

# Compiladores – Geração de Código

---

Fabio Mascarenhas - 2013.1

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

# O Back-end

---

- Até agora vimos as fases do *front-end* do compilador:
  - Análise Léxica
  - Análise Sintática
  - Análise Semântica
- O *front-end* verifica se o programa está bem formado, de acordo com as regras da linguagem, e o coloca em uma estrutura adequada para seu processamento
- O *back-end* cuida da transformação do programa para a linguagem destino, e de transformações feitas no programa para tornar o código final mais eficiente

# Geração de Código

---

- Vamos ver a forma mais simples de back-end, que gera código diretamente a partir da AST do programa, sem se preocupar em melhorar o código resultante
- Mas mesmo um gerador de código ingênuo pode ter uma implementação complexa, a depender da distância entre a linguagem fonte e a linguagem destino
- Vamos ver a geração de código para duas linguagens destino:
  - Assembler x86, para nosso compilador TINY
  - Assembler JVM, para MiniJava

# Organização da Memória

---

- Antes de tratar da geração de código em sim, precisamos entender como é a estrutura do programa quando ele está sendo executado
- Quais recursos o programa usa em sua execução, e como eles se espalham na memória
- Que construções em tempo de execução correspondem às construções que temos em tempo de compilação: variáveis globais, variáveis locais, procedimentos, parâmetros, métodos, classes, objetos...
- Todas essas construções precisam estar refletidas de alguma forma no código gerado!

# Ativações e Alcance

---

- Uma chamada de um procedimento (ou função, ou método)  $p$  é uma *ativação* de  $p$
- O *alcance* de uma ativação de  $p$  compreende todos os passos para executar  $p$ , incluindo todos os passos para executar procedimentos chamados por  $p$
- O *alcance* de uma variável  $x$  é a porção da execução do programa na qual  $x$  está definida
  - Em geral, está ligado ao *escopo* de  $x$ , mas nem sempre
  - Alcance é um dinâmico, enquanto escopo é estático

# Alcance x Escopo

- No código em *JavaScript* abaixo, o escopo e o alcance do parâmetro *n* são bem diferentes:

```
function cont(n) {  
  return function () {  
    n = n + 1;  
    return n;  
  }  
}  
var c1 = cont(1);  
console.log(c1());  
console.log(c1());  
console.log(n);
```

*escopo* [

~~n = 2~~ / ~~3~~

2  
3  
ReferenceError: n is not defined

→

↗

# Árvore de Ativações

---

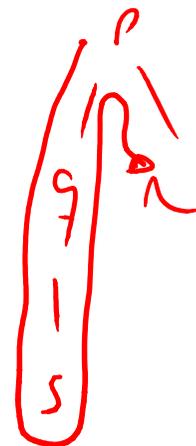
- Quando um procedimento  $p$  chama um procedimento  $q$ ,  $q$  sempre retorna antes do retorno de  $p$



- O alcance das ativações sempre é corretamente aninhado
- Isso quer dizer que as ativações durante a execução de um programa formam uma *árvore*



- A execução corresponde a um caminho nessa árvore *em profundidade*



# Árvore de Ativações - Exemplo

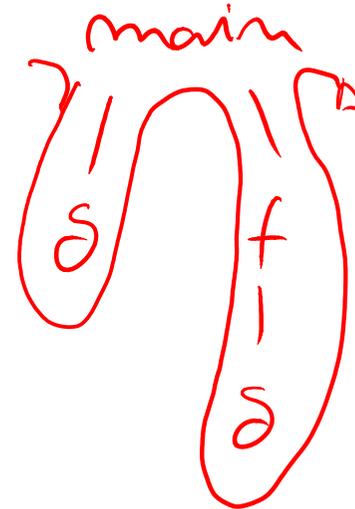
---

- Vamos desenhar a árvore de ativações para o programa TINY abaixo:

```
procedure g()  
  x := 1  
end;
```

```
procedure f()  
  g()  
end;
```

```
var x: int;  
g();  
f();  
write x
```

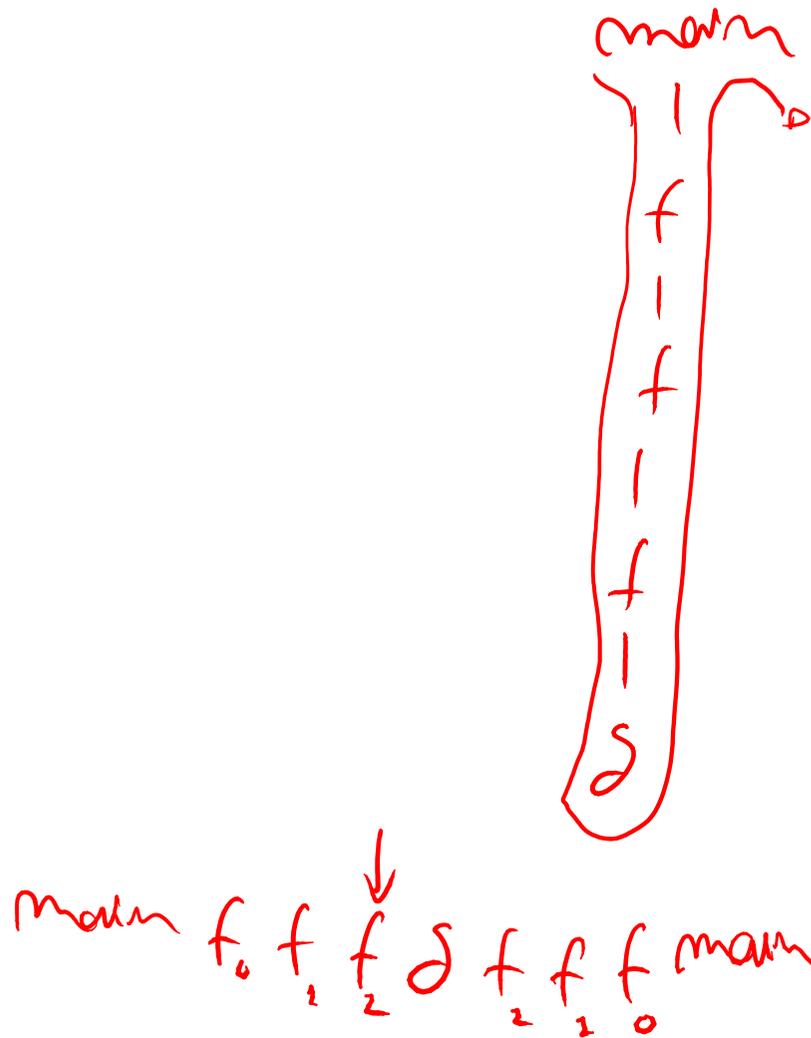


main g main f g f main

# Árvore de Ativações - Exemplo

- Vamos desenhar a árvore de ativações para o programa TINY abaixo:

```
procedure g()  
  x := 1  
end;  
  
procedure f()  
  var n: int;  
  n := x;  
  if n < 2 then  
    g()  
  else  
    x := n - 1;  
    f();  
    x := n * x  
  end  
end;  
  
var x: int;  
x := 3;  
f();  
write x
```



# Árvore de Ativações - Exemplo

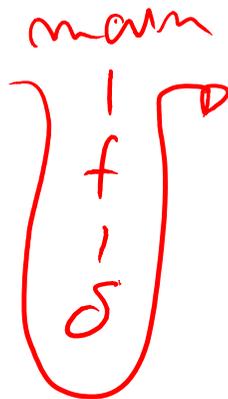
---

- Vamos desenhar a árvore de ativações para o programa TINY abaixo:

```
procedure g()  
  x := 1  
end;  
  
procedure f()  
  var n: int;  
  n := x;  
  if n < 2 then  
    g()  
  else  
    x := n - 1;  
    f();  
    x := n * x  
  end  
end;  
  
var x: int;  
read x;  
f();  
write x
```

Depende de x!

x = 1



# Árvores de Ativação

---

- A árvore de ativação depende da execução do programa, e pode ser diferente a depender da entrada para o programa
- Ou seja, a árvore de ativação do programa não pode ser determinada estaticamente!
- Mas como as ativações são sempre aninhadas, podemos manter nossa *posição* na árvore de ativação usando uma *pilha*
- Usando uma pilha podemos facilmente ter procedimentos com mais de uma ativação ao mesmo tempo (funções recursivas)

# Pilha de Ativações - Exemplo

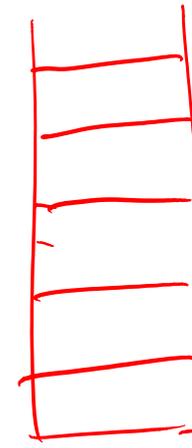
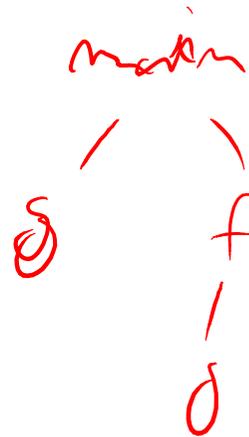
---

- Vamos desenhar a árvore de ativações para o programa TINY abaixo:

```
procedure g()  
  x := 1  
end;
```

```
procedure f()  
  g()  
end;
```

```
var x: int;  
g();  
f();  
write x
```



# Registro de Ativação

---

- A informação armazenada na pilha para gerenciar uma ativação de um procedimento se chama *registro de ativação* (AR) ou *quadro* (frame)
- O registro de ativação de um procedimento  $g$  que foi chamado por um procedimento  $f$  terá informação para:
  - Completar a execução de  $g$
  - Retomar a execução de  $f$  no ponto logo após a <sup>h</sup>camada de  $g$

# Registro de ativação x86 *cdecl* → convenção de chamada C

---

- Argumentos, de trás para frente → código de chamada
- Endereço da instrução seguinte à chamada da função → instrução *call*
- Ponteiro para o registro de ativação do chamador – **o frame pointer (EBP) aponta para cá** → endereço da função chamada
- Variáveis locais
- Espaço para valores temporários e para guardar registradores entre chamadas

# Registro de Ativação – exemplo

- Vamos desenhar o registro de ativação do programa C abaixo para a ativação da função g:

```

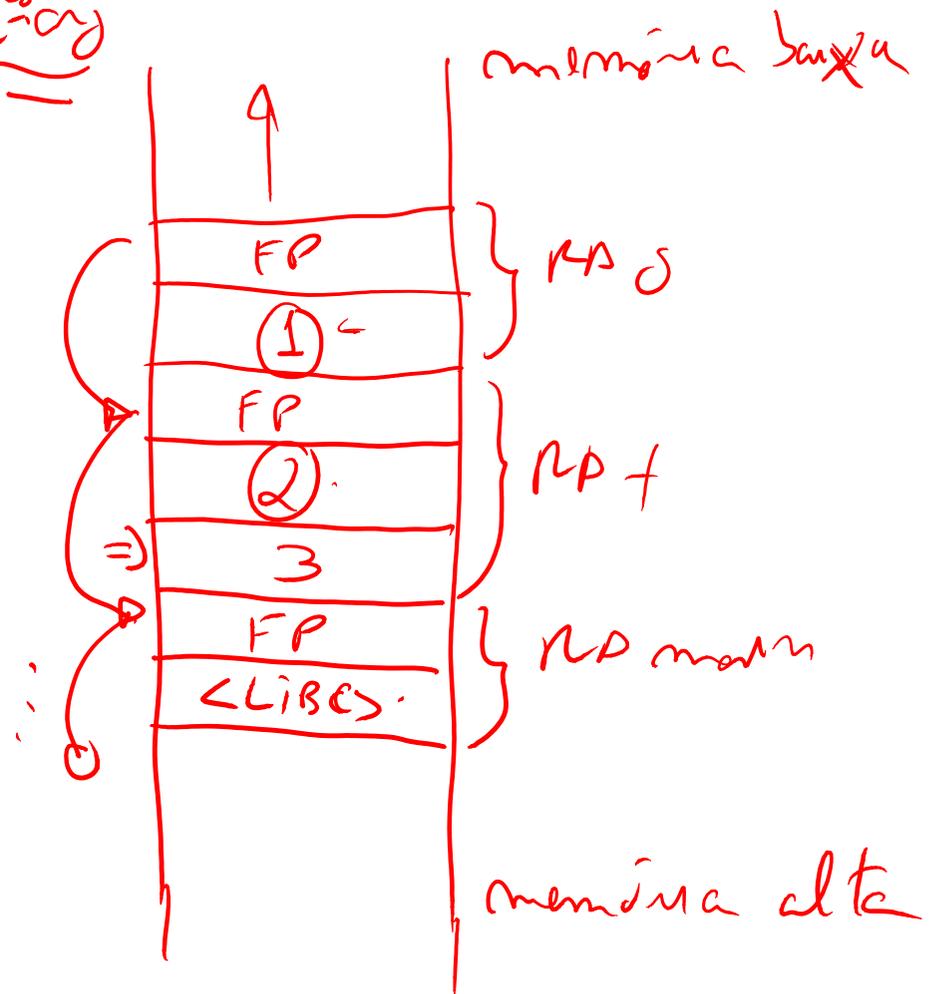
static int g() {
    return 1;
}

static int f(int x) {
    if(x < 2)
        return g();
    else
        return x * f(x-1);
}

int main() {
    return f(3);
}

```

$EBP + 8$  *link*  
 $EBP$  *link*  
 $EBP - 4$   $x = 24$   
 $EBP - 8$   $x = 20$   
 $EBP - 12$   $x = 16$   
 $EBP - 16$   $x = 12$   
 $EBP - 20$   $x = 8$   
 $EBP - 24$   $x = 4$   
 $EBP - 28$   $x$



# Registro de Ativação

---

- Na convenção de chamada x86 cdecl, o valor de retorno da função é colocado em um registrador *→ EAX → se está usando EAX tem a func*  
*valor de f do chama uma função*
- Mas outras arquiteturas podem ter registros de ativação diferentes; em x64, por exemplo, vários argumentos são passados em registradores e não na pilha; a quantidade varia em Windows e Linux  
*4 8*
- O compilador também é livre para ter o seu próprio layout registro de ativação e convenção de chamada, especialmente para procedimentos que não serão “exportados”; bons compiladores tentam usar a pilha o mínimo possível
- Guardar o endereço de retorno na mesma pilha onde estão as variáveis é a origem de muitas falhas de segurança!

# Variáveis Globais

---

- As variáveis globais precisam ser visíveis em todo o programa, e seu alcance é toda a execução do mesmo
- Não faz sentido armazená-las em um registro de ativação
- Elas possuem um endereço fixo no espaço de memória do programa
- O endereço real da global na memória vai ser determinado no momento da carga do programa, pelo *loader* do sistema operacional

# Alocação Dinâmica

---

- Existem valores cujo alcance pode ser maior do que o das variáveis que possuem *ponteiros* para eles:

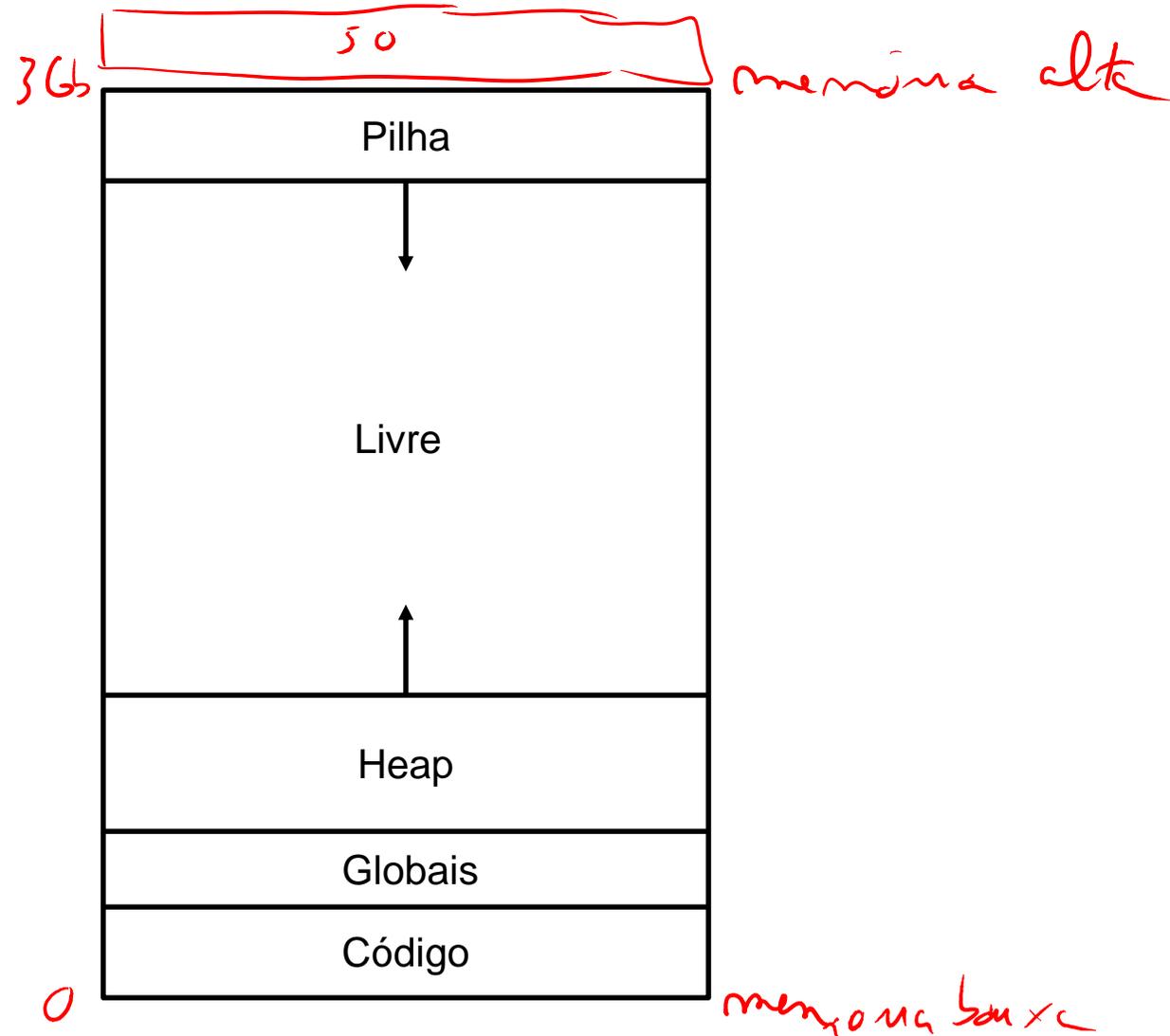
```
static int* foo() {  
    int *foos = (int*)malloc(10 * sizeof(int));  
    return foos;  
}
```

```
Foo foo() {  
    return new Foo();  
}
```

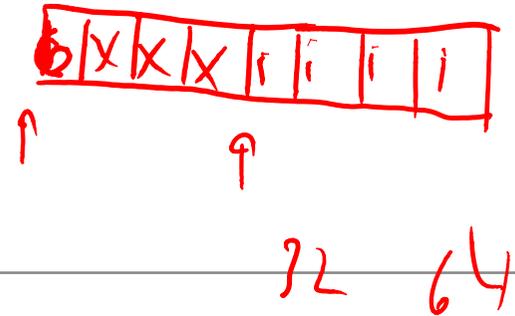
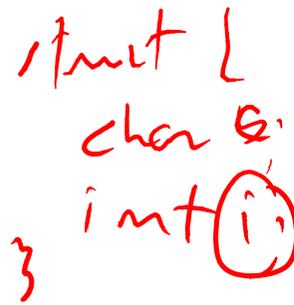
- O vetor e o objeto alocados dentro da função e do método *foo* precisam sobreviver ao registro de ativação da chamada a *foo*
- Esses valores não são armazenados na pilha, mas ficam em outra área da memória chamada heap
- A recuperação da memória no heap depois que o alcance dos valores termina pode ser *manual* (como em C, usando *free*), ou *automática* (como em Java, usando um coletor de lixo ou contagem de referências)

# Layout da memória

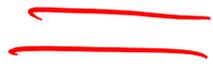
---



# Alinhamento



- A memória de um computador moderno pode ser dividida em blocos de 4 ou 8 bytes, a depender do tamanho da *palavra* do processador (32 ou 64 bits), mas os endereços de memória são contados em *bytes*
- Muitas máquinas ou não podem acessar endereços que não são *alinhados* com o início desses blocos, ou pagam um preço em desempenho nesses acessos → x86
- É responsabilidade do compilador evitar acessos não-alinhados, em geral garantindo que os endereços das variáveis respeitam o alinhamento
- Algumas plataformas podem ter regras de alinhamento mais exóticas: em Mac OS X, o local no AR onde o endereço de retorno é armazenado tem que ser alinhado a blocos de 16 bytes



# Máquinas de Pilha

---

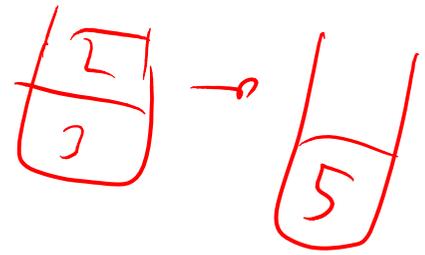
- Uma máquina de pilha é um tipo de processador em que todos os valores temporários são armazenados em uma pilha
  - Não são usados registradores
- Toda operação em uma máquina de pilha desempilha seus operandos, faz a operação e empilha o resultado
- Instruções também podem empilhar valores constantes, 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

push c  $\rightarrow$   $\begin{cases} \text{push ac} \\ \text{ac} \leftarrow c \end{cases}$

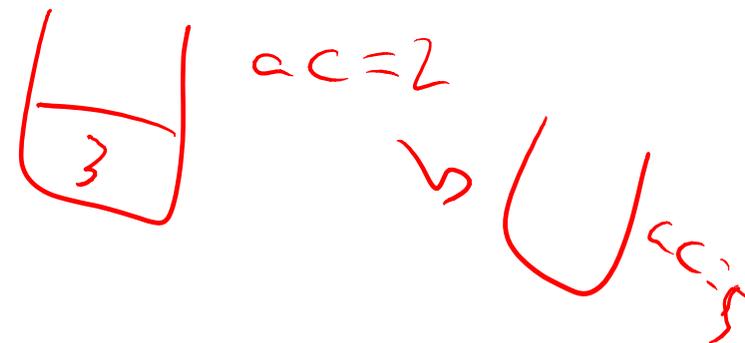
push ebx  
mov ebx, c



The diagram shows a stack with two cells, the top one containing the number 2 and the bottom one containing 3. An arrow points from this stack to a single register containing the number 5.

add d  $\rightarrow$   $\begin{cases} \text{ac} \leftarrow \text{pop} + \text{ac} \\ \text{add ebx}[\text{esp}] \\ \text{add esp}, 4 \end{cases}$

ac=2

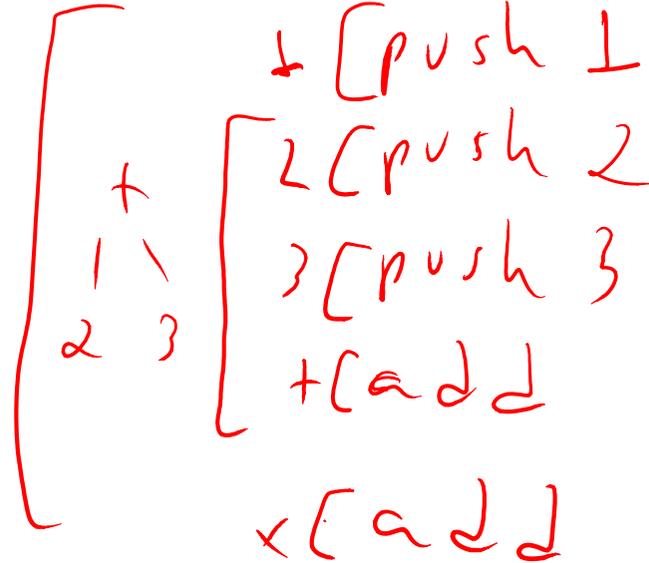
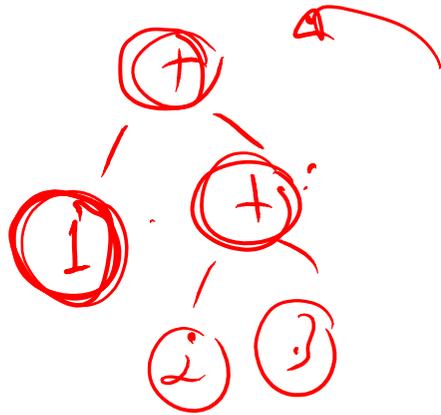


The diagram shows a stack with one cell containing the number 3. An arrow points from this stack to a register containing the number 5.

# Compilando expressões

---

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



→ pós-ordem!

# 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 0  
1
- Nossa máquina de pilha terá 13 instruções: getglobal, putglobal, iload, istore, iadd, isub, imul, idiv, invoke, return, if\_icmpeq, if\_icmplt, goto, read, write scanf printf
- Os nomes são para lembrar as instruções da JVM, que vamos usar em MiniJava

# 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  
↳  $EBP + (?)$
- 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  
↳ reservar espaço para locais no prólogo