

Compiladores – Geração de Código

Fabio Mascarenhas – 2015.1

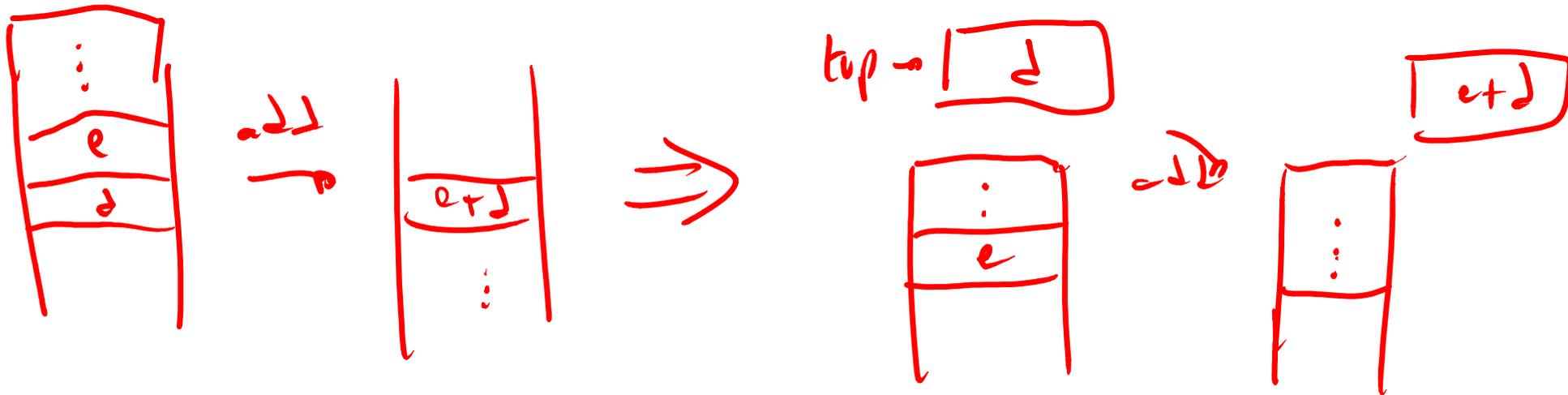
<http://www.dcc.ufrj.br/~fabiom/comp>

Máquinas de Pilha

- Uma máquina de pilha é um tipo de processador em que todos os valores temporários são armazenados em uma pilha
 - Não são usados registradores
- Toda operação em uma máquina de pilha desempilha seus operandos, faz a operação e empilha o resultado
- Instruções também podem empilhar valores constantes, ou o conteúdo de variáveis locais e endereços da memória (variáveis globais)
- Compilar para máquinas de pilha é bem fácil, mas menos eficiente que usar registradores

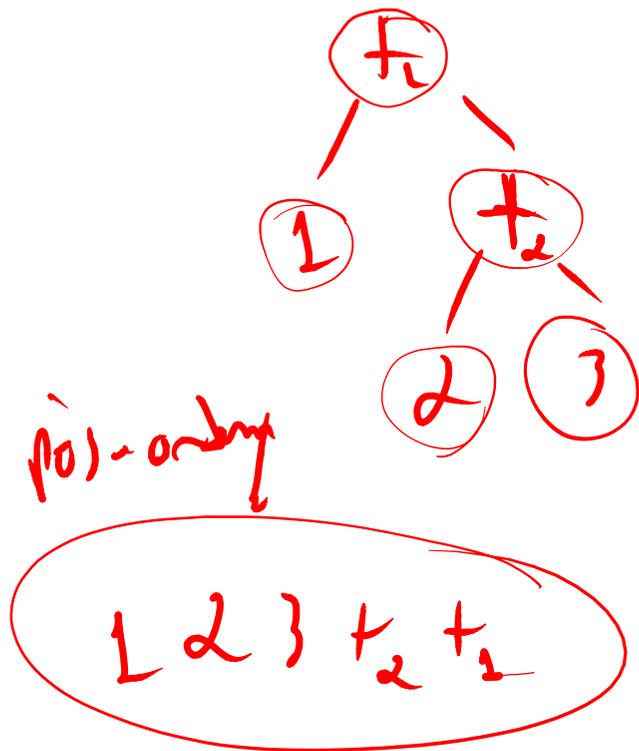
Pilha + Acumulador

- Uma otimização da máquina de pilha é manter o topo da pilha sempre em um registrador, o *acumulador*
- Algumas operações da máquina ficam mais eficientes, mas podemos usar as mesmas operações de uma máquina de pilha comum



Compilando expressões

- Para ter uma intuição de como a geração de código funciona para uma máquina de pilha, vamos gerar código para $1 + (2 + 3)$:



```
icload 1  
icload 2  
icload 3  
iadd  
iadd
```

Geração de Código para TINY em x86

- Vamos usar um modelo de máquina de pilha para gerar código para TINY com procedimentos para x86

- As instruções de nossa máquina de pilha serão implementadas por instruções de x86, usando o registrador **EAX** como acumulador e a pilha do processador como o resto da pilha

- Para simplificar, vamos tratar apenas de variáveis inteiras e booleanos

- Nossa máquina de pilha terá 15 instruções: **getglobal**, **putglobal**, **icload**, **iload**, **istore**, **iadd**, **isub**, **imul**, **idiv**, **invoke**, **if_icmpneq**, **if_icmpgeq**, **jmp**, **read**, **write**, **pop**

locos e primitivos

demora de proc.

*seta incremental
setas condicionais*

globais

constantes

o fctro L unidades

- A organização e nomes lembram os de máquinas virtuais de pilha, como a JVM

Contexto de Geração de Código

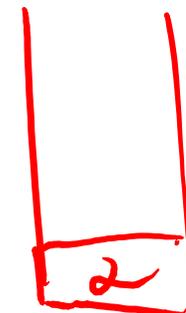
- Vamos criar uma classe para ser o *contexto* de geração de código
- O contexto implementa as instruções da máquina de pilha, gerando código x86 para elas em um buffer
- Vamos usar um contexto para cada procedimento, e depois costurar o código dos procedimentos junto com o código do corpo principal do programa e o código que declara variáveis globais
- Ele gerencia também os *labels* do programa, usados nas instruções de salto
- Os métodos de geração de código da AST só vão precisar de preocupar em chamar os métodos do contexto que correspondem às instruções da máquina

Tabelas de Símbolos e Endereços

- A geração de código também precisa de tabelas de símbolos que irão mapear nomes de variáveis e procedimentos em *endereços*
- O endereço diz se ela é global ou local, e como acessá-la: com seu *nome simbólico*, se a variável é global, ou com sua *profundidade*, se ela é local
- Com o local de uma variável podemos gerar código para empilhar seu valor, ou para desempilhar o que está no topo da pilha e escrevê-lo na variável
- O contexto de geração de código precisa também guardar uma *marca d'água*, a profundidade mais alta usada para uma local naquele contexto



a: 1 max: 4
b: 2
c: 3
d: 2 e: 3



marca: 3

Geração de Código - Comandos

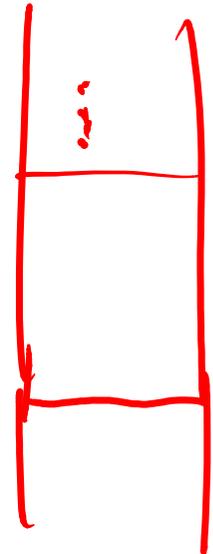
- Comandos precisam deixar a pilha do mesmo jeito que encontraram
- A geração depende do *contexto* corrente, e da tabela de símbolos que associa nomes a endereços

```
if x < 5 then  
- write 0  
else  
- write 1  
end
```



if

```
getglobal x  
icload 5  
if_icmpge $else  
icload 0  
write  
jmp $fim  
$else:  
icload 1  
write  
$fim:
```



Geração de Código - Expressões

- Expressões devem deixar a pilha com um elemento a mais no topo: o valor final da expressão
- A geração depende do contexto e da tabela de símbolos de endereços, como nos comandos

$x + (2 + y)$



```
getglobal x  
icload 2  
iload 1      ; y é local 1  
iadd  
iadd
```

- Cada subexpressão da expressão acima tem o efeito de empilhar o seu valor; ao fim a pilha será a original, mais o valor da expressão inteira

Geração de Código - Controle

- Expressões relacionais e condicionais normalmente são usadas para *controlar* a execução, através de saltos condicionais
- Normalmente temos código mais compacto se uma expressão condicional gera código que salte para determinado label se ela for *falsa* ao invés de verdadeira
 - Saltar para o bloco else, saltar para a saída do laço while, saltar para o início do corpo no laço repeat...
- Normalmente as arquiteturas sempre associam operações relacionais a saltos, então se queremos um valor booleano precisamos fazer saltos para código que carrega os valores equivalentes a true ou false