Compiladores

Geração de Código Intermediário & Otimização

Introdução

As notas de aula anteriores abordaram o funcionamento do compilador e as ações envolvidas na etapa de análise. Nas Notas de Aula sobre Análise Semântica, mostrou-se que a saída da análise semântica é a árvore sintática tipada, usada como entrada pela geração de código intermediário. Alguns autores tratam a geração de código intermediário (e a otimização) como a primeira etapa da síntese, como mostra a Figura 1. Outros autores tratam geração e otimização como uma etapa que antecede a síntese. Estas notas de aula versam sobre a geração de código intermediário, sua motivação, suas vantagens e algumas de suas formas, e otimização de código, onde são citadas algumas ações de otimização que o compilador pode realizar. O texto finaliza com uma breve explanação sobre o custo do código objeto. Além disso, alguns exercícios foram propostos ao final do material para fixação do conteúdo.

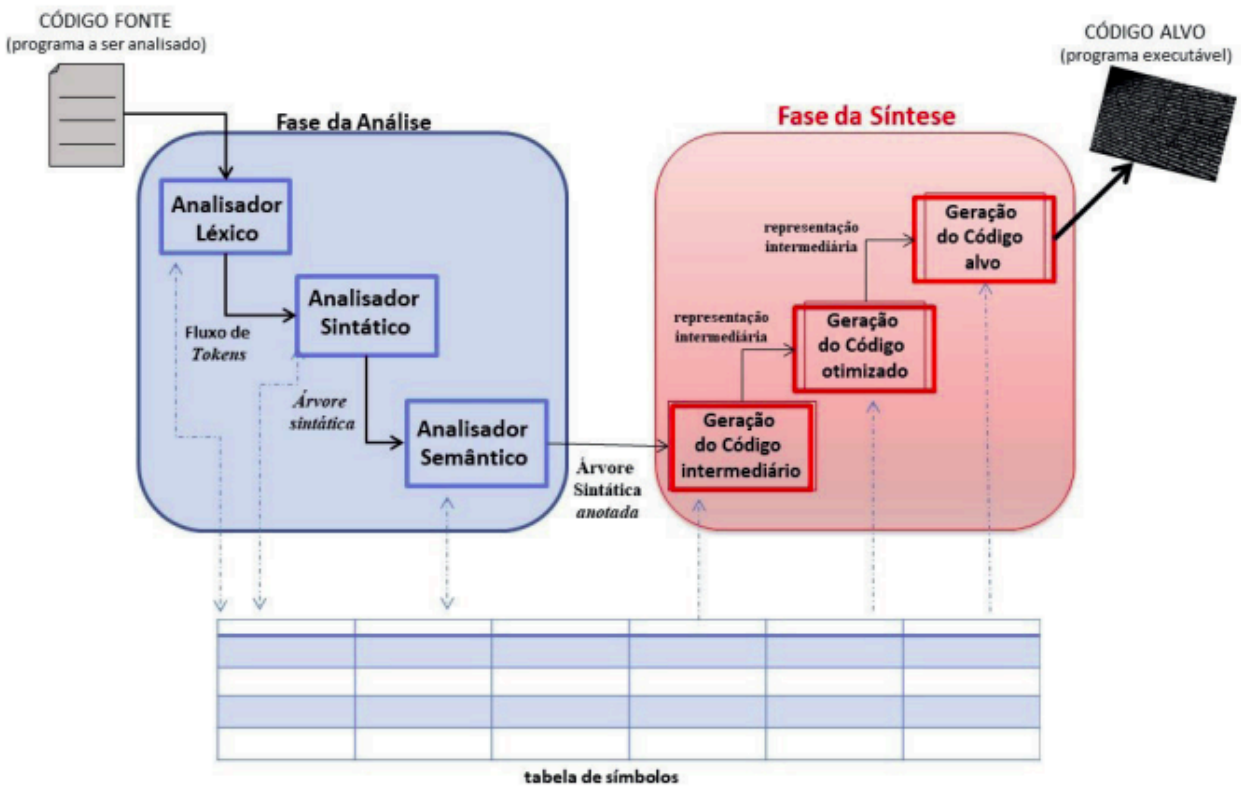


Figura 1. A geração do código intermediário no processo de compilação. Fonte: Fedozzi, 2018.

Geração de Código Intermediário & Otimização

Após a análise, o compilador gera o mapeamento do código fonte em uma representação intermediária. A geração do código é implementada em compiladores modernos pelo método de ***tradução dirigida por sintaxe***. Assim, realiza-se esta ação durante a análise sintática e conclui-se através das ações semânticas associadas à aplicação das regras de reconhecimento do *parser*.

Segundo Cardozo (s.d.), a geração de código não se dá diretamente para a linguagem *assembly* do processador-alvo, mas 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. Para facilitar a visualização, a Figura 2 mostra o papel da Representação

Intermediária – RI.

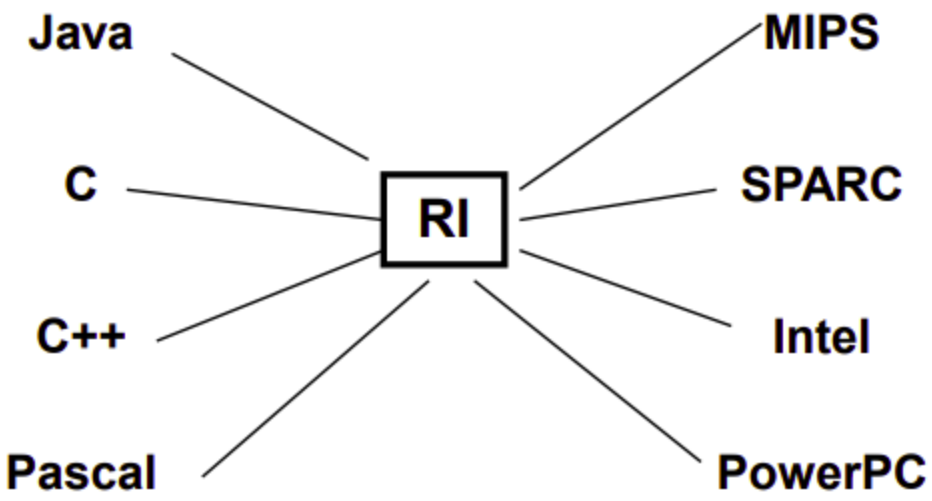


Figura 2. Representação intermediária e a relação entre linguagens e processadores. Fonte: Rigo, 2009.

No caso da Figura 2, para as linguagens Java, C, C++ e Pascal, e os processadores MIPS, SPARK, Intel e PowerPC, sem uma RI seriam necessários 16 compiladores:

- de Java para MIPS
- de Java para SPARC
- de Java para Intel
- de Java para PowerPC
- de C para MIPS
- de C para SPARC
- de C para Intel
- de C para PowerPC
- de C++ para MIPS
- de C++ para SPARC
- de C++ para Intel
- de C++ para PowerPC
- de Pascal para MIPS
- de Pascal para SPARC
- de Pascal para Intel
- de Pascal para PowerPC

Utilizando uma RI conhecida pelos *front-ends* e pelos *back-ends*, são necessários apenas 4 *front-ends* e 4 *back-ends*, desde que todos reconheçam a RI.

Uma RI é uma abstração para representar o código-fonte. Esta representação deve ser

- semanticamente fiel ao que foi escrito no código-fonte, ou seja, deve realizar exatamente o que foi implementado;
- independente das linguagens fonte e alvo (mas algumas ainda são dependentes da linguagem fonte);
- fácil de traduzir;
- fácil de otimizar.

Portanto, a RI deve trazer precisão, independência, simplicidade e reúso.

Um caso particular de RI se dá por meio de linguagens intermediárias. Toda linguagem intermediária é uma representação intermediária, mas o oposto não é verdade (nem toda representação intermediária é uma linguagem intermediária). A linguagem intermediária é desenvolvida para uma máquina específica, mesmo que hipotética. Por conveniência, pode-se ainda, segundo Rigo (2009), adotar várias RIs no processo de compilação.

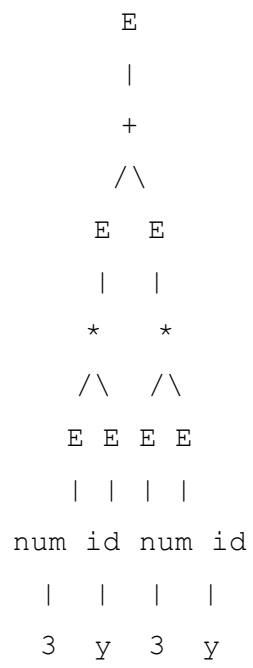
Algumas formas de RI incluem

- Árvores sintáticas otimizadas
- Notação pós-fixa e pré-fixa
- Código de três endereços

Árvores Sintáticas Otimizadas

Árvores sintáticas otimizadas são um tipo particular de grafo. Consistem no tipo mais simples de RI, ainda muito ligado ao código fonte. Na árvore sintática otimizada, os não terminais são eliminados, produções do tipo $A \rightarrow B$, $B \rightarrow C$ são substituídas $A \rightarrow C$ e a árvore é transformada em um grafo, eliminando a duplicação de avaliação de expressões em comum.

Exemplo:

Regras de Produção	Entrada	Árvore	Grafo
E → E+E	(3 * y) + (3 * y)		
E → E*E			
E → id | num			

Há expressões duplicadas sendo avaliadas!

A árvore sintática otimizada é usada como uma preparação para a interpretação do código ou para a geração de código alvo.

Notação pós-fixa e pré-fixa

As notações pós-fixa e pré-fixa facilitam a execução do código em uma máquina de pilha, tal como a JVM (*Java Virtual Machine*).

Exemplo:
1 + (3 * 4)

Notação pós-fixa	**Notação pré-fixa**
`PUSH 1;`	`ADD;`

```
PUSH 3;
PUSH 4;
MUL;
ADD;
PRINT
```

```
PUSH 1;
MUL;
PUSH 3;
PUSH 4;
PRINT
```

Exemplo:
(4 + 5) * 6

Notação pós-fixa

```
PUSH 4;
PUSH 5;
ADD;
PUSH 6;
MUL;
PRINT
```

Notação pré-fixa

```
MUL;
ADD;
PUSH 4;
PUSH 5;
PUSH 6;
PRINT
```

A JVM usa notação pós-fixa, também conhecida como ***notação polonesa reversa***[1].

A notação pré-fixa possui um formato mais parecido com o código de máquina e também pode ser executada por uma máquina de pilha (da direita para a esquerda).

```
+     1     2
ADD   #1    #2
```

Código de Três Endereços

No código de três endereços, cada instrução (linha de código) faz referência a no máximo três posições de memória ou variáveis. Assim, para cada nó interno da árvore sintática é gerado um temporário. Este é um formato considerado popular por alguns autores (ARATHIL, 2014;

[1] Notação polonesa reversa é uma forma de escrever expressões matemáticas com os operadores aparecendo após os seus operandos.

MELIKYAN, s.d.; SJÖLUND & GEBREMEDHIN, 2016).

Exemplo:

Original	Código de três endereços	Árvore sintática do código de três endereços

x := y + x * y;	tmp1 := x * y;
	tmp2 := y + tmp1;
	x := tmp2;
		:=
		/\
		/ \
		x + (tmp2)
		/\
		/ \
		y * (tmp1)
		/\
		/ \
		x y

Outro exemplo, aqui considerando a tipagem do valor constante (baseado em FEDOZZI, 2018):

Original	**Código de três endereços**

`x := a * (b + 40)`	`t0 := int(40)`
	`t1 := b + t0`
	`t2 := a * t1`
	`x := t2`

Instruções de um código de três endereços:

```
VAR := VAR OP VAR
VAR := OP VAR
VAR := VAR
GOTO LABEL
IF BOOLEAN GOTO LABEL
```

Funções, estruturas condicionais e estruturas de repetição são traduzidos para `GOTO`s. Expressões são readequadas para conterem no máximo três endereços, podendo ser divididas.

CURIOSIDADE

O compilador GCC utiliza várias representações intermediárias. Uma delas, a GIMPLE, utiliza a representação de três endereços (GCC TEAM, 2011). Um exemplo de código: **https://gcc.gnu.org/onlinedocs/gcc-4.3.6/gccint/GIMPLE-Example.html**.

Utilizando o GCC, é possível gerar a representação GIMPLE de um exemplo básico (arquivo hello-world.c) através do seguinte comando:

gcc hello-world.c -fdump-tree-gimple

O comando acima cria um arquivo chamado hello-world.c.004t.gimple. O conteúdo deste arquivo deve ser:

```
main ()
{
      int D.2250;
      {
            __builtin_puts (&"Hello, world!"[0]);
            D.2250 = 0;
            return D.2250;
      }
      D.2250 = 0;
      return D.2250;
}
```

O quadro acima mostra o código GIMPLE para o código do Hello, World. No entanto, as diferenças ficam mais visíveis se expressões que utilizem mais de três endereços forem utilizadas no código original. Isso pode ser visto ao gerar o código GIMPLE a partir do código abaixo, também disponível no arquivo teste-expressoes-gimple.c:

```
#include <stdio.h>
int main() {
      int a = 4;
```

```
      int b = 5;
      int c = 7;
      int d = a + b + c;
      printf("%i\n", d);
      return 0;
}
```

O código deve ser compilado novamente usando o comando:

`gcc teste-expressoes-gimple.c -fdump-tree-gimple`

O arquivo gerado está disponível em teste-expressoes-gimple.c.004t.gimple.

A principal vantagem do código intermediário de três endereços para o código objeto é que o intermediário não possui aspectos específicos da arquitetura, tais como registradores, referências a posições de memória etc, como mostra o exemplo da Figura 3.

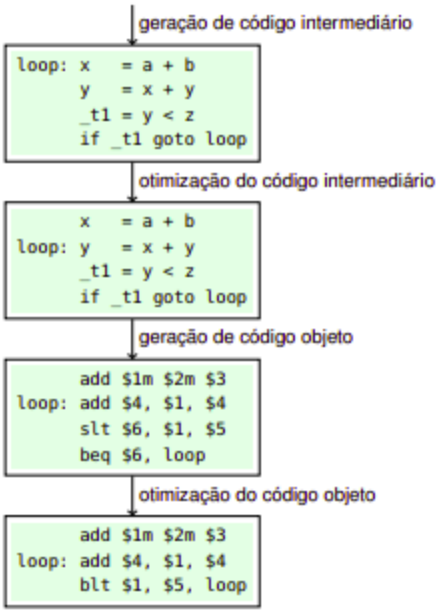


Figura 3. Ações do compilador após a análise semântica. Fonte: Malaquias, 2012.

Outras Formas de RI

Existem diversas outras formas de criar representações intermediárias. Entre elas, pode-se

destacar ainda:

- Os códigos para máquinas virtuais, tais como *Bytecodes* Java e *Common Intermediate Language* – CIL do *framework* .NET.
- Linguagens de alto nível, tais como C, utilizada como intermediária em implementações de algumas linguagens, entre elas Haskell (no *Glasgow Haskell Compiler* – GHC) e LISP (no *Embeddable Common Lisp* – ECL).

Otimização de Código

Após a tradução dirigida por sintaxe, o código gerado não é necessariamente eficiente, pois ele trabalha com a sequência de comandos de forma independente. Melhorias poderão ser aplicadas na etapa de otimização de código, mas deve-se ter em mente que o compilador precisa manter o sentido do programa.

As otimizações variam quanto ao escopo (local ou global) e (in)dependência de máquina, podendo ser aplicadas antes ou depois da geração do código de montagem (*assembly*).

Normalmente, o processo de otimização passa por duas fases: ***otimização do código intermediário*** e ***otimização de código objeto***.

- A otimização do código intermediário ainda é independente de arquitetura e possui alta reusabilidade. Nela se eliminam códigos inatingíveis, variáveis temporárias desnecessárias, expressões redundantes, alterações na ordem das instruções etc.
- Na otimização do código objeto, podem ocorrer substituições de instruções de máquina, priorizando o uso das mais rápidas, e realiza-se a alocação eficiente dos registradores sempre que possível.

Otimização ótima de código é um problema impossível de resolver, o que se faz é aplicar heurísticas para tentar conseguir o melhor código possível. Além disso, otimização de código toma tempo de compilação, razão pela qual nem sempre esta etapa é necessária. Tomando como base os códigos de exercícios feitos em aula, tipicamente pequenos e simples, nota-se que é melhor uma compilação mais rápida e sem otimizações, mas o mesmo não pode ser dito para um software comercial (DU BOIS, 2011).

Conforme Ricarte (2003), entre as otimizações possíveis estão:

1) Eliminação de desvios desnecessários
2) Eliminação de código redundante
3) Eliminação de código inalcançável
4) Aplicação de propriedades algébricas
5) Movimentação de código
6) Reúso de variáveis temporárias (DU BOIS, 2011)

1) Uma das otimizações citadas é a de ***eliminação de desvios desnecessários***.

Exemplo:

O código abaixo traz um comando `goto` que fará o desvio para `ROTULO1`. No entanto, a próxima instrução já é `ROTULO1`.

```
(...)
goto ROTULO1
ROTULO1: (...)
```

Assumindo que podem haver outros desvios para `ROTULO1`, a instrução rotulada é mantida, mas elimina-se o desvio (`goto`), pois ele é desnecessário:

```
(...)
ROTULO1: (...)
```

Observação: Caso o código não contenha mais desvios para `ROTULO1`, este rótulo também poderia ser eliminado.

2) ***Códigos redundantes*** são aqueles que são repetidos de forma desnecessária.

Exemplo:

```
x := 10;
y := 5;
x := y;
x := y;
```

Embora seja sintaticamente correto, o código do exemplo atribui o valor de `y` duas vezes a `x`, sem que exista alguma operação entre essas atribuições. Pode-se ainda assumir que existam outras instruções entre as duas atribuições repetidas, mas se estas instruções não alterarem o valor de `y`, o código permanecerá redundante. O compilador deve ser capaz de identificar essas situações e eliminar o código redundante.

Carregamento e armazenamento redundante também se enquadram nesta categoria.

Exemplo:

Código inicial	Código otimizado (notação de três endereços)
`(...)`	`(...)`
`a := b + 8;`	`d := b + 8`
`c := a;`	`e := d - 2`
`d := c;`	
`e := d - 2;`	

Observação 1: neste exemplo, assumiu-se que as variáveis removidas não são utilizadas posteriormente.
Observação 2: esta é uma das técnicas consideradas de otimização *peephole*.

3) ***Códigos inalcançáveis*** são as instruções presentes no código mas que nunca poderão ser executadas.

Exemplo:

```
(...)
goto ROTULO1
x := y
ROTULO1: (...)
```

A atribuição `x := y` é inalcançável, pois há um desvio na instrução acima. Desta forma, pode-se remover este comando.

4) A ***aplicação de propriedades algébricas*** pode fazer com que operações sejam

resumidas, eliminadas ou substituídas por outras menos custosas. Isto inclui substituir:

- x + 0 por x
- 0 + x por x
- x - 0 por x
- x / 1 por x
- 1 * x por x
- x * 1 por x
- 2 * x por x + x
- x^2 por x * x
- x / 2 por 0.5 * x
- x / (2^n) por deslocamento (*shift*) à direita, se x for uma variável inteira
- x * (2^n) por deslocamento (*shift*) à esquerda, se x for uma variável inteira

Exemplo:

```
#include <stdio.h>
int main() {
    unsigned int x = 32;
    unsigned int y = 16;

    printf("x << 1 = %d\n", x << 1);
    printf("x << 2 = %d\n", x << 2);
    printf("x * 2 = %d\n", x * 2);
    printf("x * 2^2 = %d\n", x * 2 * 2);
    printf("y >> 1 = %d\n", y >> 1);
    printf("y >> 2 = %d\n", y >> 2);
    printf("y / 2 = %d\n", y / 2);
    printf("y / 2^2 = %d\n", y / (2 * 2));

    return 0;
}
```

Exemplo de reúso de expressões:

```
a := 10 + 5;
b := 10 + 8 + 5;
// Poderia ser alterado para:
a := 10 + 5;
b := 8 + a;
```

5) Trechos de código podem ser ***movimentados*** por conveniência. Por exemplo:

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

No código acima, há cálculos sendo realizados repetidamente (e desnecessariamente) dentro do laço. Isto geraria o seguinte código intermediário:

```
ROTULO1:  _t1 := 10 * j
       if i >= _t1 goto ROTULO2
       _t2 := 2 * j
       a[i] := i + _t2
       i := i + 1
       goto ROTULO1
ROTULO2: (...)
```

Este código poderia ser melhorado para:

```
_t1 := 10 * j
_t2 := 2 * j
ROTULO1:  if i >= _t1 goto ROTULO2
       a[i] := i + _t2
       i := i + 1
       goto ROTULO1
ROTULO2: (...)
```

E depois, considerando as propriedades algébricas:

```
_t1 := 10 * j
_t2 := j + j
ROTULO1:  if i >= _t1 goto ROTULO2
        a[i] := i + _t2
        i := i + 1
        goto ROTULO1
ROTULO2: (...)
```

6) ***Variáveis temporárias*** geradas no esquema de tradução para código intermediário de três endereços podem ser facilmente ***reusadas***:

Exemplo:

```
x:= ((a * b) + (c * d)) - (e * f);
```

Código de três endereços

```
temp1 := a * b;
temp2 := c * d;
temp3 := temp1 + temp2;
temp4 := e * f;
temp5 := temp3 - temp4;
x := temp5;
```

Código de três endereços reusando variáveis temporárias

```
temp1 := a * b;
temp2 := c * d;
temp1 := temp1 + temp2;
temp2 := e * f;
temp1 := temp1 - temp2;
x := temp1
```

Observação: as duas últimas linhas do código da direita ainda poderiam virar:

```
x := temp1 - temp2;.
```

Existem estratégias para otimizar o uso de temporários através de GAD (***Grafos Acíclicos Dirigidos***). Para saber mais sobre o assunto, recomenda-se a leitura de Rangel (2000).

Custo do Código Objeto (baseado em Du Bois 2011)

O custo do código objeto é o total de instruções em um código objeto gerado para um trecho de código.

Exemplo de código objeto para uma máquina com um registrador acumulador:

```
LDA X      →     AC := X
STA X      →     X := AC
ADD X      →     AC := X + AC
SUB X      →     AC := AC - X
MUL X      →     AC := AC * X
```

Dessa forma o código intermediário para `TEMP1 := A + B` será:

```
LDA A
ADD B
STA TEMP1
```

Ou seja, esta expressão `(TEMP1 := A + B)` gera três instruções.

Seja a expressão (A + B) * (A - B) * (A - C) * (B - C)

A notação de três endereços pode ser gerada da seguinte forma:

```
T1 := A + B;
T2 := A - B;
T3 := T1 * T2;
T4 := A - C;
T5 := B - C;
T6 := T2 * T4;
T7 := T6 * T5;
```

Verifique a quantidade de instruções do código abaixo:

```
T1 := A + B;          3 instruções
T2 := A - B;          3 instruções
T3 := T1 * T2;        3 instruções
T4 := A - C;          3 instruções
T5 := B - C;          3 instruções
T6 := T3 * T4;        3 instruções
T7 := T6 * T5;        3 instruções
```

Total: 21 instruções

Rearranjando as operações é possível aproveitar os valores já presentes no registrador, evitando carregamentos desnecessários:

```
T5 := B - C;          3 instruções
T4 := A - C;          3 instruções
T2 := A - B;          3 instruções
T6 := T2 * T4;        2 instruções
T7 := T6 * T5;        2 instruções
T1 := A + B;          3 instruções
T3 := T1 * T2;        2 instruções
```

Total: 18 instruções

Reaproveitando temporários, o código poderia ficar assim:

```
T5 := B - C;
T4 := A - C;
T2 := A - B;
T6 := T2 * T4;
T4 := T6 * T5;
T2 := A + B;
T5 := T2 * T4;
```

Assim, quatro temporários (`T2`, `T4`, `T5` e `T6`) foram utilizados, uma redução de 42,86% em comparação ao primeiro código gerado, que continha sete temporários.

Conclusão

O uso de representações intermediárias é fundamental para o reaproveitamento dos *front-ends* e para a otimização de código. A capacidade de traduzir um único conjunto de instruções intermediárias para múltiplas plataformas facilita a portabilidade do software, reduzindo significativamente o esforço necessário para adaptar o código a diferentes ambientes de execução.

Códigos intermediários devem ser fáceis de gerar, otimizar e traduzir para o código de máquina final. Além disso, uma boa representação intermediária deve ser projetada com simplicidade e eficiência.

Os códigos gerados podem ser otimizados de diversas maneiras, eliminando desvios desnecessários, códigos redundantes e inalcançáveis, aplicando propriedades algébricas, alterando a ordem das instruções e reaproveitando variáveis temporárias. Tudo isso contribui para que os códigos possam ser otimizados pelos compiladores.

Exercícios

1) Analisando o código abaixo, qual, dentre as seis formas de otimizações vistas em aula, poderia ser aplicada? Por quê?

```
#define DEBUG 0
/* (...) */
if (DEBUG) {
      /* (...) */
}
```

2) Existe alguma otimização possível no código abaixo? Justifique.

```
unsigned int f(unsigned int x) {
      return x++;
}
```

3) Assumindo que os valores não são modificados após o trecho de código abaixo, qual deles pode ser considerado o melhor? Justifique.

Código (a):

```
int a = 1;
int b;
while (a <= 10) {
```

Código (b):

```
int b = 5;
while (b < 50) {
      b = b + 5;
```

```
    b = a * 5;                          }
    a++;                                printf("%d\n", b);
}
printf("%d\n", b);
```

4) Como o código abaixo pode ser melhorado?

```
if (a > b)
    m = 1;
else if (a < b)
    m = 2;
else if (a == b)
    m = 0;
else
    m = -1;
```

5) (ENADE 2021) Um compilador é um *software* que traduz um programa descrito em uma linguagem de alto nível para um programa equivalente em código de máquina para um processador. Em geral, um compilador não produz diretamente o código de máquina, mas sim, um programa em linguagem simbólica (*assembly*) semanticamente equivalente ao programa em linguagem de alto nível. O programa em linguagem simbólica é, então, traduzido para o programa em linguagem de máquina através de montadores. Para realizar esta tarefa, o compilador executa a análise léxica, sintática e semântica do código-fonte do programa que está sendo executado em linguagem abstrata para depois gerar o código de máquina.

   Considerando as informações do texto, avalie as afirmações a seguir.

   I. O analisador sintático tem a função de verificar se a sequência de símbolos gerada pelo analisador léxico compõe um programa válido ou não.

   II. Na análise léxica, o analisador irá identificar cada símbolo que tenha significado para linguagem, gerando a mesma classificação para Java, Pascal ou outra linguagem.

   III. O analisador semântico utiliza o código fonte para verificar incoerências quanto ao significado das construções implementadas.

IV. A fase de otimização do código procura melhorar o código intermediário, visando um código de máquina mais rápido em termos de execução.

É correto apenas o que se afirma em

a) I e IV.
b) II e III.
c) II e IV.
d) I, II e III.
e) I, III e IV.

6) Pesquise e descubra o que são as otimizações *peephole*.

Respostas

1) Eliminação de código inalcançável (otimização tipo 3). Como `DEBUG` é uma constante com valor `0`, a construção `if` da linha 268 nunca será alcançada. Desta forma, essa estrutura condicional pode ser removida do código.
2) Trocar `return x++`" por `return x;`, pois o pós-incremento não fará sentido já que `x` é declarada localmente.
3) O código (b), pois usa uma variável a menos e obtém o mesmo resultado, além de realizar dez atribuições, dez avaliações lógicas na estrutura condicional e nove somas. A soma é uma operação menos custosa que a multiplicação, que está presente no código (a). O código (a) avalia onze expressões lógicas, realiza 21 atribuições, dez multiplicações e dez somas. Além disso, todas as multiplicações, exceto a última, são desnecessárias, e alterando a condição para `a < 10` e a expressão `b = a * 5` para imediatamente após o *loop* o resultado já seria melhor, embora ainda pior que o da solução (b).
4) Eliminando o trecho inalcançável, isto é, removendo o teste que atribui `-1` a `m`. O código ficaria assim:

```
if (a > b)
     m = 1;
else if (a < b)
```

```
      m = 2;
else
      m = 0;
```

5) e.
6) Questão de pesquisa.

Referências

ARATHIL, Shineraj. 3 Address Code Generation. 2014. Disponível em: https://pt.slideshare.net/SHINEAPPLE4/three-address-code-in-compiler-design/3. Acesso em: 01 jun. 2024.

CARDOZO, Eleri. Capítulo 9. Geração de código e otimização. s.d. Disponível em: https://www.dca.fee.unicamp.br/~eleri/ea876/04/cap9.pdf. Acesso em: 01 jun. 2024.

DU BOIS, André Rauber. Notas de Aula sobre Compiladores. 03 nov. 2011.

FEDOZZI, Regina. Compiladores. Londrina: Editora e. Distribuidora Educacional S.A., 2018. 264 p. ISBN 978-85-522-1099-3. Disponível em: http://cm-kls-content.s3.amazonaws.com/201802/INTERATIVAS_2_0/COMPILADORES/U1/LIVRO_UNICO.pdf. Acesso em: 01 jun. 2024.

GCC TEAM. 12 GIMPLE. GCC Online Documentation. 2011. Disponível em: https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html. Acesso em: 01 jun. 2024.

MALAQUIAS, José Romildo. Construção de Compiladores - Capítulo 1: Introdução. 2012. Disponível em: http://www.decom.ufop.br/romildo/2012-2/bcc328/slides/01-introducao. Acesso em: 01 jun. 2024.

MELIKYAN, Vazgen. Compiler Optimization and Code Generation. s.d. Disponível em: https://www.ece.ucsb.edu/bears/class/ece253/compiler_opt/c2.ppt. Acesso em: 01 jun. 2024.

RICARTE, Ivan Luiz Marques. Otimização de Código. 2003. Disponível em: https://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node76.html. Acesso em: 01 jun. 2024.

RIGO, Sandro. Formas de Representação Intermediária. 2009. Disponível em: https://www.ic.unicamp.br/~sandro/cursos/mc910/2009/slides/cap4%20-%20Formas%20de%20Representa%E7%E3o%20Intermedi%E1ria.pdf. Acesso em: 01 jun. 2024.

SJÖLUND Martin; GEBREMEDHIN, Mahder. Compiler Construction. 2016. Disponível em:

https://www.ida.liu.se/~TDDB44/lessons/lessons2016/TDDB44Seminar3.pdf. Acesso em: 01 jun. 2024.