Capítulo 9

Geração de código e otimização

Os últimos capítulos apresentaram técnicas com o embasamento teórico e conceitual para permitir reconhecer os símbolos e as expressões tipicamente utilizadas em linguagens de programação de alto nível. No entanto, a operação de um compilador requer mais que o simples reconhecimento da validade de um programa; é preciso gerar o código equivalente que será efetivamente executado pelos processadores.

Além das análises léxica e sintática, as demais tarefas de um compilador são apoiadas por heurísticas e não mais por formalismos. O objetivo deste capítulo é oferecer uma visão geral dessas técnicas usadas para concluir a obtenção do código *assembly* equivalente ao programa de alto nível que foi processado pelo compilador.

9.1 Análise semântica

Embora a análise sintática consiga verificar se uma expressão obedece às regras de formação de uma dada gramática, seria muito difícil expressar através de gramáticas algumas regras usuais em linguagem de programação, como "todas as variáveis devem ser declaradas" e situações onde o contexto em que ocorre a expressão ou o tipo da variável deve ser verificado.

O objetivo da análise semântica é trabalhar nesse nível de inter-relacionamento entre partes distintas do programa. As tarefas básicas desempenhada durante a análise semântica incluem a verificação de tipos, a verificação do fluxo de controle e a verificação da unicidade da declaração de variáveis. Dependendo da linguagem de programação, outros tipos de verificações podem ser necessários.

Considere o seguinte exemplo de código em C:

```
int f1(int a, float b) {
  return a%b;
}
```

A tentativa de compilar esse código irá gerar um erro detectado pelo analisador semântico, mais especificamente pelas regras de **verificação de tipos**, indicando que o operador módulo `%` não pode ter um operador real. No compilador `gcc`, essa mensagem é

```
In function ‘f1’:
...: invalid operands to binary %
```

Em alguns casos, o compilador realiza a conversão automática de um tipo para outro que seja adequado à aplicação do operador. Por exemplo, na expressão em C

```
a = x - ’0’;
```

a constante do tipo caráter `'0'` é automaticamente convertida para inteiro para compor corretamente a expressão aritmética na qual ela toma parte; todo `char` em uma expressão é convertido pelo compilador para um `int`. Esse procedimento de conversão de tipo é denominado **coerção** *(cast)*. Em C, a seguinte seqüência de regras determina a realização automática de coerção em expressões aritméticas:

1. `char` e `short` são convertidos para `int`, `float` para `double`;
2. se um dos operandos é `double`, o outro é convertido para `double` e o resultado é `double`;
3. se um dos operandos é `long`, o outro é convertido para `long` e o resultado é `long`;
4. se um dos operandos é `unsigned`, o outro é convertido para `unsigned` e o resultado é `unsigned`;
5. senão, todos os operandos são `int` e o resultado é `int`.

Quando uma conversão imprevista ocorre, o compilador sinaliza para o programador através de uma mensagem de aviso. Por exemplo, com as declarações

```
int a;
int *p;
```

a expressão

```
a = p;
```

geraria a seguinte mensagem do compilador:

```
warning: assignment makes integer from pointer
         without a cast
```

Porém, se o programador indicar que sabe que está fazendo uma conversão “perigosa” através do operador de molde (ver Seção 6.3), então nenhuma mensagem é gerada.

Algumas linguagens de programação, como C++, permitem definir comportamentos diferenciados a operadores segundo o tipo de argumento que recebem. Por exemplo, na expressão

```
  c << x;
```

o operador `<<` será interpretado como o comando de deslocamento de bits à esquerda se `c` e `x` forem inteiros, mas será uma operação de saída se `c` for uma referência para um arquivo. Esse mecanismo de adequar o comportamento do operador segundo o tipo de seus operandos é denominado **sobrecarga de operadores**. Em geral, essas linguagens permitem também aplicar o mesmo tipo de mecanismo a rotinas. Através da **sobrecarga de funções**, o compilador seleciona entre rotinas que têm o mesmo nome aquela cuja quantidade e lista de tipos estão adequadas à forma de invocação.

Outro exemplo de erro detectado pela análise semântica, neste caso pela **verificação de fluxo de controle**, é ilustrado pelo código

```
void f2(int j, int k) {
  if (j == k)
    break;
  else
    continue;
}
```

Nesse caso, o compilador gera as mensagens:

```
In function ‘f2’:
...: break statement not within loop or switch
...: continue statement not within a loop
```

ou seja, ele reconhece que o comando `break` só pode ser usado para quebrar a seqüência de um comando de iteração *(within loop)* ou para indicar o fim de um `case` *(within switch)*. Da mesma forma, um comando `continue` só pode ser usado em um comando de iteração.

A **verificação de unicidade** detecta situações tais como duplicação em declarações de variáveis, de componentes de estruturas e em rótulos do programa. Por exemplo, a compilação do seguinte código

```
void f3(int k) {
  struct {
    int a;
    float a;
  } x;
  float x;
  switch (k) {
  case 0x31: x.a = k;
  case ’1’: x = x.a;
  }
}
```

causaria a geração das seguintes mensagens de erro pelo compilador `gcc`:

```
In function ‘f3’:
...: duplicate member ‘a’
...: previous declaration of ‘x’
...: duplicate case value
```

A primeira mensagem detecta que dois membros de uma mesma definição de estrutura recebem o mesmo nome, `a`, o que não é permitido. Na segunda mensagem a indicação refere-se às duas variáveis de mesmo nome, `x`. A terceira mensagem indica que dois `case`s em uma expressão `switch` receberam o mesmo rótulo, o que também não é permitido. Observe que, mesmo embora a forma de expressar o valor nos `case`s tenha sido diferente, o compilador verificou que `0x31` e `'1'` referiam-se ao mesmo valor e acusou a situação de erro.

9.2 Geração de código

A tradução do código de alto nível para o código do processador está associada à traduzir para a linguagem-alvo a representação da árvore gramatical obtida para as diversas expressões do programa. Embora tal atividade possa ser realizada para a árvore completa após a conclusão da análise sintática, em geral ela é efetivada através das ações semânticas associadas à aplicação das regras de reconhecimento do analisador sintático. Este procedimento é denominado **tradução dirigida pela sintaxe**.

Em geral, a geração de código não se dá diretamente para a linguagem *assembly* do processador-alvo. Por conveniência, o analisador sintático gera código para uma máquina abstrata, com uma linguagem próxima a *assembly* porém independente de processadores específicos. Em uma segunda etapa de geração de código, esse código intermediário é traduzido para a linguagem *assembly* desejada. Dessa forma, grande parte do compilador é reaproveitada para trabalhar com diferentes tipos de processadores.

9.2.1 Código intermediário

A linguagem utilizada para a geração de um código em formato intermediário entre a linguagem de alto nível e a linguagem *assembly* deve representar, de forma independente do processador para o qual o programa será gerado, todas as expressões do programa original. Duas formas usuais para esse tipo de representação são a notação posfixa e o código de três endereços.

9.2.2 Código de três endereços

O código de três endereços é composto por uma seqüência de instruções envolvendo operações binárias ou unárias e uma atribuição. O nome “três endereços” está associado à especificação, em uma instrução, de no máximo três variáveis: duas

para os operadores binários e uma para o resultado. Assim, expressões envolvendo diversas operações são decompostas nesse código em uma série de instruções, eventualmente com a utilização de variáveis temporárias introduzidas na tradução. Dessa forma, obtém-se um código mais próximo da estrutura da linguagem *assembly* (Seção 10.1) e, conseqüentemente, de mais fácil conversão para a linguagem-alvo.

Uma possível especificação de uma linguagem de três endereços envolve quatro tipos básicos de instruções: expressões com atribuição, desvios, invocação de rotinas e acesso indexado e indireto.

Instruções de atribuição são aquelas nas quais o resultado de uma operação é armazenado na variável especificada à esquerda do operador de atribuição, aqui denotado por :=. Há três formas para esse tipo de instrução. Na primeira, a variável recebe o resultado de uma operação binária:

```
x := y op z
```

O resultado pode ser também obtido a partir da aplicação de um operador unário:

```
x := op y
```

Na terceira forma, pode ocorrer uma simples cópia de valores de uma variável para outra:

```
x := y
```

Por exemplo, a expressão em C

```
a = b + c * d;
```

seria traduzida nesse formato para as instruções:

```
_t1 := c * d
  a := b + _t1
```

As **instruções de desvio** podem assumir duas formas básicas. Uma instrução de desvio incondicional tem o formato

```
goto L
```

onde `L` é um rótulo simbólico que identifica uma linha do código. A outra forma de desvio é o desvio condicional, com o formato

```
if x opr y goto L
```

onde `opr` é um operador relacional de comparação e `L` é o rótulo da linha que deve ser executada se o resultado da aplicação do operador relacional for verdadeiro; caso contrário, a linha seguinte é executada.

Por exemplo, a seguinte iteração em C

```
while (i++ <= k)
  x[i] = 0;
x[0] = 0;
```

poderia ser traduzida para

```
_L1: if i > k goto _L2
     i := i + 1
     x[i] := 0
     goto _L1
_L2: x[0] := 0
```

A **invocação de rotinas** ocorre em duas etapas. Inicialmente, os argumentos do procedimento são "registrados" com a instrução `param`; após a definição dos argumentos, a instrução `call` completa a invocação da rotina. A instrução `return` indica o fim de execução de uma rotina. Opcionalmente, esta instrução pode especificar um valor de retorno, que pode ser atribuído na linguagem intermediária a uma variável como resultado de `call`.

Por exemplo, considere a chamada de uma função `f` que recebe três argumentos e retorna um valor:

```
f(a, b, c);
```

Neste exemplo em C, esse valor de retorno não é utilizado. De qualquer modo, a expressão acima seria traduzida para

```
param a
param b
param c
_t1 := call f,3
```

onde o número após a vírgula indica o número de argumentos utilizados pelo procedimento `f`. Com o uso desse argumento adicional é possível expressar sem dificuldades as chamadas aninhadas de procedimentos.

O último tipo de instrução para códigos de três endereços refere-se aos modos de endereçamento indexado e indireto. Para atribuições indexadas, as duas formas básicas são

```
x := y[i]
x[i] := y
```

As atribuições associadas ao modo indireto permitem a manipulação de endereços e seus conteúdos. As instruções em formato intermediário também utilizam um formato próximo àquele da linguagem C:

```
x := &y
w := *x
*x := z
```

A representação interna das instruções em códigos de três endereços dá-se na forma de armazenamento em tabelas com quatro ou três colunas. Na abordagem que utiliza **quádruplas** (as tabelas com quatro colunas), cada instrução é representada por uma linha na tabela com a especificação do operador, do primeiro argumento, do segundo argumento e do resultado. Por exemplo, a tradução da expressão `a=b+c*d;` resultaria no seguinte trecho da tabela:

	operador	arg 1	arg 2	resultado
1	`*`	`c`	`d`	`_t1`
2	`+`	`b`	`_t1`	`a`

Para algumas instruções, como aquelas envolvendo operadores unários ou desvio incondicional, algumas das colunas estariam vazias.

Na outra forma de representação, por **triplas**, evita a necessidade de manter nomes de variáveis temporárias ao fazer referência às linhas da própria tabela no lugar dos argumentos. Nesse caso, apenas três colunas são necessárias, uma vez que o resultado está sempre implicitamente associado à linha da tabela. No mesmo exemplo apresentado para a representação interna por quádruplas, a representação por triplas seria

	operador	arg 1	arg 2
1	`*`	`c`	`d`
2	`+`	`b`	(1)

9.2.3 Notação posfixa

A notação tradicional para expressões aritméticas, que representa uma operação binária na forma `x+y`, ou seja, com o operador entre seus dois operandos, é conhecida como notação infixa. Uma notação alternativa para esse tipo de expressão é a notação posfixa, também conhecida como notação polonesa[1], na qual o operador é expresso após seus operandos.

O atrativo da notação posfixa é que ela dispensa o uso de parênteses. Por exemplo, as expressões

```
a*b+c;
a*(b+c);
(a+b)*c;
(a+b)*(c+d);
```

seriam representadas nesse tipo de notação respectivamente como

```
a b * c +
a b c + *
a b + c *
a b + c d + *
```

[1] O criador desse tipo de notação, J. Lukasiewicz, era polonês.

Instruções de desvio em código intermediário usando a notação posfixa assumem a forma

```
L jump
x y L jcc
```

para desvios incondicionais e condicionais, respectivamente. No caso de um desvio condicional, a condição a ser avaliada envolvendo `x` e `y` é expressa na parte `cc` da própria instrução. Assim, `j`*cc* pode ser uma instrução entre `jeq` (desvio ocorre se `x` e `y` forem iguais), `jne` (se diferentes), `jlt` (se `x` menor que `y`), `jle` (se `x` menor ou igual a `y`), `jgt` (se `x` maior que `y`) ou `jge` (se `x` maior ou igual a `y`).

Expressões em formato intermediário usando a notação posfixa podem ser eficientemente avaliadas em máquinas baseadas em pilhas, também conhecidas como máquinas de zero endereços. Nesse tipo de máquinas, operandos são explicitamente introduzidos e retirados do topo da pilha por instruções `push` e `pop`, respectivamente. Além disso, a aplicação de um operador retira do topo da pilha seus operandos e retorna ao topo da pilha o resultado de sua aplicação.

Por exemplo, a avaliação da expressão `a*(b+c)` em uma máquina baseada em pilha poderia ser traduzida para o código

```
push a
push b
push c
add
mult
```

9.3 Otimização de código

A etapa final na geração de código pelo compilador é a fase de otimização. Como o código gerado através da tradução orientada a sintaxe contempla expressões independentes, diversas situações contendo seqüências de código ineficiente podem ocorrer. O objeto da etapa de otimização de código é aplicar um conjunto de heurísticas para detectar tais seqüências e substituí-las por outras que removam as situações de ineficiência.

As técnicas de otimização que são usadas em compiladores devem, além de manter o significado do programa original, ser capazes de capturar a maior parte das possibilidades de melhoria do código dentro de limites razoáveis de esforço gasto para tal fim. Em geral, os compiladores usualmente permitem especificar qual o grau de esforço desejado no processo de otimização. Por exemplo, em `gcc` há opções na forma `-O...` que dão essa indicação, desde `-O0` (nenhuma otimização) até `-O3` (máxima otimização, aumentando o tempo de compilação), incluindo também uma opção `-Os`, indicando que o objetivo é reduzir a ocupação de espaço em memória.

Algumas heurísticas de otimização são sempre aplicadas pelos compiladores. Por exemplo, se a concatenação de código gerado por duas expressões no programa original gerou uma situação de desvio incondicional para a linha seguinte, como em

```
     <a>
     goto _L1
_L1: <b>
```

esse código pode ser seguramente reduzido com a aplicação da técnica de **eliminação de desvios desnecessários**, resultando em

```
     <a>
_L1: <b>
```

Outra estratégia de otimização elimina os rótulos não referenciados por outras instruções do programa. Assim, se o rótulo `_L1` estivesse sendo referenciado exclusivamente por essa instrução de desvio, ele poderia ser eliminado em uma próxima aplicação das estratégias de otimização.

As técnicas de otimização podem ser classificadas como independentes de máquina, quando podem ser aplicadas antes da geração do código na linguagem *assembly*, ou dependentes de máquina, quando aplicadas na geração do código *assembly*.

A otimização independente de máquina tem como requisito o levantamento dos blocos de comandos que compõem o programa. Essa etapa da otimização é conhecida como a análise de fluxo, que por sua vez contempla a análise de fluxo de controle e a análise de fluxo de dados. Estratégias que podem ser aplicadas analisando um único bloco de comandos são denominadas estratégias de otimização local, enquanto aquelas que envolvem a análise simultânea de dois ou mais blocos são denominadas estratégias de otimização global.

Algumas estratégias básicas de otimização, além da já apresentada eliminação de desvios desnecessários, são apresentadas a seguir.

A estratégia de **eliminação de código redundante** busca detectar situações onde a tradução de duas expressões gera instruções cuja execução repetida não tem efeito. Por exemplo, em

```
     x := y
       ...
     x := y
```

se não há nenhuma mudança no valor de `y` entre as duas instruções, então a segunda instrução poderia ser eliminada. O mesmo aconteceria se a segunda instrução fosse

```
     y := x
```

e o valor de `x` não fosse alterado entre as duas instruções.

Outra estratégia básica é a **eliminação de código não-alcançável**, ou "código morto". Por exemplo, se a seguinte seqüência de código fosse gerada por um conjunto de expressões,

```
       ...
     goto _L1
     x := y
_L1:   ...
```

a instrução contendo a atribuição de `y` a `x` nunca poderia ser executada, pois é precedida de um desvio incondicional e não é o destino de nenhum desvio, pois não contém um rótulo na linha. Assim, essa linha poderia ser eliminada e provocar posteriormente a aplicação da estratégia de eliminação de desvios desnecessários.

O **uso de propriedades algébricas** é outra estratégia de otimização usualmente aplicada. Nesse caso, quando o compilador identifica que uma expressão aritmética foi reduzida a $x + 0$ ou $0 + x$ ou $x - 0$ ou $x \times 1$ ou $1 \times x$ ou $x/1$, então ele substitui a expressão simplesmente por x. Outra classe de propriedades algébricas que é utilizada tem por objetivo substituir operações de alto custo de execução por operações mais simples, como

$$\begin{aligned} 2.0 \times x &= x + x \\ x^2 &= x \times x \\ x/2 &= 0.5 \times x \end{aligned}$$

Particularmente no último caso, se `x` for uma variável inteira a divisão por dois pode ser substituída por um deslocamento da representação binária à direita de um bit. Genericamente, a divisão inteira por 2^n equivale ao deslocamento à direita de n bits na representação binária do dividendo. Da mesma forma, a multiplicação inteira por potências de dois pode ser substituída por deslocamento de bits à esquerda.

A utilização de propriedades algébricas permite também o reuso de subexpressões já computadas. Por exemplo, a tradução direta das expressões

```
a = b + c;
e = c + d + b;
```

geraria o seguinte código intermediário:

```
  a := b + c
_t1 := c + d
  e := _t1 + b
```

No entanto, o uso da comutatividade e associatividade da adição permite que o código gerado seja reduzido usando a **eliminação de expressões comuns**, resultando em

```
  a := b + c
  e := a + d
```

Diversas oportunidades de otimização estão associadas à análise de comandos iterativos. Uma estratégia é a **movimentação de código**, aplicado quando um cálculo realizado dentro do laço na verdade envolve valores invariantes na iteração. Por exemplo, o comando

```
while (i < 10*j) {
  a[i] = i + 2*j;
  ++i;
}
```

estaria gerando um código intermediário equivalente a

```
_L1:  _t1 := 10 * j
      if i >= _t1 goto _L2
      _t2 := 2 * j
      a[i] := i + _t2
      i := i + 1
      goto _L1
_L2:   ...
```

No entanto, a análise de fluxo de dados mostra que o valor de `j` não varia entre iterações. Assim, o compilador poderia mover as expressões que envolvem exclusivamente constantes na iteração para fora do laço, substituindo esse código por

```
      _t1 := 10 * j
      _t2 := 2 * j
_L1:  if i >= _t1 goto _L2
      a[i] := i + _t2
      i := i + 1
      goto _L1
_L2:   ...
```

Um exemplo de otimização dependente de máquina pode ser apresentado para a expressão

```
x := y + K
```

onde `K` é alguma constante inteira. Na tradução para o código *assembly* de um processador da família 68K, a forma genérica poderia ser algo como

```
MOVE.L  y,D0
ADDI.L #K,D0
MOVE.L D0,x
```

No entanto, se o compilador verificasse que o valor de `K` fosse uma constante entre 1 e 8, então a segunda instrução *assembly* poderia ser substituída por `ADDQ.L`, que utilizaria em sua codificação apenas dois bytes ao invés dos seis bytes que seriam requeridos pela instrução `ADDI.L`.