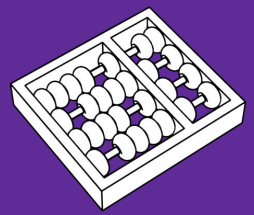
Procedimentos

Geração de Código

Hervé Yviquel
herve@ic.unicamp.br

Universidade Estadual de Campinas (Unicamp)
Instituto de Computação (IC)
Laboratório de Sistemas de Computação (LSC)

MC921 • Projeto e Construção de Compiladores • 2023 S2



Aula Anterior

Resumo

- Representação Intermediária
- Tipos de IRs
- IRs de Compiladores
- Exemplo de IR

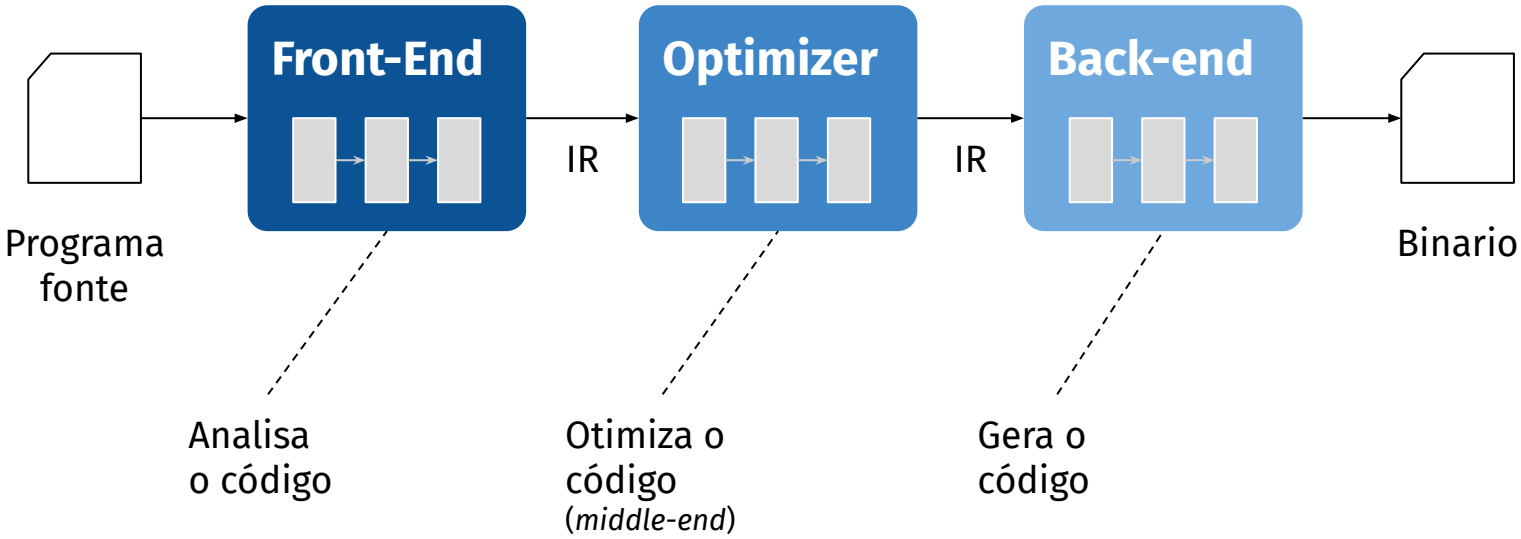
Aula de Hoje

Plano

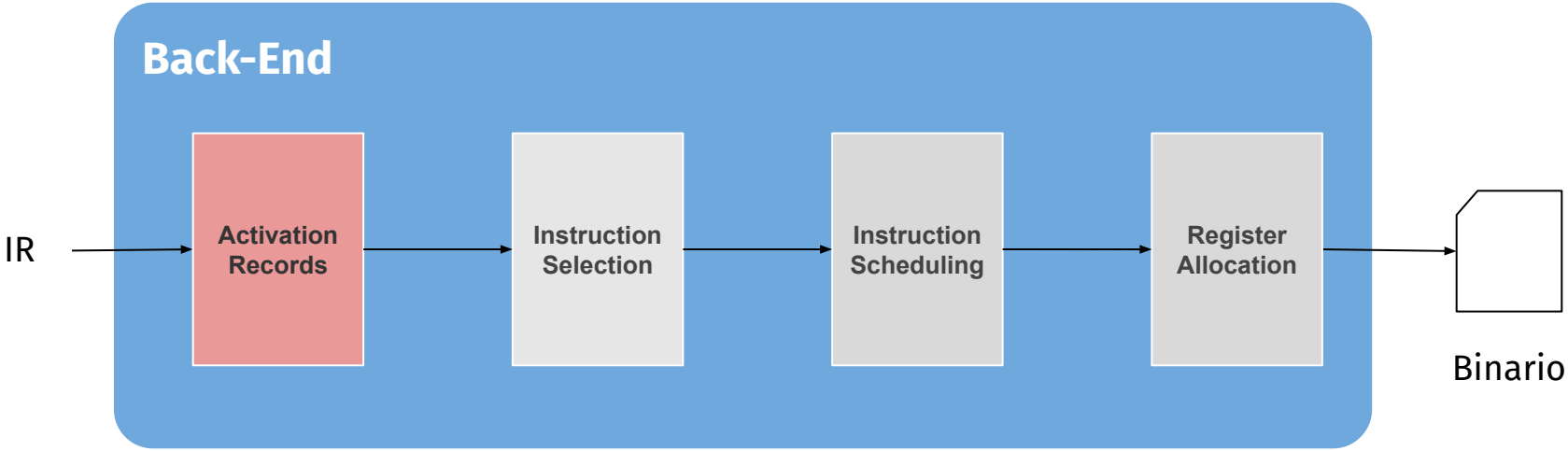
- Introdução
- Ativações
- Registro de Ativação
- Registradores

Introdução

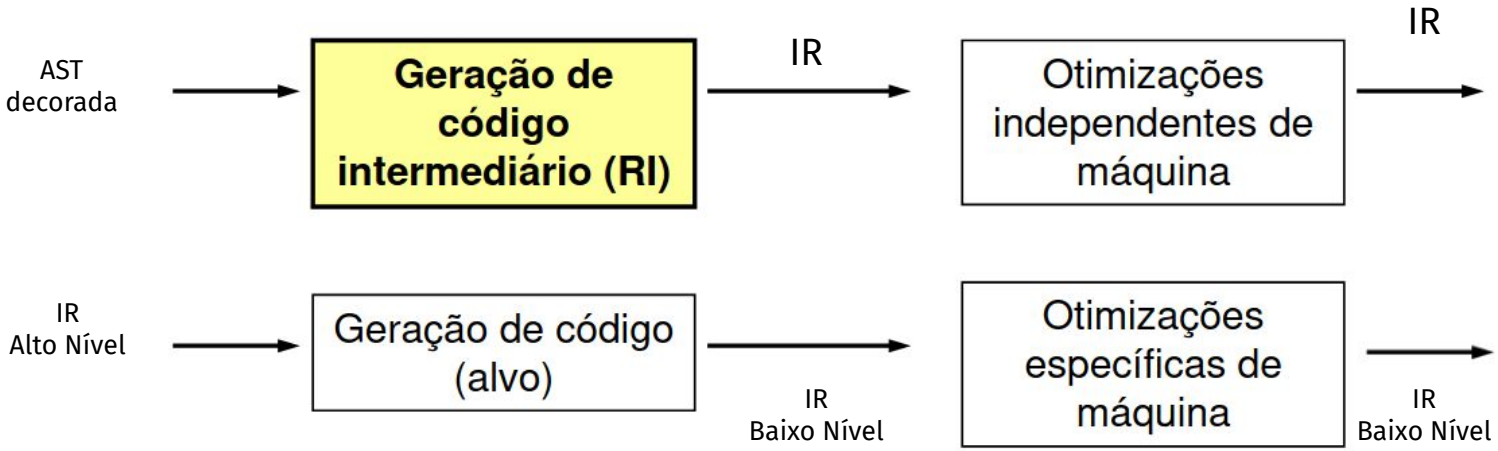
Estrutura Moderna do Compilador



Back-End do Compilador



Geração de Código



Organização da Memória

- Antes de tratar da geração de código em si, 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!

Gerenciamento de recursos em tempo de execução

- Terminologia importante
  - **Estática:** acontece em tempo de compilação
  - **Dinâmica:** acontece em tempo de execução
- É importante deixar claro o que acontece estaticamente e o que acontece dinamicamente
- Eles interagem:
  - Estaticamente, o compilador gera código que organiza dinamicamente a computação
    - por exemplo, garbage collector, invocação de procedimentos/métodos, alocação de memória heap, etc.

Ativações

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 dinâmico (*dynamic scope*), enquanto escopo é estático (*static* or *lexical scope*)

Dynamic Scope

- Escopo dinâmico significa que as pesquisas de variáveis ocorrem no escopo onde uma função é chamada
  - não onde ela está definida
  - usado, por exemplo, em bash, LaTeX, Perl, etc
- Mas escopo estático é preferido em geral

```
$ x=3
$ func1 () { echo "in func1: $x"; }
$ func2 () { local x=9; func1; }
$ func2
in func1: 9
$ func1
in func1: 3
```

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
  - então as ativações durante a execução de um programa formam uma árvore
  - A execução corresponde a um caminho nessa árvore em profundidade

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

procedure f()
  g()
end;

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

Árvores de Ativação

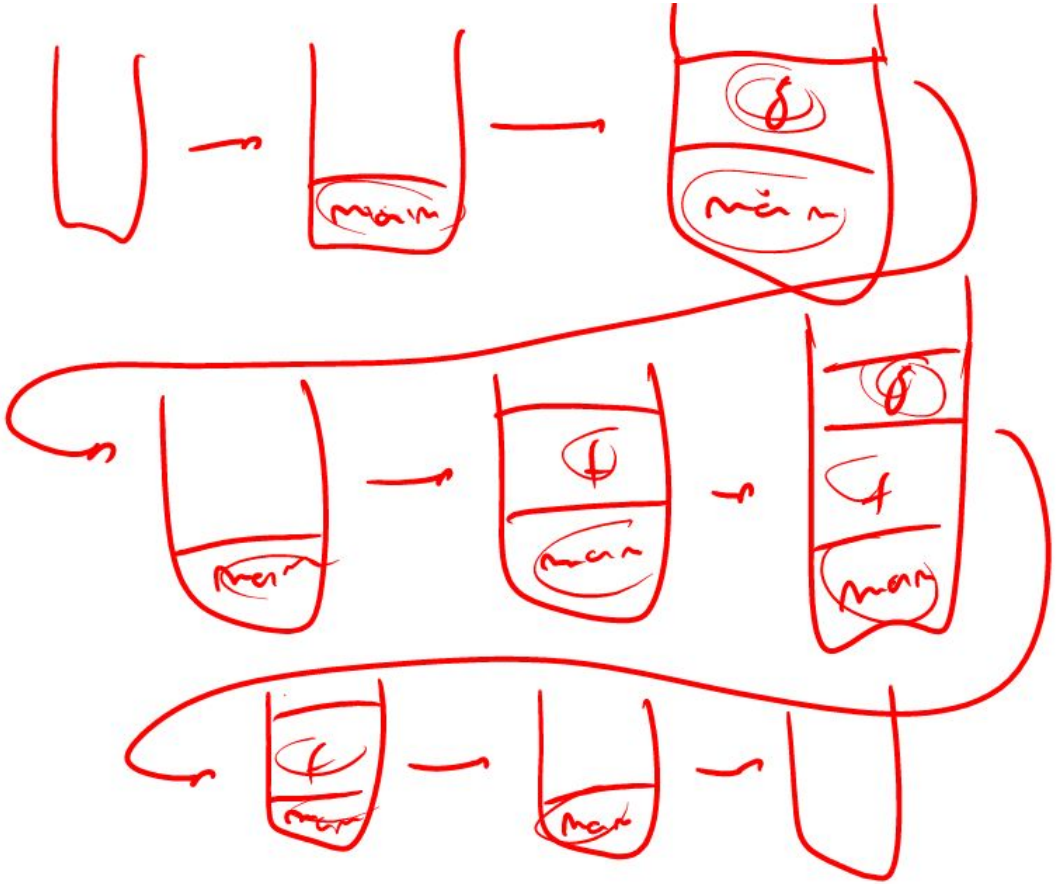
- A árvore de ativação depende da execução do programa, e pode ser diferente a depender da entrada para o programa
  - Ou seja, a árvore de ativação do programa não pode ser determinada estaticamente
- Mas como as ativações são sempre aninhadas, podemos manter nossa posição na árvore de ativação usando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

Pilha de Ativações - Exemplo


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

procedure f()
  g()
end;

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



Registro de Ativação

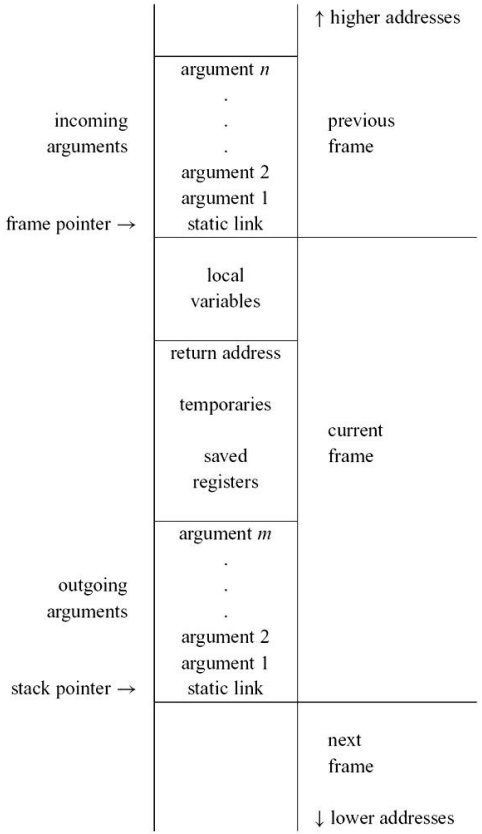
Registro de Ativação

- A informação armazenada na pilha para gerenciar uma ativação de um procedimento se chama registro de ativação (AR) ou quadro (frame)
  - Activation Record ou Stack Frame
- O registro de ativação de um procedimento g que foi chamado por um procedimento f terá informação para:
  - Completar a execução de g – Argumentos e variáveis locais
  - Retomar a execução de f no ponto logo após a chamada de g – Endereço de retorno
- Procedimentos têm variáveis locais
  - Devem ser criadas na chamada da função
  - Sobrevivem até que a função retorne (C, Pascal, Java)
- Recursão
  - Cada instância do procedimento tem seus próprios parâmetros e locais

Registro de Ativação

- As operações push e pop não podem ser feitas individualmente para cada variável
  - Manipula-se conjuntos de variáveis
  - Precisamos ter acesso a todas elas
- Stack Pointer (SP)
  - Todas as posições além do SP são lixo
  - Todas as anteriores estão alocadas
- Registro de Ativação
  - Área na pilha reservada para os dados de uma função
  - parâmetros, locais, endereço de retorno, etc

Pilha



- A pilha normalmente cresce para baixo
- O formato do frame depende
  - Linguagem sendo compilada
  - Características do processador alvo
  - Normalmente a arquitetura do processador determina um layout padrão
  - Funções escritas numa linguagem podem chamar funções de outra linguagem

Frame Pointer

- Suponha que f() chama g(a1, a2, ...,an)
  - f() é conhecida como *caller* (chamador)
  - g() é conhecida como *callee* (chamado)
- Na chamada de g()
  - SP aponta para o primeiro argumento sendo passado a g()
  - g() aloca seu frame subtraindo o tamanho de SP
- O antigo SP se torna o atual FP
- Em algumas arquiteturas o FP é um registrador
  - Seu valor antigo é salvo no frame e restaurado no retorno

Frame Pointer

- FP é útil quando o tamanho dos frames pode variar ou não são contíguos na pilha
- Com frames de tamanho fixo:
  - O FP sempre diferirá de SP por um tamanho conhecido
  - Não é necessário gastar um registrador para isso
- Por que um FP?
  - O tamanho do frame só pode ser calculado muito adiante no processo de compilação
  - É necessário saber o número de temporários e registradores a serem salvos
  - Porém é útil saber os offsets dos parâmetros e locais
  - São alocados primeiro, próximos ao FP
    - Offset conhecido mais cedo

Parâmetros

- Estudos mostram que quase todas chamadas têm no máximo 4 argumentos
- Quase nenhuma tem mais que 6
- Antigamente, passagem era sempre feita na pilha
  - Tráfego de memória desnecessário
- Hoje em dia
  - Primeiros k parâmetros passados em registradores, o resto na pilha
  - K = 4 ou 6 é um valor típico

Registradores

Registradores

- Registradores são unidades de armazenamento internas do processador
  - Tempo de acesso é milhares de vezes mais rápido que a memória
- O bom uso dos registradores é essencial para um bom desempenho do programa
- O número de registradores é limitado
  - Por exemplo, 32
- É comum instruções aritméticas poderem acessar valores somente em registradores
  - RISC

Caller-save e Callee-save

- Muitas funções podem precisar dos registradores ao mesmo tempo
- Suponha que f() chama g() e ambas usam o registrador r
  - Caller-save: É responsabilidade de f salvar e restaurar o registrador r
  - Callee-save: É responsabilidade de g salvar e restaurar o registrador r
- Normalmente não existe uma diferença física entre os registradores
  - Diferenciação entre caller e callee saves é convenção da arquitetura
- Uma boa escolha entre caller e callee save registers pode economizar acessos à memória
- Exemplo: No MIPS
  - $t0-$t9: caller-saves
  - $s0-$s7: callee-saves
- Como sabe em qual alocar?

Caller-save ($t0-$t9)

Aloca x a $t1 pois x não é usado depois da chamada de g()

```
f(…) {
  x = y + 1
  $t1 = x

  g (a1, a2,…)

}
```

x not used after

Alocador remove moves desnecessários

```
   f( ….) {
$t1  x = y + 1
     $t1 = x

     g (a1,a2…)
   }
```

Caller-save ($t0-$t9)

g() não precisa salvar pois o caller salva se precisar!

```
g (a1,a2,…) {

    $t1 = 2 * $t2

}
```

Callee-save ($s0-$s7)

No início x é alocado
a um registrador callee-save ($s1)

```
f ( ….) {
  x = y + 1
  $s1 = x

  g (a1,a2…)
  x = $s1

  y = 2 * x
}
```

Depois o alocador remove moves
desnecessários

```
    f ( ….) {
$s1  x = y + 1
     $s1 = x
     g (a1, a2, …)
     x = $s1
     y = 2 * x   $s1
}
```

Callee-save ($s0-$s7)

g() somente salva se precisar usar $s1 dentro!

```
g (a1,a2,…) {

  $s1 not used here

}
```

```
g (a1,a2,…) {
  stack = $s1

  $s1 = 2 * k

  $s1 = stack
}
```

Endereço de Retorno

- Antigamente era colocado na pilha no momento da chamada da função
- Atualmente é mais comum ser mantido em registrador
  - No MIPS: ra ou $31
- Procedimentos que não são folhas devem salvá-lo na pilha

Variáveis na Pilha

- Razões para alocar uma variável na pilha:
  - Passagem por referência (& em C)
  - Variáveis acessadas por funções aninhadas
  - Valor muito grande para caber em um registrador
- Alguns compiladores podem dividir acesso entre registrador e memória
  - Registrador ocupado pela variável se torna necessário para outro propósito
  - Spill: há variáveis locais e temporários demais para caber todos em registradores, alguns são forçadamente colocados na pilha

Resumo

- Ativações
- Registro de Ativação
- Registradores

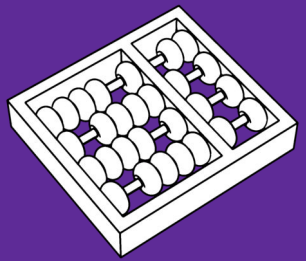
Leitura Recomendada

- Capítulo 8 do livro do Aho
- Capítulo 6 do livro do Cooper

Próxima Aula

- Seleção de Instruções

Obrigado!
Merci!



UNICAMP

Pallete

Bubbles

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

DRACULA

Tables

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Emojis