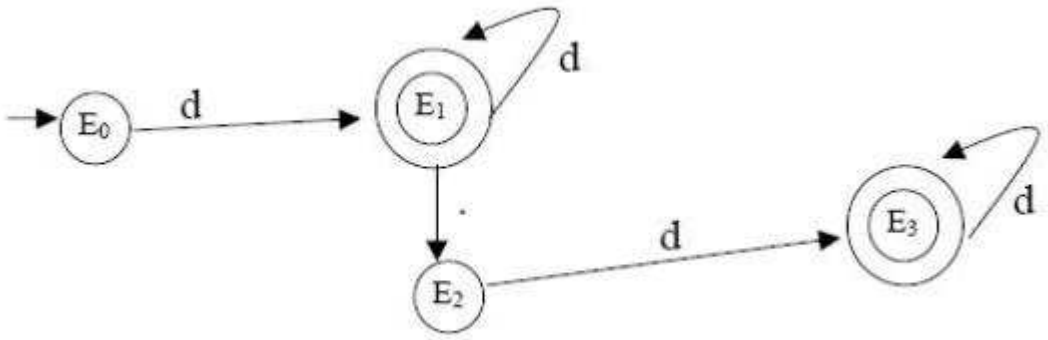

Figura 2. Diagrama de transições [3]

4. Análise Léxica

A análise léxica ou scanner é a primeira fase do processo de compilação que lê todos os caracteres do programa fonte, gera uma seqüência de símbolos individuais e os agrupam em “palavras” designadas por *tokens* [1], os *tokens* podem ser palavras reservadas, identificadores, operadores etc., que são atribuídos a uma tabela de símbolos. Durante essa fase são ignorados os espaços em brancos e comentários.

O analisador léxico identifica se o token é ou não uma palavra reconhecida pela linguagem, retornando mensagem de erro, caso seja necessário [4].

4.1 Tokens

Na varredura do programa fonte, são identificados os símbolos desse código que são denominados *tokens*, que podem ser identificadores, palavras reservadas, operadores lógicos, aritméticos etc [1].

Cada token pode ser especificado por três informações que serão aplicadas na tabela de símbolos.

- A classe que irá identificar qual será seu tipo, como por exemplo, palavras reservadas, identificadores, operadores etc [4];
- O valor que depende diretamente de sua classe, pois se trata do conteúdo do símbolo léxico, pode ser token simples que não tem nenhum valor associado, como por exemplo, palavras reservadas e operadores, ou token com argumentos que são os identificadores e constantes que possuem valores associados [4];
- Por último será determinada a posição em que se encontra o token, a fim de identificar, caso exista a presença de erros, de modo a facilitar a consulta e localização do programador [4];

4.2 Funções do Analisador Léxico

Uma linguagem de programação possui um conjunto de palavras (alfabeto) [5], que seguem um determinado padrão, atribuído pelo compilador, o analisador léxico verifica se as palavras contidas no código fonte são válidas.

Durante a varredura do programa fonte, para que se determine o tipo de token encontrado são utilizadas as expressões regulares, como mecanismo de geração de sentenças que seguem uma determinada regra de produção para que faça seu reconhecimento.

4.3 Tabela de Símbolos

A Tabela de símbolos é uma estrutura de dados, onde são armazenadas as informações obtidas pelas fases de análise no programa fonte, como os identificadores, tipos, formas, tamanho, localização na memória e outras características que dependem

de cada linguagem [1]. À medida que o analisador léxico varre o código fonte, são identificados os *tokens* e registradas as informações na tabela de símbolos [5].

As Informações contidas na tabela de símbolos farão interações com o analisador sintático, de forma que, a cada token registrado é reconhecido pelo analisador léxico, posteriormente será registrado em uma árvore, de tal forma que a tabela de símbolos intermedeie essas duas fases de análise [3].

5. Análise Sintática

A análise sintática ou *parser* é a segunda fase de um compilador, em que sua principal função é verificar através de uma gramática livre de contexto, se uma determinada cadeia de *tokens* que recebe através do analisador léxico, segue uma estrutura sintática e se seus elementos são organizados em sentenças definidas pela linguagem de programação [1].

O analisador léxico é uma sub-rotina do analisador sintático, sendo que a tabela de símbolo faz a interação entre essas duas fases de análise.

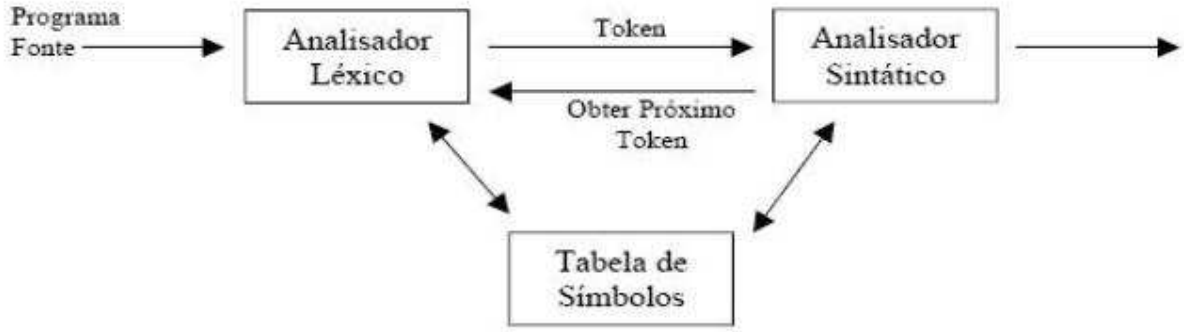

Figura 3. Interação entre análise léxica e sintática

O analisador sintático recebe uma seqüência de *tokens* e constrói uma árvore gramatical, em seguida verifica se a sentença gerada faz parte da sintaxe definida pela linguagem [3].

5.1 Análise Sintática Top-Down

O Sistema de análise sintática *top-down* constrói uma árvore gramatical a partir do primeiro token denominada raiz [1], logo após, vasculha toda árvore, até atingir cada terminal de acordo com as regras de produções da linguagem. Essa análise sempre será feita iniciando-se pela raiz até ás folhas.

A análise sintática *top-down* possui duas formas de “implementações”, a descida recursiva que faz a derivação mais à esquerda de uma cadeia de entrada e constrói a árvore pelo topo (raiz) para o fundo (folhas) e a análise preditiva não recursiva que utiliza uma pilha para consultar se a frase está sintaticamente correta e constrói a árvore pelas folhas até a raiz [1].

5.1.1 Análise Sintática de Descendência Recursiva

A análise sintática de descendência recursiva nada mais é do que a tentativa de encontrar uma derivação mais à esquerda nas construções de uma árvore gramatical. Devido à recursividade à esquerda, poderá haver retrocesso na derivação da árvore. O analisador sintático poderá entrar em laço infinito [4].

5.1.2 Análise Sintática Preditiva

A análise sintática preditiva não necessita de um retrocesso, já que em cada gramática é eliminada sua recursividade à esquerda e fatorada [4].

Para se criar um analisador sintático preditivo, basta criar um estado inicial e final de forma que cada produção da gramática crie um percurso a partir do estado inicial até o final.

5.1.3 Análise Sintática Preditiva Não recursiva

O analisador, preditivo não recursivo, utiliza um sistema de empilhamento, ao invés de chamadas recursivas.

Para se criar um analisador, preditivo não recursivo, será necessário possuir um buffer de entrada na pilha, uma tabela sintática e um fluxo de saída [2]. A pilha estará com o símbolo inicial, a partir disso, será informada a cadeia de entrada, se a derivação mais à esquerda do símbolo de saída, for à mesa da entrada, a pilha vai se desempilhando, até que fique vazia ou relate algum tipo de erro.

5.2 Análise Sintática Buttom – UP

A análise sintática *buttom-up* constrói uma árvore a partir das folhas e vasculha a árvore até chegar à raiz mais conhecida como empilhar e reduzir. Sempre que uma

subcadeia reconheça o lado direto de uma produção é desempilhado um símbolo à esquerda da produção [3].

São efetuadas duas ações sobre os *tokens* da entrada e uma pilha auxiliar retirando um token corrente e colocando-o na pilha e logo em seguida, substituindo os símbolos do topo da pilha por um não terminal [3], seguida uma regra gramatical.

6. Tradução Dirigida Pela Sintaxe

Assim que a arvore sintática estiver construída, a partir do *parser*, será feita a tradução do código alvo em cada derivação que ocorra na árvore, pode-se executar outras ações, simultaneamente, como interpretar o código, informar a presença de erros, guardarem informações na tabela de símbolos [3].

Em cada derivação de uma árvore, por exemplo, podemos associar outras informações na árvore, sendo assim, algumas informações, como tipo de atributos, ações semânticas entre outras, na quais, irão definir valores às sentenças [1].

Os atributos podem ser herdados a partir de pais ou irmãos ou os atributos podem ser sintetizados a partir de valores dos filhos.

7. Verificação de Tipos

Após o compilador ter feito todas as análises no código fonte e feito a tradução dirigida pela sintaxe, um verificador de tipos checa se o tipo de uma construção corresponde àquele esperado no contexto, por exemplo, se efetuarmos duas operações que trabalham com tipos inteiros, qualquer outra associação feita nessa expressão, deverá equivaler ao tipo [1]. A estrutura do verificador de tipos está ligada à análise sintática, aos tipos de dados e as regras para a construção de linguagens.

8. Geração de código intermediário

O código intermediário trata-se do resultado das fases de análises, ou seja, é uma máquina abstrata que é fácil de produzir e de transformar no programa alvo [2]. É semelhante ao código de montagem, só que, na montagem os registradores seriam os endereços de memória e com somente um operador por instrução.

A Geração de código intermediário é a transformação da árvore de derivação em um segmento de código que não tem semelhança da linguagem alvo, pois a árvore de derivação é uma representação do código fonte [4].

O código intermediário torna o processo de compilação mais lento, já que ele é uma fase a mais, na compilação, mas esse código pode ser posteriormente "otimizado", para obter um código final mais eficiente [2].

Possibilitar a tradução do código por diversas máquinas é umas das características que irá facilitar na divisão do compilador em vanguarda e retaguarda.

9. Reconstrução das interfaces

Dividir o compilador em fases de vanguarda e retaguarda tem por importante objetivo a transportabilidade, pois existem operações que dependem diretamente da linguagem alvo ou da linguagem fonte [4].

A Vanguarda do compilador enquadra as fases de análise léxica, sintática e semântica e a geração do código intermediário. A Retaguarda do compilador depende da linguagem alvo, como o processo de síntese do código intermediário, assim o código intermediário serve como ligação de comunicação entre essas duas fases.

Existem dificuldades que limitam a divisão entre esses dois processos, tal como mudanças nas estruturas das linguagens de programações e também nas arquiteturas de hardwares [3].

A interface de vanguarda está associada à interface de retaguarda, como a retaguarda recebe o código intermediário para fazer a síntese, pode-se então criar diferentes interfaces de vanguarda para uma interface de retaguarda.

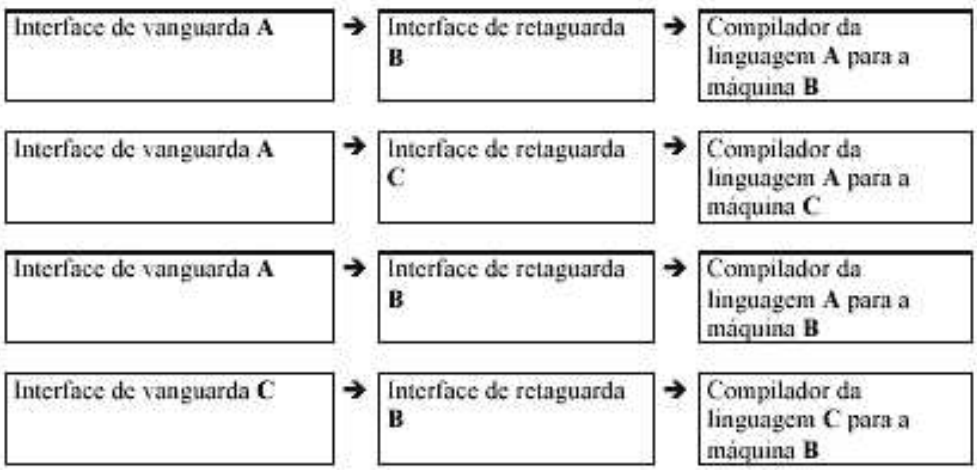

Figura 4. Divisão das interfaces [3]

10. Interface de Vanguarda para linguagem C e Pascal

Como foi definida anteriormente, a vanguarda de um compilador enquadra as fases de análise, até a geração de código intermediário, a partir da linguagem fonte.

Para cada linguagem de programação, temos um compilador que faz a leitura do código fonte e o transforma em linguagem de máquina, mas para que isso aconteça, será necessário que o código fonte atende as necessidades sintáticas da linguagem.

As linguagens de programação Pascal e C, utilizam compiladores distintos, já que, cada uma possui sintaxes e especificações diferentes. Mesmo havendo diferenças significantes entre essas duas linguagens, podemos gerar um código intermediário equivalente, entre elas.

A fase de análise para um compilador Pascal seria basicamente a leitura do código fonte, seguida da análise léxica, tratamento de erros, fluxo de *tokens*, um tradutor preditvo e verificador de tipo. Enquanto na linguagem C, são utilizados os processos de análise léxica e sintática seguida da geração de código intermediário. A interface de vanguarda das linguagens C e Pascal são distintas, mas a partir da geração de código intermediário, podemos utilizar à mesma retaguarda do compilador.

Abaixo, temos exemplos simplificados de transformações de uma parte de código em linguagem Pascal para seu código intermediário, que atribui à variável “C”, o resultado da multiplicação de um numeral com a soma de dois identificadores “A” e “B”, dividido pela soma de dois números inteiros.

```
c := ( a + b ) * 2 / ( 5 + 3) ;
```

A partir do exemplo acima, será gerado o código intermediário do programa em linguagem Pascal.

```
temp1 := a + b ;
temp2 := temp1 * 2 ;
temp3 := 6 + 7 ;
temp4 := temp2 / temp3 ;
c := temp4 ;
```

Logo, temos um outro exemplo de geração de código intermediário, mas pela linguagem C que atribui à variável “C”, o resultado da multiplicação de um numeral com a soma de dois identificadores “A” e “B”, dividido pela soma de dois números inteiros.

```
c = ( a + b ) * 2 / ( 5 + 3) ;
```

Em seguida é transformada a linguagem acima, para o código intermediário, seguindo as regras de análise, definida pela linguagem.

```
temp1 = a + b ;
temp2 = temp1 * 2 ;
temp3 = 6 + 7 ;
temp4 = temp2 / temp3 ;
c = temp4 ;
```

Para dar continuidade na comparação da equivalência entre códigos intermediários, temos um outro exemplo de código em linguagem Pascal onde se tem operação aritmética entre identificadores e numerais.

```
x :=  a +  ( b * c) - 4 ;
temp1 := b * c ;
temp2 := temp1 – 4 ;
temp3 := temp2 + a;
x := c ;
```

Segue o mesmo exemplo acima, só na forma de código intermediário para a linguagem em C.

```
x =  a +  ( b * c) - 4 ;
temp1 = b * c ;
temp2 = temp1 – 4 ;
temp3 = temp2 + a;
x = c ;
```

No próximo exemplo em Pascal será subtraído de uma variável “A” o resto da divisão entre dois identificadores, logo em seguida multiplicado por um numeral, a fim

de identificar a geração de código intermediário, já que os comandos entre as linguagens de programação Pascal e C são desatentos.

```
x := a - (b mod c) * 36;
```

Feita a convecção para o código intermediário, temos a linha de código abaixo, representando a expressão anterior.

```
temp1 := b mod c ;
temp2 := temp 1 * 36 ;
temp3 :=  a – temp2 ;
x := temp3 ;
```

Para a linguagem C, seguiremos o exemplo anterior, alterando somente os *tokens* para seus equivalentes.

```
x = a - (b % c) * 36 ;
```

Assim temos a representação intermediária do código fonte da linguagem C, de acordo com o especificado pela linguagem.

```
temp1 := b % c ;
temp2 := temp 1 * 36 ;
temp3 :=  a – temp2 ;
x := temp3 ;
```

Existem algumas limitações para gerar o mesmo código intermediário para linguagens diferentes, a linguagem C, por exemplo, trabalha com um pré-processador, que faz concatenação das bibliotecas, formando um único código fonte, mas utilizando os comandos básicos de cada linguagem, como atribuição, comparação, repetição etc., podemos então, trabalhar com uma retaguarda comum entre linguagens distintas, fazendo somente alterações nas fases de análise.

11. Conclusão

Através dos processos utilizados para se gerar um código intermediário para a linguagem Pascal e C, ficou definida a característica de transportabilidade do compilador, já que os códigos intermediários gerados foram equivalentes. Assim a retaguarda do compilador irá sintetizá-los de uma forma idêntica.

Pode-se criar diferentes interfaces de retaguarda para várias interfaces de vanguardas. Assim todos os processos que estão ligados às fases de análise, por exemplo, podem entender linguagens de programações diferentes. Portanto para criar linguagens de programações diferentes bastaria somente utilizar os processos da vanguarda do compilador.

Pelo fato de um compilador ser representado em duas fases, retaguarda e vanguarda, existe a possibilidade de se redirecionar parte do compilador para uma outra máquina. Conhecer os processos de análise de compilação auxilia não só no entendimento da ferramenta, mas, serve também, como base para criação de novas linguagens, devido à compreensão e o modo de tratar as fases de um compilador serem equivalentes.

Para desenvolver uma nova linguagem de programação será sempre necessário utilizar uma estrutura que faça a conversão de uma linguagem de alto nível para uma linguagem de máquina. Como processos de revisão bibliográfica foram prestadas as fases da vanguarda de um compilador, de modo a amenizar a complexidade de projeto de compiladores e demonstrar como e porque a finalidade de se conhecer os princípios dos compiladores.

12. Referências Bibliográficas

[1] AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D.; Compiladores Princípios, Técnicas e Ferramentas, Rio de Janeiro: Livro Técnico e Científico.

[2] PRICE, Ana Maria de Alencar; TOSCANI, Simão Sirineo. Implementação de Linguagens de Programação: Compiladores, Rio Grande do Sul: Sagra Luzzato, 2001.

[3] LOVISI, Elio Filho; Material de Apoio Pedagógico da Disciplina Compiladores, Juiz de Fora, 2004.

[4] LOUDEN, Kenneth C. Louden; Compiladores Princípios e Práticas, São Paulo: Thomson, 2004.

[5] DIVERIO, Tiarajú Asmuz; MENEZES, Paulo Blath; Teoria da Computação Máquinas Universais e Computabilidade, Rio Grande do Sul: Sagra Luzzato, 2000.