Construção de Compiladores
Daniel Lucrédio, Helena Caseli, Mário César San Felice e Murilo Naldi
Tópico 08 - Geração de código - Lista de Exercícios Resolvida
(Última revisão: fev/2020)

1. Qual a diferença entre código intermediário e código da máquina alvo? Por que é interessante utilizar código intermediário ao invés de gerar diretamente o código de máquina?

R. O código intermediários omite uma série de detalhes da máquina alvo, como endereços de registradores, endereços de memória, chamadas do sistema operacional, entre outras coisas. Gerar o código intermediário primeiro facilita o processo, pois durante a geração do código intermediário pode-se efetuar uma série de tarefas independentes da máquina, como a linearização de expressões aritméticas, cálculo de endereços relativos, expansão de procedimentos e funções, entre outras. E o gerador de código de máquina alvo pode realizar uma série de tarefas específicas da máquina, considerando a arquitetura, as instruções da máquina, as limitações de memória, entre outras. Com isso, tem-se as seguintes vantagens:

- cada gerador lida com detalhes diferentes, deixando o projeto mais modularizado e fácil de manter;
- é possível realizar algumas otimizações independente da máquina-alvo, também simplificando o gerador final;
- é possível reutilizar o código intermediário para gerar código para diferentes máquinas-alvo.

2. Quais são as diferenças entre gerar código por meio de ações semânticas inseridas na própria gramática e o uso de um visitante? Quais as vantagens e desvantagens de cada abordagem?

R: São as mesmas da análise semântica (ver Lista anterior):
- Ao inserir ações semânticas na própria gramática, um gerador automático pode produzir um parser que realiza as ações durante a própria análise sintática. Ou seja, o analisador vai analisando a semântica AO MESMO TEMPO em que analisa a sintaxe e cria a árvore. Em contrapartida, um visitante atua sobre uma árvore já pronta, APÓS a análise sintática ter sido concluída.
- As vantagens de realizar análise semântica na própria gramática são: a) abordagem mais simples e direta. b) as ações ficam visíveis dentro da gramática, o que pode facilitar seu entendimento em casos mais simples, ajudando na sua criação/manutenção
- As desvantagens de realizar análise semântica na própria gramática são: a) as ações são executadas durante a análise sintática, portanto é necessário posicioná-las em local correto para que sejam executadas após o código necessário ter sido analisado. b) pelo mesmo motivo acima, é impossível, a partir de uma ação semântica, acessar trechos de código que só serão analisados mais para a frente (ex: um método que só foi declarado no final do arquivo não estará na tabela de símbolos no começo do arquivo). c) caso haja muitas ações, a gramática poderá se tornar ilegível, dificultando o trabalho. d) fica difícil realizar duas tarefas distintas sobre a mesma gramática, pois o código estará misturado (ex: durante a declaração de variáveis, é preciso manipular a tabela de símbolos e gerar código, ambas as ações ficarão misturadas na gramática).
- As vantagens de realizar análise semântica em um visitante são: a) preserva a legibilidade da gramática, mantendo-a em um arquivo separado. b) possibilita o acesso a qualquer parte do programa a qualquer momento, já que a árvore já está toda pronta. c) possibilita a criação de múltiplos visitantes, cada um com uma função diferente (ex: um visitante para

verificar tipos, um visitante para analisar consistências diversas, um visitante para gerar código, etc). o que deixa o código mais modularizado e fácil de se manter.
- As desvantagens de realizar análise semântica em um visitante são: a) a necessidade de realizar duas (ou mais) análises distintas em momentos separados, o que pode deixar a compilação menos eficiente. b) pode causar duplicação de código (ex: tanto a análise de tipos como a geração de código dependem da tabela de símbolos, portanto pode haver código de manipulação da tabela de símbolos duplicado em dois visitantes). É possível usar herança e modularização para resolver os principais problemas, no entanto, mas é preciso cuidado. c) a associação entre as ações e a gramática é menos visível, visto que as regras sintáticas e as ações semânticas estão em arquivos separados.

3. Quais as vantagens e desvantagens de se gerar código em uma linguagem que já existe (como C ou Java)?

R: As vantagens são a facilidade de reaproveitar todo o trabalho já consagrado dessas linguagens consolidadas, como otimizações e suporte para múltiplas plataformas. As desvantagens são a necessidade de uma segunda compilação para possibilitar a execução final. Além disso, perde-se a oportunidade de se criar um código específico para uma plataforma que não é suportada por essa linguagem.

4. Qual a principal diferença entre usar um gerador de parsers (como ANTLR) e um workbench de linguagens (como Xtext)?

R: Um gerador de parsers apenas lida com a análise léxica/sintática, deixando todo o resto por conta do programador, incluindo a análise semântica e geração de código. Um workbench de linguagens já realiza outras tarefas, como algumas análises semânticas, suporte gráfico para criação das linguagens, e apoio ferramental que facilita a vida do utilizador da linguagem.

5. Cite quais são os 2 tipos de código intermediário apresentados em aula e suas características principais. Quais são as diferenças entre eles?

R. O **código de três endereços** é uma forma de representação intermediária na qual cada instrução pode envolver, no máximo, três endereços de memória:

x = y op z

em que op é um operador aritmético (+ ou , por exemplo) ou algum outro operador que possa operar sobre os valores de y e z. Atenção: o uso de x é diferente do uso de y e z já que y e z podem representar constantes ou literais sem endereços na execução, enquanto é necessário conhecer o endereço de x para que a atribuição possa ser realizada. Isso fica bem claro com o P-código. Outra característica relevante do código de três endereços é a necessidade de utilizar temporários (t1, t2 etc.) para armazenar os valores intermediários das operações. Esses temporários correspondem aos nós interiores da árvore sintática e representam seus valores computados; podem ser armazenados na memória ou em registradores.

O **P-código**, por sua vez, surgiu como um código de montagem alvo padrão produzido pelos compiladores Pascal e foi projetado como código de uma máquina hipotética baseada em pilhas (P-máquina) com interpretador para diversas máquinas reais. Como foi projetado para ser executado diretamente, o P-código contém uma descrição implícita de um ambiente de execução, entre outros detalhes abstraídos quando o consideramos como código intermediário nesse curso.

Comparação

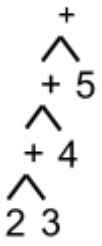
Código de três endereços vs. **P-código**

Código de três endereços	**P-código**
- é mais compacto (menos instruções) - é mais próximo do código de máquina - é autossuficiente no sentido de que não precisa de uma pilha para representar o processamento - precisa de temporários para armazenar os valores das computações intermediárias	- é mais próximo do código de máquina - as instruções exigem menos endereços (os endereços omitidos estão na pilha implícita) - não precisa de temporários, uma vez que a pilha contém todos os valores temporários

6. Apresente a sequência de instruções de código de três endereços correspondente a cada uma das expressões aritméticas a seguir. Quais são as árvores sintáticas abstratas que correspondem à geração de código?

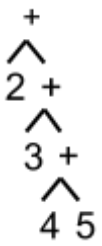
a) 2+3+4+5

R.




```
t1 = 2 + 3
t2 = t1 + 4
t3 = t2 + 5
```

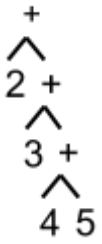
ou




```
t1 = 4 + 5
t2 = 3 + t1
t3 = 2 + t2
```

b) 2+(3+(4+5))

R.



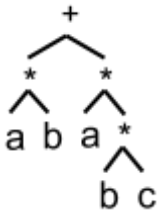

```
t1 = 4 + 5
t2 = 3 + t1
t3 = 2 + t2
```

c) a*b+a*b*c

R.


```
read(a)
read(b)
t1 = a * b
t2 = a * b
t3 = t2 * c
t4 = t1 + t3
```

ou




```
read(a)
read(b)
t1 = a * b
t2 = b * c
t3 = a * t2
t4 = t1 + t3
```

7. Apresente a sequência de instruções de P-código correspondente às expressões aritméticas do exercício anterior. Obs: considere os seguintes operadores e suas ações correspondentes:

Operador	Ação correspondente
rdi	Retira um endereço A do topo da pilha, lê um valor X da entrada e armazena X na memória, no endereço A
wri	Retira um valor X do topo da pilha e escreve X na saída
lda A	Insere o endereço A no topo da pilha
ldc C	Insere a constante C no topo da pilha
lod A	Lê o conteúdo da memória no endereço A e o insere no topo da pilha
mpi	Retira um valor X do topo da pilha, retira outro valor Y do topo da pilha, e insere X * Y no topo da pilha
dvi	Retira um valor X do topo da pilha, retira outro valor Y do topo da pilha, e insere Y / X no topo da pilha
adi	Retira um valor X do topo da pilha, retira outro valor Y do topo da pilha, e insere X + Y no topo da pilha
sbi	Retira um valor X do topo da pilha, retira outro valor Y do topo da pilha, e insere Y - X no topo da pilha
sto	Retira um valor X do topo da pilha, retira um endereço A do topo da pilha, e armazena X na memória, no endereço A
grt	Retira um valor X do topo da pilha, retira outro valor Y do topo da pilha, e armazena Y > X (um booleano) no topo da pilha
let	Retira um valor X do topo da pilha, retira outro valor Y do topo da pilha, e armazena Y < X (um booleano) no topo da pilha
gte	Retira um valor X do topo da pilha, retira outro valor Y do topo da pilha, e armazena Y >= X (um booleano) no topo da pilha
lte	Retira um valor X do topo da pilha, retira outro valor Y do topo da pilha, e armazena Y <= X (um booleano) no topo da pilha
equ	Retira um valor X do topo da pilha, retira outro valor Y do topo da pilha, e armazena X == Y (um booleano) no topo da pilha
neq	Retira um valor X do topo da pilha, retira outro valor Y do topo da pilha, e armazena X != Y (um booleano) no topo da pilha
and	Retira um valor booleano X do topo da pilha, retira outro valor booleano Y do topo da pilha, e armazena X && Y (um booleano) no topo da pilha
or	Retira um valor booleano X do topo da pilha, retira outro valor booleano Y do topo da pilha, e armazena X || Y (um booleano) no topo da pilha
lab L	Sem efeito na execução. Apenas marca uma posição de código com um rótulo L
ujp L	Salta para a instrução marcada com L
fjp L	Retira um valor booleano X do topo da pilha e caso seja falso, salta para a instrução marcada com L. Caso seja verdadeiro, não executa nada.
stp	Interrompe a execução

R.
a)2+3+4+5

```
ldc 2
ldc 3
adi
ldc 4
adi
ldc 5
adi
```

ou

```
ldc 4
ldc 5
adi
ldc 3
adi
ldc 2
adi
```

b) 2+(3+(4+5))

```
ldc 2
ldc 3
ldc 4
ldc 5
adi
adi
adi
```

ou

```
ldc 4
ldc 5
adi
ldc 3
adi
ldc 2
adi
```

c) a*b+a*b*c

```
Obs: endereço de a=0,b=1,c=2
rdi 0
rdi 1
rdi 2
lod 0
lod 1
mpi
lod 0
lod 1
mpi
lod 2
```

```
      mpi
      adi
```

8. Escreva as ações semânticas para geração de código de três endereços para a gramática de expressões aritméticas de inteiros a seguir. Teste, gerando o código de três endereços para todas as expressões da questão 2

R.

```
grammar Expressoes;

@members{
int contador=1;
String newTemp() {
    return "t"+(contador++);
}
}

programa returns [ String tacode ]:
    expressao
    { $tacode = $expressao.tacode; }
;
expressao returns [ String tacode, String nome ]:
    exp2=expressao op1 termo
    { $nome = newTemp();
      $tacode = $exp2.tacode + "\n" +
                $termo.tacode + "\n" +
                $nome + "=" +
                $exp2.nome +
                $op1.nome +
                $termo.nome;
    }
    | termo
    { $tacode = $termo.tacode;
      $nome = $termo.nome; }
;
termo returns [ String tacode, String nome ]:
    t2=termo op2 fator
    { $nome = newTemp();
      $tacode = $t2.tacode + "\n" +
                $fator.tacode + "\n" +
                $nome + "=" +
                $t2.nome +
                $op2.nome +
                $fator.nome;
    }
    | fator
    { $tacode = $fator.tacode;
      $nome = $fator.nome; }
;
fator returns [ String tacode, String nome ]:
    '(' expressao ')'
```

```
    { $tacode = $expressao.tacode;
      $nome = $expressao.nome; }
    | NUM
    { $tacode = "";
      $nome = $NUM.getText(); }
    | ID
    { $tacode = "";
      $nome = $ID.getText(); }
    ;
op1 returns [ String nome ]:
    '+' {$nome="+";}
    | '-' {$nome="-";}
;
op2 returns [ String nome ]:
    '*' {$nome="*";}
    | '/' {$nome="/";}
;
NUM: '0'..'9'+;
ID: ('a'..'z'|'A'..'Z')+;
WS: ( ' ' | '\n' | '\r' | '\t' ) -> skip;
```

9. Considerando-se a mesma gramática do exercício anterior, escreva as ações semânticas para a geração de P-código. Teste, gerando o P-código para todas as expressões da questão 2

R.

```
grammar Expressoes;

programa returns [ String pcode ]:
    expressao
    { $pcode = $expressao.pcode; }
;
expressao returns [ String pcode ]:
    exp2=expressao op1 termo
    { $pcode = $exp2.pcode + "\n" +
               $termo.pcode + "\n" +
               $op1.pcode;
    }
    | termo
    { $pcode = $termo.pcode; }
;
termo returns [ String pcode ]:
    t2=termo op2 fator
    { $pcode = $t2.pcode + "\n" +
                $fator.pcode + "\n" +
                $op2.pcode;
    }
    | fator
    { $pcode = $fator.pcode; }
;
fator returns [ String pcode ]:
    '(' expressao ')'
```

```
    { $pcode = $expressao.pcode; }
    | NUM
    { $pcode = "ldc "+ $NUM.getText(); }
    | ID
    { $pcode = "lod "+ $ID.getText(); }
    ;
op1 returns [ String pcode ]:
    '+' {$pcode="adi";}
    | '-' {$pcode="sbi";}
;
op2 returns [ String pcode ]:
    '*' {$pcode="mpi";}
    | '/' {$pcode="dvi";}
;
NUM: '0'..'9'+;
ID: ('a'..'z'|'A'..'Z')+;
WS: ( ' ' | '\n' | '\r' | '\t' ) -> skip;
```