Compiladores

Capítulo 1: Introdução

1. A estrutura global do compilador

Este capítulo pretende apresentar a estrutura geral de um compilador, sem entretanto entrar em detalhes que só podem ser apresentados após uma discussão mais longa, após a apresentação de importantes conceitos e resultados adicionais. Vamos apenas descrever aqui as partes componentes principais de um compilador "típico", e o seu funcionamento simplificado.

O nome *compilador*, criado nos anos 50, faz referência ao processo de composição de um programa pela reunião de várias rotinas de biblioteca; o processo de tradução (de uma linguagem fonte para uma linguagem objeto), considerado hoje a função central de um compilador, era então conhecido como *programação automática*.

Nesse processo de tradução, há duas tarefas básicas a serem executadas por um compilador:

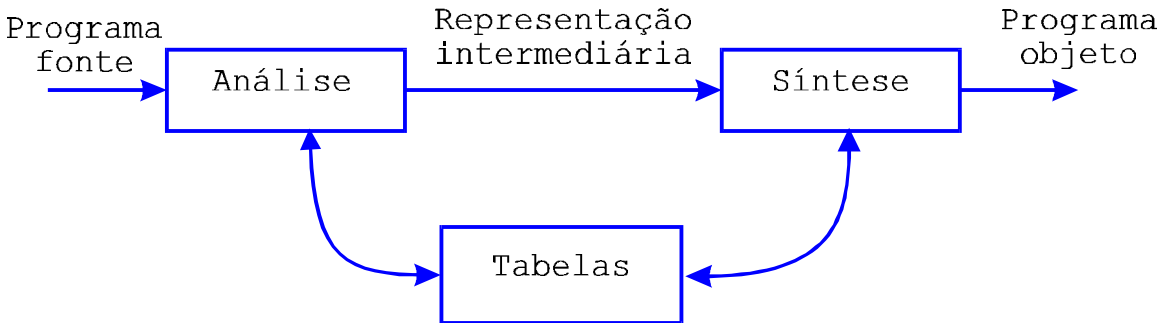
- *análise*, em que o texto de entrada (na linguagem fonte) é examinado, verificado e compreendido
- *síntese*, ou *geração de código*, em que o texto de saída (na linguagem objeto) é gerado, de forma a corresponder ao texto de entrada.

Normalmente, pensamos nessas tarefas como fases do processo de compilação, mas não é absolutamente necessário que a análise de todo o programa seja completada antes que o primeiro trecho de código objeto seja gerado: essas duas fases podem ser intercaladas. Como exemplos, um compilador pode analisar cada comando do programa de entrada, e gerar imediatamente o código de saída correspondente a esse comando; alternativamente, o compilador pode esperar o fim da análise de cada unidade de programa (rotina, procedimento, função, ...) para então gerar o código correspondente à unidade. Para melhor aproveitamento de memória durante sua execução, compiladores mais antigos costumavam ser divididos em vários *passos*, executados em seqüência, freqüentemente de forma aparente para o usuário. Cada passo executa parte do processo de tradução, transformando o código fonte em alguma forma intermediária adequada, cada vez mais próxima do código objeto final.

Naturalmente, a tarefa de análise deve ter como resultado uma representação do programa fonte que contenha informação suficiente para a geração do programa objeto correspondente. Normalmente, essa representação (conhecida como *representação intermediária*) é complementada por tabelas que contêm informação adicional sobre o programa fonte. Em alguns casos, a representação intermediária pode tomar a forma de um programa em uma *linguagem intermediária*, a partir da qual seja fácil a tradução para a linguagem objeto desejada.

Independentemente da forma que possa tomar a representação intermediária, ela deve conter toda a informação necessária para a geração do código objeto. Uma das características da representação intermediária é que as estruturas de dados empregadas devem garantir acesso eficiente a todas as informações, podendo, para isso, ser conveniente algum grau de redundância. Normalmente esse acesso é consideravelmente

mais rápido que o acesso ao programa fonte em formato de texto, ou a código executável em alguma máquina real.



Uma das formas mais comuns de tabela utilizada nessa representação intermediária é a *tabela de símbolos*, em que se guarda para cada identificador (*símbolo*) usado no programa as informações correspondentes, tais como natureza (variável, constante, procedimento, ...), tipo, endereço, espaço ocupado, etc.

Um dos modelos possíveis para a construção de compiladores faz a separação total entre o *front-end*, encarregado da fase de análise, e o *back-end*, encarregado da geração de código, de forma que

- front-end e back-end se comunicam apenas através da representação intermediária;
- o front-end depende exclusivamente da linguagem fonte (e, portanto, independe da linguagem ou da máquina objeto);
- o back-end depende exclusivamente da linguagem objeto (e, portanto, independe da linguagem fonte).

Essa idéia visa simplificar a implementação de várias linguagens de programação para várias máquinas: em princípio, basta escrever um front-end para cada linguagem, e um back-end para cada máquina. Ou seja, para implementar *m* linguagens em *n* máquinas, precisamos fazer *m* front-ends e *n* back-ends, em vez de *mn* compiladores completos. Este esquema é mais facilmente aplicado quando existe alguma semelhança entre as máquinas e o mesmo acontece entre as linguagens.

Os primeiros compiladores (da linguagem FORTRAN) foram construídos quase quarenta anos atrás (1957), quando técnicas de projeto e implementação de linguagens ainda não estavam muito desenvolvidas. Uma boa idéia da tecnologia disponível para a implementação dos primeiros compiladores pode ser obtida em [Rosen[1]], onde estão reunidos alguns dos papers mais importantes dos anos 50.

Quase todos os compiladores fazem hoje uso de uma técnica chamada *tradução dirigida pela sintaxe*, em que as regras de construção do programa fonte são utilizadas diretamente para guiar todo o processo de compilação. Assim, por exemplo, a regra da gramática da linguagem fonte que contém o sinal de + deve ser a que deflagra o processo de geração da instrução ADD.

2. Análise

[1]S. Rosen, **Programming Systems and Languages**, McGraw-Hill, 1967.

Normalmente associamos a *sintaxe* a idéia de forma, em oposição a *semântica*, associada a significado, conteúdo. Assim, em princípio, a sintaxe de uma linguagem de programação deve descrever todos os aspectos relativos à forma de construção de programas corretos na linguagem, enquanto a semântica deve descrever o que acontece quando o programa é executado. Portanto, toda a análise está relacionada com sintaxe, e a semântica deveria corresponder apenas à geração de código, que deve preservar o significado do programa fonte, construindo um programa objeto com o mesmo significado.

Uma observação cabe aqui. Do ponto de vista da teoria, apenas programas corretos pertencem à linguagem, e os programas incorretos não tem nenhum interesse. Um programa ou é da linguagem (está correto) ou não é da linguagem (está incorreto). Do ponto de vista da prática, entretanto, no momento em que avisa que um programa está incorreto, um bom compilador deve ser capaz de indicar como esse fato foi descoberto, e, de alguma forma, ajudar o usuário a transformá-lo em um programa correto. O tratamento de erros deve incluir mensagens informativas e um reparo (ou recuperação) do programa incorreto, para que a análise possa ser continuada e outros erros sinalizados.

Por razões de conveniência prática, a fase de análise normalmente se subdivide em análise léxica, análise sintática e análise semântica. Sabe-se que é possível representar completamente a sintaxe de uma linguagem de programação através de uma gramática sensível ao contexto[2], mas como não existem algoritmos práticos para tratar essas gramáticas, a preferência recai em usar gramáticas livres de contexto. Assim, a separação entre análise sintática e análise semântica é dependente de implementação: deixa-se para a análise semântica a verificação de todos os aspectos da linguagens que não se consegue exprimir de forma simples usando gramáticas livres de contexto.

Por outro lado, sabe-se que a implementação de reconhecedores de linguagens regulares (autômatos finitos) é mais simples e mais eficiente do que a implementação de reconhecedores de linguagens livres de contexto (autômatos de pilha). Como é possível usar expressões regulares para descrever a estrutura de componentes básicos das linguagens de programação, tais como identificadores, palavras reservadas, literais numéricos, operadores e delimitadores, essa parte da tarefa de análise (análise léxica) é implementada separadamente, pela simulação de autômatos finitos.

Assim, a análise léxica tem como finalidade a separação e identificação dos elementos componentes do programa fonte; normalmente esses componentes são especificados através de expressões regulares. A análise sintática deve reconhecer a estrutura global do programa, descrita através de gramáticas livres de contexto. A análise semântica se encarrega da verificação das regras restantes. Essas regras tratam quase sempre da verificação de que os objetos são usados no programa da maneira prevista em suas declarações, por exemplo verificando que não há erros de tipos.

---

[2]Podem ser usadas também gramáticas de dois níveis, como as gramáticas de van Wijngaarden, que foram usadas para descrever a sintaxe completa da linguagem Algol 68. Ver: A. van Wijngaarden (Editor) **Report on the Algorithmic Language Algol-68**, Numerische Mathematik, 14, (1969) pp 79-218

Não existe um modelo matemático inteiramente adequado para descrever o que deve ser verificado na análise semântica, ao contrário do que acontece nas duas outras fases da análise, mas mecanismos como gramáticas de atributos tem sido usados com sucesso para simplificar a construção de analisadores semânticos.

Existem diversas ferramentas de software que auxiliam a construção de compiladores, construindo módulos do compilador a partir de especificações. Considera-se normal, hoje em dia, o uso dessas ferramentas, sendo as mais comuns os construtores de analisadores léxicos e sintáticos, caso em que os módulos gerados automaticamente são tanto ou mais eficazes que os construídos a mão.

Já é possível especificar totalmente a semântica de uma linguagem de programação, e construir automaticamente um compilador para a linguagem a partir dessa especificação. Entretanto, ainda não é possível a construção totalmente automática de compiladores de qualidade comercial (*production quality*), razão pela qual partes dos compiladores são ainda escritas à mão. Em geral, um compilador passa por um processo de ajuste (sintonização, *fine tuning*), em que se determina quais as partes do compilador que consomem mais recursos (tempo de execução, espaço em memória), e então se procura re-escrever estas partes de forma melhorada.

3. Análise léxica

Como vimos, cabe à análise léxica a separação e identificação dos elementos componentes do programa fonte; à análise léxica cabe também a eliminação dos elementos "decorativos" do programa, tais como espaços em branco, marcas de formatação de texto e comentários. Isso significa que, após a análise léxica, itens como identificadores, operadores, delimitadores, palavras chave, palavras reservadas estão identificados, normalmente através de duas informações: um código numérico e uma cadeia de símbolos (o trecho do programa fonte correspondente). Estes itens são conhecidos como componentes léxicos do programa, lexemas, ou ainda *tokens*.

Por exemplo, considere o trecho de programa Pascal:

```
if x>0 then          { x e' positivo }
     modx := x
else                 { x e' negativo ou nulo }
     modx := (-x)
```

Após a análise léxica, a seqüência de tokens identificada é:

tipo do token	valor do token
palavra reservada if	`if`
identificador	`x`
operador maior	`>`
literal numérico	`0`
palavra reservada then	`then`
identificador	`modx`
operador de atribuição	`:=`
identificador	`x`
palavra reservada else	`else`
identificador	`modx`

tipo do token	valor do token
operador de atribuição	`:=`
delimitador abre parêntese	`(`
operador menos unário	`-`
identificador	`x`
delimitador fecha parêntese	`)`

Normalmente os tipos dos tokens (na primeira coluna) são representados por valores de um tipo de enumeração ou por códigos numéricos apropriados.

Usualmente, a implementação de um analisador léxico (um *scanner*) se baseia em um autômato finito que reconhece as diversas construções. Por exemplo, o conjunto dos identificadores permitidos em Pascal pode ser descrito pela expressão regular

`letra` $\circ$ (`letra` $\vee$ `digito` $\vee$ `sublinhado`)*,

onde os operadores $\circ$, $\vee$, e * representam respectivamente concatenação, união e repetição, (zero ou mais vezes) e `letra`, `digito` e `sublinhado` representam conjuntos (ou classes) de caracteres:

`letra` = { 'A', ..., 'Z', 'a', ... 'z' }
`digito` = { '0', ..., '9' }
`sublinhado` = { '_' }.

A parte correspondente do autômato finito pode ser construída a partir dessa expressão regular.

A análise léxica pode ser mais complicada em outras linguagens. Por exemplo, FORTRAN não tem palavras reservadas, tem apenas palavras-chave, que também podem ser usadas como identificadores; além disso, FORTRAN permite também o uso de espaços dentro de identificadores e palavras-chave. O exemplo clássico dos problemas que podem ser encontrados é a cadeia

```
DO 10 I = 5
```

que pode ser a parte inicial de um comando de atribuição

```
DO 10 I = 5.
```

em que a variável real `DO10I` recebe o valor real `5.`, ou pode ser o começo do comando de repetição,

```
      DO 10 I = 5, 20
      ...
10    CONTINUE
```

que especifica que os comandos entre o comando `DO` e o comando rotulado por `10` devem ser executados uma vez para cada valor de `I` = 5, 6, ..., 20. Note que até que o ponto ou a vírgula sejam encontrados, não é possível decidir qual é a interpretação correta para `DO10I`.

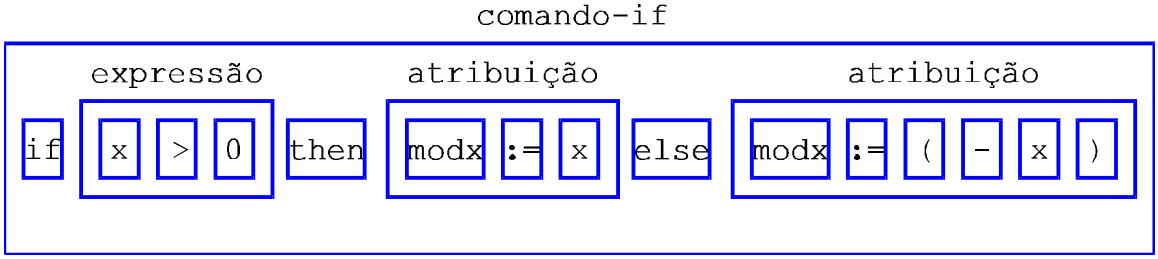
O projeto da maioria das linguagens modernas procura simplificar a implementação dos analisadores léxicos, que são, em geral, bem mais simples e mais rápidos que os de FORTRAN. Mesmo assim, uma parte substancial do tempo de compilação é gasta em análise léxica.

4. Análise sintática

A análise sintática deve reconhecer a estrutura global do programa, por exemplo, verificando que programas, comandos, declarações, expressões, etc têm as regras de composição respeitadas. Caberia à análise sintática reconhecer a estrutura do trecho

```
if x>0 then modx := x else modx := (-x)
```

identificando que se trata de um <comando>, no caso um <comando-if>, composto pela palavra reservada if, seguida de uma <expressão>, seguida da palavra reservada then, etc. Os itens <expressão> e <atribuição> ainda podem ser decompostos em fragmentos menores.



Quase universalmente, a sintaxe das linguagens de programação é descrita por gramáticas livres de contexto, em uma notação chamada BNF (*Forma de Backus-Naur* ou ainda *Forma Normal de Backus*), ou em alguma variante ou extensão dessa notação. Essa notação foi introduzida por volta de 1960, para a descrição da linguagem Algol[3]. Entre as regras para a sintaxe de comando em Pascal podemos ter as regras abaixo, algumas das quais seriam usadas na análise do trecho de código Pascal apresentado acima. A notação com os *parênteses bicudos* "<" e ">" é freqüentemente usada para indicar que <comando>, <comando-if>, <atribuição>, etc. são símbolos não terminais de uma gramática livre de contexto.

```
<comando> ::=
        <comando-if>
      | <atribuição>
      | ...                          ← outras opções de comando

<comando-if> ::=
        IF <condição> THEN <comando>
      | IF <condição> THEN <comando> ELSE <comando>

<atribuição> ::=
        <variável> ::= <expressão>
```

Algumas extensões da notação BNF permitem a introdução de expressões regulares no lado direito das regras. Por exemplo, um comando if de Ada pode ser representado pela regra

```
<comando-if> ::=
      IF <expressão> THEN <comandos>
      { ELSIF <expressão> THEN <comandos> }
```

[3] Ver: Peter Naur (Editor), **Revised Report on the Algorithmic Language Algol-60,** Numerische Mathematik, 4, (1963) pp 420-453.

```
[ ELSE <comandos> ]
END IF ;
```

onde as chaves e os colchetes indicam que a parte `ELSIF` pode ser repetida zero ou mais vezes, e que a parte `ELSE` pode ocorrer zero ou uma vez. De forma semelhante, temos:

```
<comandos> ::= <comando> { <comando> }
```

Estas extensões abreviam as especificações de gramáticas de linguagens de programação, sem no entanto aumentar a classe de linguagens que podem ser representadas. Algumas ferramentas de construção de analisadores sintáticos aceitam estas e outras extensões à notação BNF.

A teoria de gramáticas e linguagens livres de contexto fornece algoritmos determinísticos que permitem determinar se uma cadeia pertence à linguagem de uma gramática livre de contexto qualquer em tempo polinomial ($O(n^3)$) no comprimento da cadeia. Entretanto, na prática, restrições nas classes de gramáticas utilizadas permitem garantir o funcionamento dos analisadores em tempo linear ($O(n)$). As classes de gramáticas mais utilizadas são LL(1), LR(1) e suas variantes.

5. Análise semântica

Fundamentalmente, a análise semântica trata os aspectos sensíveis ao contexto da sintaxe das linguagens de programação. Por exemplo, não é possível representar em uma gramática livre de contexto uma regra como *"Todo identificador deve ser declarado antes de ser usado."*, e a verificação de que essa regra foi aplicada cabe à análise semântica. A regra pode ser representada em uma gramática sensível ao contexto, mas não existem algoritmos rápidos para essa classe de gramáticas. A idéia fundamental é a de usar tabelas externas ao processo de análise sintática, em que as informações são coletadas para posterior consulta. Por exemplo, uma *tabela de símbolos* (ou tabela de identificadores) pode guardar informações sobre as declarações dos identificadores, e essas informações podem ser consultadas para verificar a correção de cada uso de um identificador.

Esse processo é implementado de forma *dirigida pela sintaxe*: associa-se a cada regra da gramática uma ação (*ação semântica*) a ser executada quando o analisador sintático sinaliza um uso da regra. Essas ações são freqüentemente implementadas como chamadas de *rotinas semânticas*, e podem ser responsáveis por efetuar a análise semântica e a geração de código, pelo menos parcialmente.

No exemplo acima, toda vez que o analisador sintático indicar o uso de uma regra associada a uma declaração, a rotina semântica associada a essa regra é chamada para acrescentar à tabela o identificador correspondente, fornecido pelo analisador léxico. Quando uma regra associada a um uso de um identificador for sinalizada, a rotina semântica correspondente será chamada para verificar se o identificador (novamente fornecido pelo analisador léxico) consta da tabela.

Não existe uma fronteira definida entre o que deve ser tratado pelo analisador sintático e o que deve ser tratado pelo analisador semântico, cabendo ao programador do compilador a escolha, segundo suas preferências. Por exemplo, algumas versões de Pascal exigem que as declarações em um bloco ocorram numa ordem determinada,

começando pelas declarações de rótulos e terminando pelas de procedimentos e funções. Isto pode ser feito (na sintaxe) por uma regra

```
<decs> ::= <dec1> <dec2> <dec3> <dec4> <dec5>,
```

ou (na semântica) através de um número inteiro que representa a fase em que o processo se encontra: na fase *i*, só podem ser aceitas declarações <dec*j*> para j≥i.

Durante o curso, encontraremos outras situações semelhantes, em que a escolha da forma de implementação será feita de acordo com as preferências do implementador.

6. Geração de Código e Otimização Dependente de Máquina

Observamos antes que uma representação intermediária do programa fonte deve ser construída durante a fase de análise, para ser usada como base para a geração do programa objeto. Se a forma dessa representação intermediária é bem escolhida, a complexidade do processo de geração de código depende apenas da arquitetura da máquina (real ou virtual) para a qual o código está sendo gerado. Máquinas mais simples oferecem poucas opções e por isso o processo de geração de código é mais direto.

Por exemplo, se uma máquina tem apenas um registrador (acumulador) em que as operações aritméticas são realizadas, e apenas uma instrução para realizar cada operação (uma instrução para soma, uma para produto, ...), existe pouca ou nenhuma possibilidade de variação no código que pode ser gerado. Considere o comando de atribuição

```
x := a + b * c
```

A primeira operação a ser realizada é o produto de `b` por `c`. Seu valor deve ser guardado numa posição temporária, que indicaremos aqui por `t1`. (Para sistematizar o processo, todos os resultados de operações aritméticas serão armazenados em posições temporárias.) Em seguida, devemos realizar a soma de `a` com `t1`, cujo valor será guardado numa posição temporária `t2`. (Naturalmente, neste caso particular, o valor poderia ser armazenado diretamente em `x`, mas no caso geral, a temporária é necessária.) Finalmente, o valor de `t2` é armazenado em `x`.

```
t1:=b*c
t2:=a+t1
x:=t2
```

Podemos fazer um gerador de código relativamente simples usando regras como:

1. toda operação aritmética (binária) gera 3 instruções:

instrução	exemplo: `b * c`
carrega o primeiro operando no acumulador	`Load b`
usa a instrução correspondente a operação com o segundo operando, deixando o resultado no acumulador	`Mult c`
armazena o resultado em uma temporária nova	`Store t1`

2. um comando de atribuição gera sempre duas instruções:

instrução	exemplo: `x := t2`
carrega o valor da expressão no acumulador	`Load t2`
armazena o resultado na variável	`Store x`

O comando de atribuição

`x := a + b * c`,

gera o código

```
1  Load b                    { t1:=b*c }
2  Mult c
3  Store t1
4  Load a                   { t2:=a+t1 }
5  Add t1
6  Store t2
7  Load t2                    { x:=t2 }
8  Store x
```

Embora correto, este código pode obviamente ser melhorado:

- a instrução *7* é desnecessária e pode ser retirada: copia para o acumulador o valor de `t2`, que já se encontra lá.
- (após a remoção da instrução *7*) a instrução *6* é desnecessária e pode ser retirada: o valor da variável `t2` nunca é utilizado.
- (considerando que a soma é comutativa) as instruções *4* e *5* podem ser trocadas por *4′* e *5′*, preparando novas alterações:

```
4′ Load t1
5′ Add a
```

- As instruções *3* e *4′* são desnecessárias e podem ser retiradas (pelas mesmas razões que *6* e *7* acima).

O código final após as transformações é consideravelmente melhor que o original:

```
1  Load b
2  Mult c
5′ Add a
8  Store x
```

Normalmente, as máquinas oferecem várias instruções (ou variantes de instruções) com características semelhantes, e o gerador deve escolher a mais apropriada

entre elas. Como exemplo, vamos examinar o caso da operação de soma. Em geral, podemos observar os seguintes pontos:

- há várias instruções de soma, correspondendo a vários tipos de dados e a vários modos de endereçamento;
- há instruções de soma aplicáveis a casos particulares importantes, como instruções de incremento e decremento: soma com ±1;
- algumas das somas a serem efetuadas não foram especificadas explicitamente pelo programador. Entre essas citamos as somas usadas no cálculo de endereços de variáveis componentes de vetores, matrizes e estruturas situadas em registros de ativação de procedimentos ou funções. Freqüentemente, essas somas podem ser incluídas no código de forma implícita através da escolha de modos de endereçamento adequados;
- instruções cuja finalidade principal não é a soma podem mesmo assim efetuar somas. Por exemplo, as instruções que manipulam a pilha de hardware, incrementam ou decrementam o registrador apontador do topo da pilha.

Esses pontos devem ser levados em consideração pelo gerador de código na seleção de instruções. Outro problema que também deve ser tratado é o da escolha do local para guarda dos valores das variáveis definidas pelo usuário e das variáveis temporárias introduzidas pelo compilador. Além da alocação de posições de memória a essas variáveis, é freqüente a disponibilidade de vários registradores de uso geral, que também podem ser usados com essa finalidade. A alocação de registradores, entretanto, não é independente da seleção de instruções, já que muitas instruções usam registradores ou combinações de registradores para operações específicas.

Cabe ao projetista do gerador de código decidir como implementar a geração de código de maneira a fazer bom uso dos recursos disponíveis na máquina. Cabe também ao projetista decidir se a geração do código deve ser feita com cuidado, gerando diretamente código de qualidade aceitável, ou se é preferível usar um esquema mais simples de geração de código, seguido por uma "otimização" do código depois de gerado. Esta otimização do código leva em consideração principalmente as características da máquina alvo, e por isso é normalmente chamada de *otimização dependente de máquina.*

7. Otimização independente de máquina

Algumas transformações feitas no código gerado por um compilador independem da máquina para o qual o código está sendo gerado. Normalmente estas transformações são feitas no código intermediário, pela facilidade de acesso já mencionada anteriormente. Vamos nesta seção apresentar alguns exemplos deste tipo de otimização.

Exemplo 1: *Sub-expressões comuns.* Considere a seqüência de comandos de atribuição da primeira coluna da tabela.

Fonte	código intermediário original	código intermediário otimizado

`w:=(a+b)+c;`	`t1:=a+b`
	`t2:=t1+c`
	`w:=t2`
		`t1:=a+b`
		`t2:=t1+c`
		`w:=t2`

`x:=(a+b)*d;`	`t3:=a+b`
	`t4:=t3*d`
	`x:=t4`
		
		`t4:=t1*d`
		`x:=t4`

`y:=(a+b)+c;`	`t5:=a+b`
	`t6:=t5+c`
	`y:=t6`
		
		`y:=t2`

`z:=(a+b)*d+e;`	`t7:=a+b`
	`t8:=t7*d`
	`t9:=t8+e`
	`z:=t9`
		
		`t9:=t4+e`
		`z:=t9`

Claramente, as (sub-)expressões `a+b`, `(a+b)+c`, e `(a+b)*d` não precisam ser calculadas mais de uma vez. (Isto só é verdade porque os valores de `a`, `b`, `c` e `d` não se alteram no trecho em questão.) Podemos alterar a representação intermediária correspondente (segunda coluna) para a forma intermediária equivalente otimizada apresentada na terceira coluna.

Exemplo 2: Retirada de comandos *invariantes de loop*. Considere o trecho de código a seguir:

```
for i:=1 to n do begin
     pi:=3.1416;
     pi4:=pi/4.;
     d[i]:=pi4 * r[i] * r[i];
end;
```

Claramente, os dois primeiros comandos de atribuição podem ser retirados do *loop*, uma vez que seu funcionamento é independente do funcionamento do loop. Obteríamos

```
pi:=3.1416;
pi4:=pi/4.;
for i:=1 to n do
     d[i]:=pi4 * r[i] * r[i];
```

que é uma versão "otimizada" do trecho de código anterior. Note, entretanto, que só há uma melhora no tempo de execução se o valor de `n` for maior que zero. Se `n=0`, o código foi piorado: os dois comandos de atribuição serão sempre executados.

Normalmente, as transformações realizadas no programa durante a otimização são simples: eliminar ou alterar instruções, ou ainda mover instruções para outras posições. A parte mais trabalhosa é a verificar que a transformação pode ser feita. Por exemplo, para eliminar um comando da forma `v:=e`, é preciso verificar que o valor de `v` calculado neste comando não será usado por nenhum outro comando, e, portanto, examinar toda a parte do programa que poderá ser executada a seguir. Por essa razão, o principal assunto discutido no capítulo de otimização é a análise de fluxo de dados (*dataflow analysis*), que visa obter informação sobre o funcionamento do programa, em

particular especificando os pontos do programa onde as variáveis recebem valores, e onde os valores são usados.

8. Conclusão

Em 1977, Aho e Ullman indicaram, na capa de seu livro[4], as armas disponíveis contra o *Dragão* da Complexidade do Projeto de Compiladores: o gerador de analisadores sintáticos LALR, a tradução dirigida pela sintaxe, e a análise de fluxo de dados. O projeto de um compilador se inicia pela escolha de uma gramática para a linguagem fonte, e uma ferramenta existente se encarrega da construção do analisador sintático; as ações (rotinas semânticas) a serem executadas são associadas às regras dessa gramática; técnicas de análise de fluxo de dados fornecem a informação para boa geração/otimização de código.

Muitos sistemas tem sido propostos com a finalidade de construir um compilador a partir de especificações sintáticas e semânticas das linguagens fonte e objeto. Até hoje, entretanto, nenhum deles mereceu ser chamado de *compilador de compiladores*[5], e a construção de compiladores continua a ser feita à mão, ainda que auxiliada por ferramentas cada vez mais poderosas. Veremos alguma coisa sobre o uso de algumas dessas ferramentas nos capítulos seguintes.

(rev. mar 99)

[4]Alfred V. Aho, Jeffrey D. Ullman, **Principles of Compiler Design**, Addison-Wesley, 1977, ou
Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, **Compilers: Principles, Techniques and Tools**, Addison-Wesley, 1986

[5]O nome do conhecido gerador de analisadores sintáticos *yacc* é uma brincadeira sobre esse fato. Ver:
S. C. Johnson, Yacc - **Yet Another Compiler Compiler**, Computing Science Technical Report 32, AT&T Bell Laboratories, 1975