

Compiladores – Ambiente de Execução

Fabio Mascarenhas – 2015.2

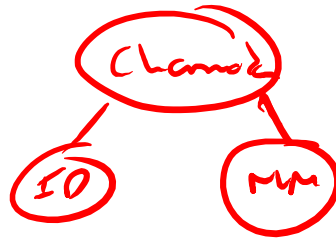
<http://www.dcc.ufrj.br/~fabiom/comp>

O Back-end

- Até agora vimos as fases do *front-end* do compilador:

- Análise Léxica - *IO '(' fun ')' '!' ;*

- Análise Sintática -



- Análise Semântica -

IO tem 2 parâmetros de tipo int ou real

- 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 uma versão simplificada da linguagem de máquina x86, para o compilador TINY

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
tempo
- 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 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());  
var c2 = cont(1);  
console.log(c2());  
console.log(c1());  
console.log(n);
```

2

3

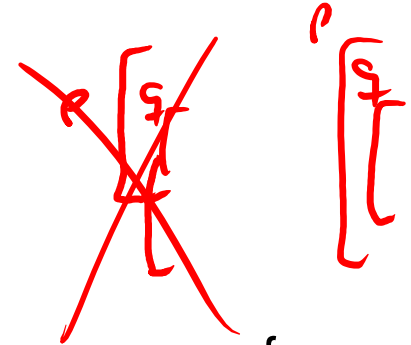
2

4

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*



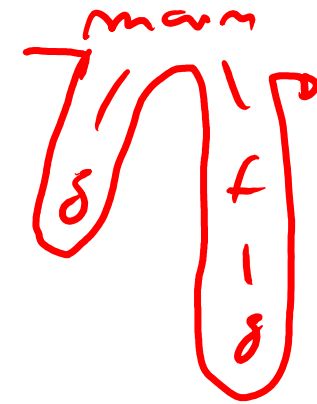
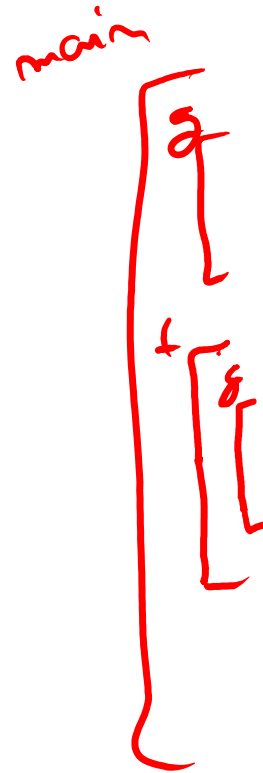
Árvore de Ativações - Exemplo

- 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
```



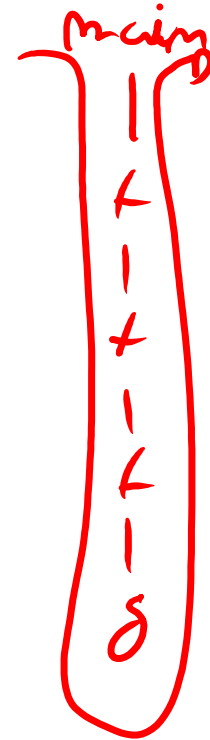
L: começo
2: terminou

```
main L  
g L  
g 2  
f L  
g L  
g 2  
f 2  
main 2
```


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
```



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

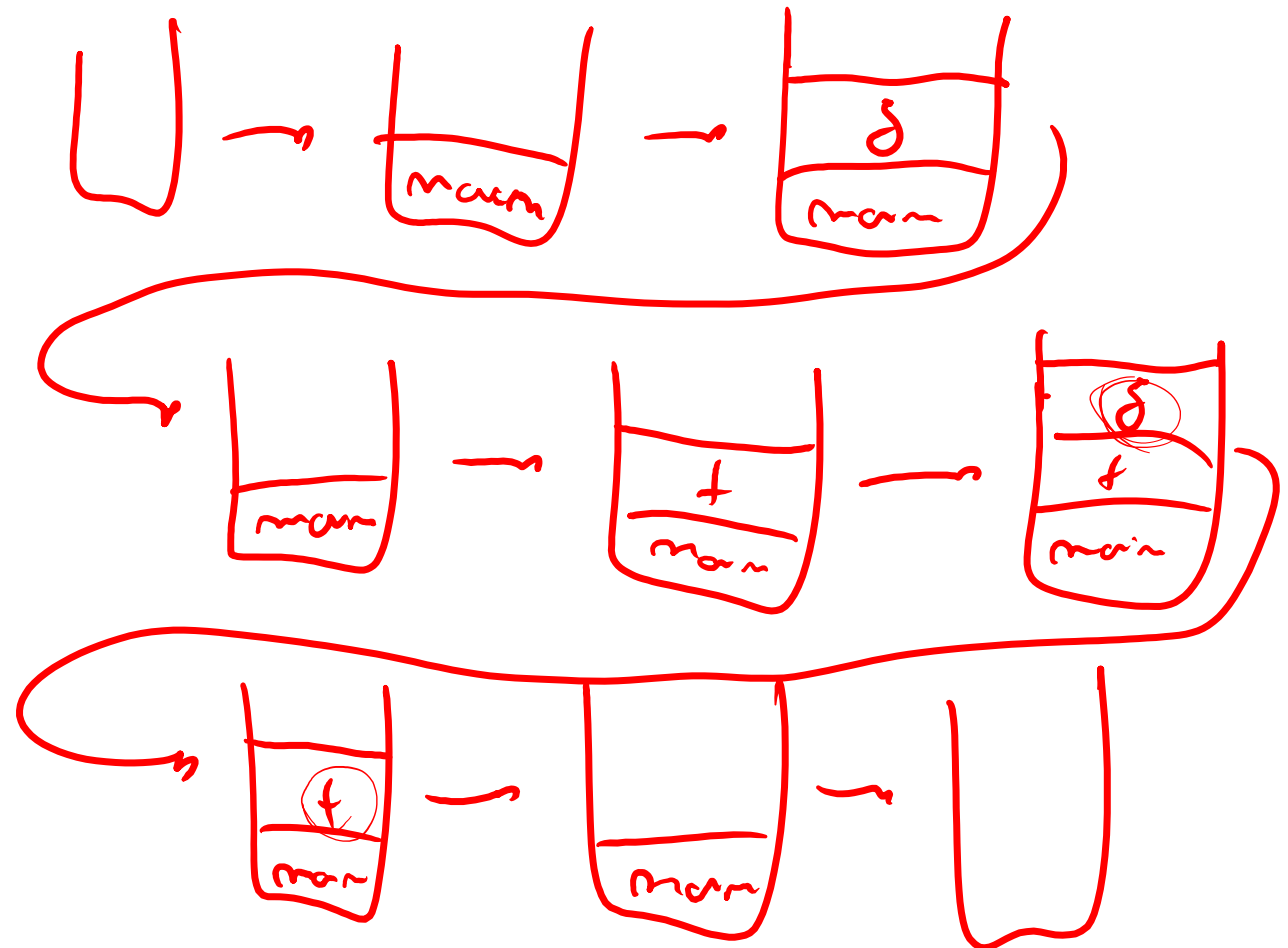
- Vamos desenhar a pilha de ativações para o programa TINY abaixo:

exclusão

```
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 — variáveis locais e argumentos
 - Retomar a execução de f no ponto logo após a chamada de g
↳ endereço de retorno

chamador (caller)

Registro de ativação x86 cdecl

(language C)

- Argumentos, de trás para frente
- Endereço da instrução seguinte à chamada da função

endereço de retorno
call

- Ponteiro para o registro de ativação do chamador – o **frame pointer (EBP)** aponta para cá

- Variáveis locais

- Espaço para valores temporários e para guardar registradores entre chamadas

chamado (callee)

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);  
}
```

