Geração de Código

Thierson Couto Rosa

Instituto de Informática
Universidade Federal de Goiás

9 de janeiro de 2024

Geração de Código

Observações em Relação a um Compilador de Produção

- A fase de geração de código recebe como entrada a representação intermediária (RI) do programa-fonte juntamente com a tabela de símbolos e produz um código-objeto semanticamente equivalente ao programa fonte.
- Para gerar código objeto eficiente, os compiladores incluem uma fase de otimização antes da geração de código. Nesta fase, pode ser necessário converter de uma RI em outra, e realizar diversas otimizações ainda a nível de RI.
- um gerador de código é composto por três tarefas principais: seleção de instruções, alocação e atribuição de registradores e escalonamento de instrução.
- A fase de geração de código é complexa. É muito dependente da RI utilizada e da arquitetura da máquina alvo.

Geração de Código para a Linguagem Cafezinho

Observações em Relação a um Projeto Didático

- A disciplina tem duração de um semestre (na prática, quatro meses)
- O objetivo é gerar um compilador de caráter didático que possibilite mostrar ao aluno as fases de um compilador completo, sem incluir a otimização de código: análise léxico-sintática, análise semântica e geração da RI, tradução da RI em uma linguagem assembly e execução do código gerado em um simulador.
- Portanto, devido ao caráter puramente didático e ao prazo para se obter um código objeto, várias simplificações foram feitas e praticamente omitimos as principais técnicas de geração de código necessárias em um compilador de produção (ver Capítulo 8 do Dragon Book para uma visão detalhada desta fase.)

Geração de Código para a Linguagem Cafezinho

Simplificações Cosideradas

- Utilizamos a árvore sintática gerada na fase de análise como RI.
- Consideramos que o código objeto será executado em uma máquina de pilha, conforme será explicado a seguir.
- Utilizaremos uma arquitetura MIPS para simular a máquina de pilha. Isso permite uma grande simplificação na geração de código para um projeto didático, mas é uma abordagem totalmente sem sentido para um compilador de produção, pois estamos forçando uma máquina baseada em registradores a usar excessivamente a memória.
- Escolhemos uma arquitetura MIPS devido à simplicidade de suas instruções. Os alunos precisam aprender apenas um sub-conjunto pequenos do conjunto reduzido de instruções do MIPS.

Máquina de pilha

Stack machine

- Uma máquina de Pilha é um modelo de máquina cuja única memória disponível funciona como uma pilha.
- De modo genérico cada instrução de máquina pode ser vista como uma expressão do tipo $r = f(a_1, a_2, \ldots a_n)$.
- r é o resultado da instrução, f representa uma operação e a_i são os argumentos da operação f.
- Em uma máquina de pilha, os n operandos se encontram na pilha.
- Para executar a instrução a máquina desempilha pares de parâmetros no topo da pilha, aplica a operação f sobre os parâmetros desempilhados e empilha o resultado, até que o resultado final r fique no topo da pilha.

Exemplo

- Suponha a operação de soma de dois números: $3 + 4$.
- Supondo que no topo da pilha estejam os dois operandos: $4, 3, X$, onde o elemento do topo é 4 e X representa o restante dos elementos na pilha.
- A soma remove os dois elementos do topo, deixando a pilha com a seguinte configuração: X.
- Os valores desempilhados são somados e o resultado é empilhado: $7, X$.
- importante notar a seguinte propriedade da máquina de pilha: o dado X abaixo dos parâmetros, que existia antes da operação, permanece após a operação.

Programação da máquina de pilha

Característica

- As instruções da máquina de pilha não contêm endereços explícitos, pois os operandos sempre se encontram nas últimas posições no topo da pilha.
- Em contraste, uma máquina com registradores, necessita indicar onde estão os operandos:
  - add r1,r2,r3
- Programas em máquinas de pilha tendem a ser mais compactos, pois o número de instruções é menor. Isto é uma das razões porque o *Bytecode* corresponde a uma máquina de pilha.

Programação da máquina de pilha

Característica

- As máquinas de registradores por outro lado são bem mais rápidas, pois o acesso aos registradores é muito mais rápido do que o acesso à memória.
- Uma máquina intermediária é uma máquina de pilha com n registradores, a qual se aproxima mais das arquiteturas atuais.
- Um caso particular é uma máquina de pilha com um registrador (acumulador).

Máquina de pilha com um registrador

Acumulador

- Em uma máquina de pilha pura, a instrução de soma implica em três acesos à memória: 2 pop e 1 push.
- Em uma máquina de pilha com o acumulador, um operando fica no acumulador e outro fica no topo da pilha.
- Para somar dois números, o valor no topo da pilha é desempilhado e somado com o valor do acumulador.
- O resultado fica armazenando no acumulador.

Gerando código para expressões em máquinas de pilha

Acumulador

- Considere a expressão $op(e_1, e_2, \ldots, e_n)$, onde cada $e_i, 1 \leq i \leq n$ é uma expressão na linguagem de alto nível.
- Para cada e_i, $(1 \leq i < n)$ compute e_i. O resultado de e_i fica no acumulador. Empilhe o valor do acumulador.
- Para a última expressão e_n compute a expressão e deixe o resultado no acumulador (o valor não é empilhado)
- Em seguida repita as operações seguintes $n - 1$ vezes:
  - Desempilhe o elemento do topo e aplique o operador op sobre esse elemento e o valor que está no acumulador.
  - O resultado fica no acumulador.
- **Invariante:** Após avaliar $op(e_1, e_2, \ldots, e_n)$, a pilha permanece inalterada, contendo o mesmo dado que continha antes da avaliação da expressão - a avaliação da expressão preserva a pilha.

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha
$acc \leftarrow 3$	3	?

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha
$acc \leftarrow 3$	3	?
push acc	3	3, ?

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha
$acc \leftarrow 3$	3	?
push acc	3	3, ?
$acc \leftarrow 2$	2	3,?

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha
$acc \leftarrow 3$	3	?
push acc	3	3, ?
$acc \leftarrow 2$	2	3,?
push acc	2	2,3, ?

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha
$acc \leftarrow 3$	3	?
push acc	3	3, ?
$acc \leftarrow 2$	2	3,?
push acc	2	2,3, ?
$acc \leftarrow 4$	4	2,3,?

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha
$acc \leftarrow 3$	3	?
push acc	3	3, ?
$acc \leftarrow 2$	2	3,?
push acc	2	2,3, ?
$acc \leftarrow 4$	4	2,3,?
$acc \leftarrow acc + top$	6	2, 3,?

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha
$acc \leftarrow 3$	3	?
push acc	3	3, ?
$acc \leftarrow 2$	2	3,?
push acc	2	2,3, ?
$acc \leftarrow 4$	4	2,3,?
$acc \leftarrow acc + top$	6	2, 3,?
pop	6	3,?

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha
$acc \leftarrow 3$	3	?
push acc	3	3, ?
$acc \leftarrow 2$	2	3,?
push acc	2	2,3, ?
$acc \leftarrow 4$	4	2,3,?
$acc \leftarrow acc + top$	6	2, 3,?
pop	6	3,?
$acc \leftarrow acc + top$	9	3,?

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha
$acc \leftarrow 3$	3	?
push acc	3	3, ?
$acc \leftarrow 2$	2	3,?
push acc	2	2,3, ?
$acc \leftarrow 4$	4	2,3,?
$acc \leftarrow acc + top$	6	2, 3,?
pop	6	3,?
$acc \leftarrow acc + top$	9	3,?
pop	9	?

Exemplo

$3 + (2 + 4)$

Código	Acumulador	Pilha
$acc \leftarrow 3$	3	?
push acc	3	3, ?
$acc \leftarrow 2$	2	3,?
push acc	2	2,3, ?
$acc \leftarrow 4$	4	2,3,?
$acc \leftarrow acc + top$	6	2, 3,?
pop	6	3,?
$acc \leftarrow acc + top$	9	3,?
pop	9	?

Gerando código para expressões em máquinas de pilha

- Geraremos código para uma máquina (virtual) de pilha com um registrador.
- O código resultante deve executar em uma máquina real (MIPS) ou em um simulador de uma máquina real.
- Então, simularemos as instruções de uma máquina de pilha com um registrador, usando instruções e registradores do MIPS.

Simulação em MIPS

- Utilizaremos o registrador $s0 do MIPS como o acumulador.
- A Pilha será mantida em memória.
  - A pilha cresce em direção aos endereços mais baixos.
  - Isso é uma convenção padrão na programação do MIPS.
- O endereço da próxima localização de memória é mantido no registrador $sp.
- $sp aponta para a posição seguinte ao topo da pilha. Portanto, o topo da pilha está na posição $sp+4.

Simulação em MIPS

MIPS

- MIPS - Reduced Instruction Set - arquitetura antiga (década de 80).
- Conjunto reduzido de instruções.
- A maioria das instruções são realizadas com registradores.
- Usa instruções load e store para transferir dados entre registradores e memória.
- Possui 32 registradores de propósito geral.
  - Utilizaremos: $a0, $fp $s0, $s1, $sp, $t1 $v0
  - Ler bibliografia indicada.

Simulação em MIPS

Instruções inicialmente utilizadas

- `lw` reg_1`,` `offset(`reg_2`)`
  - Carrega em reg_1 a palavra de 32 bits que está na posição $reg_2 + offset$ de memória.
- `add` reg_1`,` reg_2`,` reg_3
  - $reg_1 \leftarrow reg_2 + reg_3$
- `sw` reg_1`,` $offset(reg_2)$
  - Armazena a palavra que está no registrador reg_1 no endereço de memória indicado por $reg_2 + offset$
- `addiu` reg_1`,` reg_2, imm
  - $reg_1 \leftarrow reg_2 +$ imm (u - underflow não é verificado)
- `li` reg_1, imm
  - $reg_1 \leftarrow$ imm

Simulação em MIPS

Exemplo

- Código de máquina de pilha para a expressão $7 + 5$ usando MIPS

- $acc \leftarrow 7$
- `push` acc
- $acc \leftarrow 5$
- $acc \leftarrow acc + top$
- `pop`

- `li $s0, 7`
- `sw $s0, 0($sp)`
- `addiu $sp, $sp, -4`
- `li $s0, 5`
- `lw $t1, 4($sp)`
- `addiu $sp, $sp, 4`
- `add $s0, $s0, $t1`

Sistematizando a geração de código

- Apresentamos a seguir algumas técnicas para sistematizar a geração de código para construções de linguagens de alto nível.
- Utilizaremos a máquina de pilha implementada em MIPS para facilitar a geração de código.
- As regras de adotadas para programação em máquina de pilha que vimos anteriormente possibilitam sistematizar e até mesmo automatizar a geração de código, como veremos a seguir.
- Iniciamos com a geração de código para expressões da linguagem de alto nível.

Geração de código para expressões

- Para cada expressão e geramos código MIPS que:
  - Calcula o valor de e e deixar o valor armazenado em \$s0 (nosso acumulador).
  - Preserva \$sp e o conteúdo da pilha.
  - definimos uma função de alto nível $cgenEx(e)$ que gera o código que implementa a expressão e.
- A função $cgenEx(e)$ deve tratar dois tipos distintos de expressões.

Geração de código para expressões

$cgenEx(e)$

- Caso 1: a expressão é uma constante i:
  - $cgenEx(i) =$ `print ("li $s0 i")`
  - Preserva a pilha !

Geração de código para expressões

$cgenEx(e)$

- Caso 2: $cgenEx(e_1 + e_2)$:
- $cgenEx(e_1 + e_2) = \{$
  - $cgenEx(e_1)$ - chamamos $cgenEx$ para gerar código para e_1. Ao ser executado esse código deve deixar o resultado de e_1 em \$s0.

Geração de código para expressões

$cgenEx(e)$

- Caso 2: $cgenEx(e_1 + e_2)$:
- $cgenEx(e_1 + e_2) = \{$
  - $cgenEx(e_1)$ - chamamos $cgenEx$ para gerar código para e_1. Ao ser executado esse código deve deixar o resultado de e_1 em $s0.
  - `print ("sw $s0, 0($sp)"); print ("addiu $sp, $sp -4");` - empilhamos o conteúdo de $s0 ($e_1$) antes de gerar código para e_2.

Geração de código para expressões

$cgenEx(e)$

- Caso 2: $cgenEx(e_1 + e_2)$:
- $cgenEx(e_1 + e_2) = \{$
  - $cgenEx(e_1)$ - chamamos $cgenEx$ para gerar código para e_1. Ao ser executado esse código deve deixar o resultado de e_1 em $s0.
  - `print ("sw $s0, 0($sp)"); print ("addiu $sp, $sp -4");` - empilhamos o conteúdo de $s0 ($e_1$) antes de gerar código para e_2.
  - $gcenEx(e_2)$ - chamamos $cgenEx$ para gerar código para e_2. O código gerado deve colocar o resultado de e_2 em $s0.

Geração de código para expressões

$cgenEx(e)$

- Caso 2: $cgenEx(e_1 + e_2)$:
- $cgenEx(e_1 + e_2) = \{$
  - $cgenEx(e_1)$ - chamamos $cgenEx$ para gerar código para e_1. Ao ser executado esse código deve deixar o resultado de e_1 em $s0.
  - `print ("sw $s0, 0($sp)"); print ("addiu $sp, $sp -4");` - empilhamos o conteúdo de $s0 ($e_1$) antes de gerar código para e_2.
  - $gcenEx(e_2)$ - chamamos $cgenEx$ para gerar código para e_2. O código gerado deve colocar o resultado de e_2 em $s0.
  - `print("lw $t1, 4($sp)");` - copia o valor de e_1 que está no topo da pilha em $t1.

Geração de código para expressões

$cgenEx(e)$

- Caso 2: $cgenEx(e_1 + e_2)$:
- $cgenEx(e_1 + e_2) = \{$
  - $cgenEx(e_1)$ - chamamos $cgenEx$ para gerar código para e_1. Ao ser executado esse código deve deixar o resultado de e_1 em $s0.
  - `print ("sw $s0, 0($sp)"); print ("addiu $sp, $sp -4");` - empilhamos o conteúdo de $s0 ($e_1$) antes de gerar código para e_2.
  - $gcenEx(e_2)$ - chamamos $cgenEx$ para gerar código para e_2. O código gerado deve colocar o resultado de e_2 em $s0.
  - `print("lw $t1, 4($sp)");` - copia o valor de e_1 que está no topo da pilha em $t1.
  - `print("addiu $sp, $sp, 4");` - pop - preserva a pilha.

Geração de código para expressões

$cgenEx(e)$

- Caso 2: $cgenEx(e_1 + e_2)$:
- $cgenEx(e_1 + e_2) = \{$
  - $cgenEx(e_1)$ - chamamos $cgenEx$ para gerar código para e_1. Ao ser executado esse código deve deixar o resultado de e_1 em $s0.
  - `print ("sw $s0, 0($sp)"); print ("addiu $sp, $sp -4");` - empilhamos o conteúdo de $s0 ($e_1$) antes de gerar código para e_2.
  - $gcenEx(e_2)$ - chamamos $cgenEx$ para gerar código para e_2. O código gerado deve colocar o resultado de e_2 em $s0.
  - `print("lw $t1, 4($sp)");` - copia o valor de e_1 que está no topo da pilha em $t1.
  - `print("addiu $sp, $sp, 4");` - pop - preserva a pilha.
  - `print("add $s0, $s0, $t1");` - soma o valor de e_1 com o valor de e_2 que está em $s0 e deixa o resultado em $s0.

Geração de código para expressões

Formato geral

- O algoritmo para $cgenEx(e)$ dado anteriormente pode servir de formato para geração de outras expressões aritméticas como subtração, multiplicação e divisão.
- O que é necessário é substituir o comando `print("add $0, $t1, $s0'')` por um conjunto de comandos que gere código para a operação correspondente.
- Pode ser utilizado em caminhamento na árvore abstrata gerada como representação intermediária.

Instruções de Desvio Condicional

- *Branch on Equal* - desvia o controle para "label" se $reg_1 = reg_2$: `beq` reg_1, reg_2, `label`
- *Branch on Not Equal* - desvia o controle para "label" se $reg_1 \neq reg_2$: `bne` reg_1, reg_2, `label`
- *Branch on Less Than* - desvia o controle para "label" se $reg_1 < reg_2$: `blt` reg_1, reg_2, `label`
- *Branch on Greater Than* - desvia o controle para "label" se $reg_1 > reg_2$: `bgt` reg_1, reg_2, `label`
- *Branch on Less Than or Equal* - desvia o controle para "label" se $reg_1 \leq reg_2$: `ble` reg_1, reg_2, `label`
- *Branch on Greater Than or Equal* - desvia o controle para "label" se $reg_1 \geq reg_2$: bge reg_1, reg_2, `label`

Instruções de Desvio Incondicional

- *Branch* - vá para o "label":

```
b label
```

- *Jump* - vá para o "label":

```
j label
```

Geração de código para comandos com desvios

Geração de Código para IF-THEN-ELSE

$cgenIf(e_1, e_2, c_{then}, c_{else})$ {

- Gera código para e_1: cgenEx(e_1);
- Imprime instruções para empilhar e_1: `print("sw $s0, 0($sp)");` `print("addiu $sp, $sp, −4");`
- Gera código para e_2: cgenEx(e_2);
- Imprime instruções para desempilhar o valor de e_1 em $t1: `print ("lw $t1, 4($sp)"); print("addiu $sp, $sp, 4");`
- `Imprime comparação de` e_1 `com` e_2 `e desvio para o código do then:` `print("beq $s0, $t1, then"); print ("else:");`
- Gera código para os comados após *else*: cgenCmd(c_{else})
- Imprime desvio para o fim do if: `print("b End_if");`
- Imprime o rótulo para o início dos comandos correspondentes a *then*: `print("then:");`
- Gera código para os comando do *then*: cgenCmd(c_{then});
- Imprime rótulo para o fim do if: `print("End_if:");`

}

Geração de código para comandos com desvios

Geração de Código para WHILE

$cgenWhile(e_1, e_2, c_{while})$ {

- Imprime rotulo para o inicio do while: `print ("while:");`
- Gera código para e_1: cgenEx(e_1);
- Imprime instruções para empilhar e_1: `print("sw $s0, 0($sp)");` `print("addiu $sp, $sp, −4");`
- Gera código para e_2: cgenEx(e_2);
- Imprime instruções para desempilhar o valor de e_1 em $t1: `print ("lw $t1, 4($sp)"); print("addiu $sp, $sp, 4");`
- `Imprime comparação de` e_1 `com` e_2 `e desvio para o fim do comando while: print("bne $s0, $t1, fim_while");`
- Gera código para os comados dentro do while: cgenCmd(c_{while})
- Imprime desvio para o início do while: `print("b while");`
- Imprime o rótulo para o do comando while : `print("fim_while:");`

}

Chamada de função

Instruções e registradores adicionais

- A instrução `jal label` (*jump and link*) é utilizada para fazer chamada à função cujo código tem início a partir do endereço com rótulo *label*.
  - A instrução `jal` salva no registrador $ra - *return address* o endereço da instrução que a segue no código (i. e. o endereço de retorno - o *link* com o código chamador). Em seguida, realiza um desvio para `label`.
- A instrução `jr $ra` é executada ao final da função chamada para retornar o controle da execução para instrução imediatamente após a instrução `jal` executada na chamada.
- O registrador $fp (*frame pointer*) é utilizado para marcar o início de um frame. Seu uso é opcional na geração de código para funções.

Sequência de chamada

Códigos de controle para a execução de uma função

- Para que uma chamada de função possa ocorrer, é necessário que o compilador gere códigos para empilhar os elementos do *frame* da função chamada.
- As instruções que formam esse código são denominadas *sequência de chamada*.
- Parte da sequência de chamada fica na função ou programa que chama a função a ser executada.
- Essa parte da sequência de chamada é denominada *sequência de chamada do código chamador*.
- A outra parte da sequência de chamda fica na função chamada e é denominada. *sequência de chamada do código chamado*.

Sequência de chamada de função

Sugestão de sequência de chamada no código chamador

- Seja uma chamada a uma função f passando n expressões como argumentos de entrada: $f(e_1, e_2, \ldots, e_n)$.
- Sugerimos a seguinte sequência para implementar a chamada da função f:
- O código chamador:
  - empilha o valor atual do frame pointer ($fp).
  - gera código para cada expressão correspondente a cada um dos n parâmetros e empilha o valor resultante. Expressões são geradas na ordem inversa.

Sequência de chamada de função

Sequência de chamada do código chamador no MIPS

- $cgenf(f(e_1, e_2, \ldots, e_n))$

```
print("sw $fp, 0($sp)") # empilha o valor de $fp
print("addiu $sp, $sp, -4")
#gera código para as expressões na ordem inversa e empilha
os resultados
```

$cgen(e_n)$

```
print("sw $s0, 0($sp)")
print("addiu $sp, $sp, -4")
```

$\vdots$

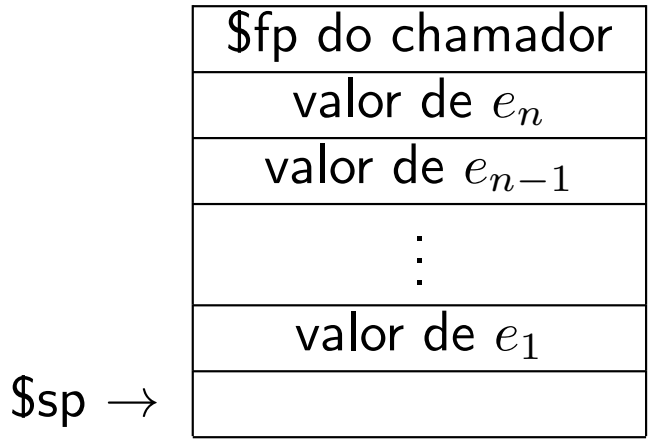
$cgen(e_1)$

```
print("sw $s0, 0($sp)")
print("addiu $sp, $sp, -4")
#gera chamada à função
print("jal f_entry")
```

Sequência de chamada de função

Estrutura parcial do registro de ativação

A figura abaixo mostra como fica o registro de ativação após a execução da sequência de instruções efetuadas pelo código chamador



Sequência de chamada de função

Sugestão de sequência de chamada no código chamado.

- Convencionaremos que o valor retornado por uma função sempre fica no acumulador ($s0).
- O código chamado:
  - Empilha o valor de $ra.
  - Atualiza o $fp para apontar para o topo da pilha (para o valor de $ra empilhado).
  - Aloca espaço na pilha para suas variáveis locais (se existirem).
  - Gera código para seus comandos.

Sequência de chamada de função

Sequência de chamada do código chamado em MIPS

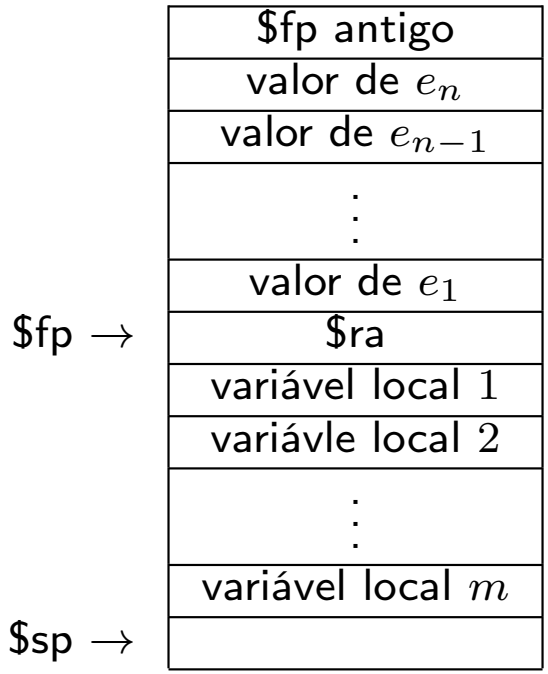
`print("move $fp, $sp")` - $fp passa apontar a posição após a última expressão - `move` é uma pseudo instrução do montandor do MIPS que permite copiar o valor de um registrador ($sp) para outro registrador ($fp)

```
print("sw $ra, 0($sp)") # empilha $ra
print("addiu $sp, $sp, −4")
# aloca espaço na pilha para as m variáveis locais de f,
m ≥ 0:
print("addiu $sp, $sp,", −4 × m) #$sp aponta para a
próxima posição após a m−ésima variável.
#gera código para os comandos de f
```

Sequência de chamada de função

Estrutura do registro de ativação

A figura abaixo mostra como fica o registro de ativação após a execução da sequência de instruções efetuadas pelo código chamado



Registro de ativação da função

Características

- A forma como foi projetado, o RA:
  - Permite fácil acesso aos argumentos de chamada a partir do $fp
  - Permite fácil acesso às variáveis locais, também a través do $fp
  - Permite fácil acceso ao valor de $ra a ser restaurado, também através do $fp.

Código para retorno de procedimento

Sequência no código chamado

- O valor de $ra no registro de ativação deve ser restaurado.
- O valor do frame pointer que existia no momento chamada deve ser restaurado.
- O stack pointer deve voltar a apontar para o endereço que apontava antes da chamada.

```
print("lw $ra, 0($fp")# restaura o valor do $ra
print("move $sp, $fp") # Desempilha variáveis locais.
print("addiu $sp, $sp", z) # onde z = (n + 1) × 4 ( n é o
núm.  de parâmetros)
print("lw $fp, 0($sp)") #restaura o valor de $fp do
chamador
print("jr $ra") #executa o retorno
```

Acesso a variáveis não locais

Variáveis globais ao programa como um todo

- Pode-se utilizar um registrador ($s1) para armazenar o início das variáveis globais.
- O valor de $sp no início da geração de código é copiado em $s1
- O valor de $s1 não é alterado durante a execução do programa.
- O $sp é deslocado para baixo na pilha para cada variável global.
- O deslocamento é igual ao tamanho da variável.
- Na tabela de símbolos, é armazenado para cada variável global: o seu escopo (0) e sua posição em relação ao início da lista de variáveis.
- Uma variável array de tamanho n conta como n variáveis escalares em termos de posicionamento na lista.

Acesso a variáveis não locais

Acesso a variáveis globais

- O acesso a uma variável global g é feito do seguinte modo:
  - Pesquisar pelo nome de g na tabela de símbolos. A entrada para g na tabela deve conter escopo (0) e sua posição na lista de variáveis globais.
  - Seja p a posição de g na lista; o acesso a g é feito a partir de \$s1 do seguinte modo: $-(p-1)*4(\$s1)$.

Por exemplo:

- Amazenamento em g:
  `print ("sw $s0,", -(p-1)*4,"($s1)")`
- Carga a partir de g:
  `print("lw $s0,", -(p-1)*4,"($s1)")`

Acesso a variáveis globais

Exemplo

- Suponha as seguintes declarações globais: `int x; int y[4]; int z;`
- Na tabela de símbolos os valores de posição para cada variável são: $1, 2, 6$ (`z` inicia na posição 6, dado que y ocupa 4 posições).
- A geração de código para armazenamento em `z`: `print("sw $s0,", -20,"($s1)")`
- A geração de código para armazenamento em em y[2]: `print("sw $s0,", -k,"($s1)")# onde` $k = (p - 1 + indice) * 4$ =$((2 - 1) + 2) * 4 = 12$

Acesso a variáveis de blocos aninhados

Implementação de escopos em bloco

- Na linguagem Cafezino e na linguagem C pode ocorrer aninhamento de blocos.
- Uma variável x declarada em um bloco pode ser acessada em um bloco mais interno, a menos que outra variável com mesmo nome seja declarada no bloco mais interno.
- O aninhamento de blocos segue também o comportamento de uma pilha: as variáveis do bloco mais externo devem aparecer mais cedo na pilha e as variáveis do bloco mais interno devem aparecer no topo da pilha. Quando termina o bloco mais interno, suas variáveis são desempilhadas.

Acesso a variáveis de blocos aninhados

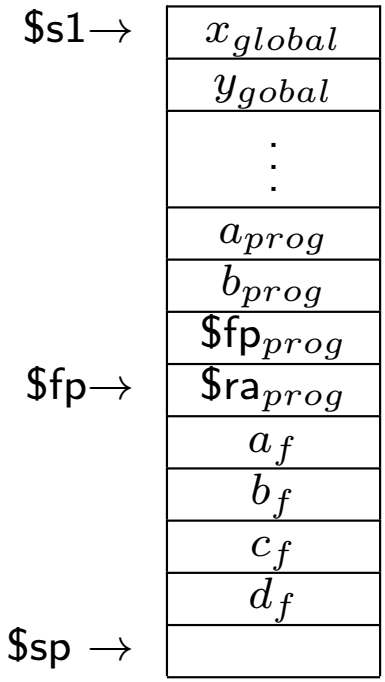
Exemplo

```
int x,y; //variáveis globais
int f(){
    int a,b;
    ...
    { int c,d; ... a=2; }
}
programa{
    int a,b;
    ...
    f();
    ...
}
```

Acesso a variáveis de blocos aninhados

Imagem da memória

- Imagem da memória quando o bloco mais interno de f está ativo:



Acesso a variáveis de blocos aninhados

Sugestão

- As variáveis são alocadas na pilha na sequência em que aparecem nas declarações. Ver figura anterior.
- As variáveis declaradas no bloco mais externo na função têm escopo igual a 1 (0 é o escopo das variáveis globais).
- A cada bloco mais interno o escopo é aumentado de um.

Acesso a variáveis em blocos aninhados

Sugestão

- O acesso às variáveis é feito tomando-se por base o endereço de memória apontado por $fp.
- Durante a geração de código para uma função, mantém-se um vetor $escopo$, indexado pelos escopos (níveis de bloco).
- No índice i, $i \geq 1$ do vetor $escopo$, é armazenado o número de variáveis no escopo i.
- Seja $esc(v)$ o escopo (ou bloco) de uma variável v dentro da função e seja $pos(v)$ sua posição na declaração de variáveis no bloco. O acesso a v é feito do seguinte modo:

$desloc = 0$;
`for(`$i = 1; i < esc(v); i++$`)`
    $desloc += escopo[i]$;
o endereço de v corresponde a: fp-$(desloc + pos(v) + indice) * 4$,
onde $indice = 0$ se v for escalar

Acesso a variáveis em blocos aninhados

Informações sobre escopo e posição de variáveis

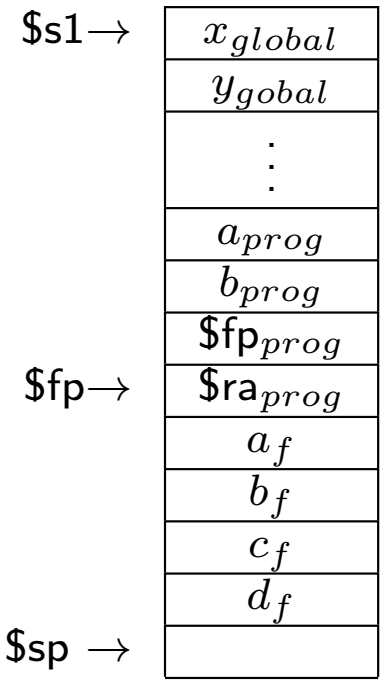
- Os valores de $esc(v)$ - escopo da variável v e $pos(v)$ devem estar disponíveis na entrada da tabela de símbolos para a variável v.
- O vetor $escopo$ deve ser eliminado ao final da geração de código de uma função. Como cada função ativa deve ter seu próprio vetor $escopo$, vários vetores $escopo$ podem coexistir. Uma possibilidade é alocar dinamicamente o vetor $escopo$ assim que a geração de código para uma função for iniciada. A área do vetor é desalocada assim que a geração de código para a função terminar.

Acesso a variáveis em blocos aninhados - Exemplo

Exemplo de código

```
int x,y; //variáveis globais
int f(){
    int a,b;
    ...
    { int c,d; ... a=d; }
}
programa{
    int a,b;
    ...
    f();
    ...
}
```

Memória



Acesso a a e c

Ao gerar código para a=2; no bloco mais interno de f:

- Consulta-se a tabela de símbolos com o nome a;
- A consulta retorna informações sobre a_f: $esc(a_f) = 1$ e $pos(a_f) = 1$.
- O acesso a a (a_f na figura) é dado por: \$fp $-(desloc + pos(a_f)) * 4 =$ \$fp $-(0 + 1) * 4$.
- De modo semelhante, o acesso a c (c_f na figura) é dado por \$fp $-(desloc + pos(c_f)) * 4 =$ \$fp $-(2 + 1) * 4$

Exemplo - continuação

Cálculo do deslocamento.

- O deslocamento é computado utilizando-se o número de variáveis nos escopos que envolvem o escopo da variável considerada.
- No caso de a_f, o escopo é 1. Nesse caso, considera-se no deslocamento apenas a posição da variável dentro do escopo, ou seja, o vetor $escopo$ não é considerado.
- No caso de c_f, o escopo é 2. Nesse caso, considera-se no deslocamento, o número de variáveis no escopo 1, isto é, $escopo[1]$, e a posição da variável c_f na lista de variáveis no bloco onde c ocorre.
- o vetor $escopo$ para a função f no exemplo têm os seguintes valores: $escopo[1] = 2$ e $escopo[2] = 2$.

Operações de EntradaSaída no SPIM ou Mars

System Calls

- Os simuladores SPIM e Mars possuem chamadas ao sistema para escrita de números e strings.
- Cada chamada possui como código um valor inteiro a ser colocado no registrador $v0 antes da chamada.
- A tabela abaixo mostra as principais tarefas de entrada e saída realizada por esses simuladores:

Código	Argumentos de entrada	Resultado
1	$a0 = valor inteiro	Imprime valor inteiro
4	$a0=string de caracteres	Imprime cadeia
5		Lê valor inteiro em $v0

- O valor a ser operado fica em $a0.

Geração de código para o comando escreva

Exemplo

- Considere os seguinte comandos:
  `escreva("A resposta e:  "); escreva (5);`
- O código correspondente em MIPS:

```
        .data
str:
        .asciiz "A resposta e:  "
        .text
        li $v0, 4 # código de chamada para impressao de string
        la, $a0, str # carrega em $a0 o endereço da string
        syscall #executa a chamda ao sistema
        #
        li $v0, 1 # código para imprimir um valor inteiro
        li $a0, 5 # carrega em $a0 o valor a ser impresso
        syscall
```

Geração de código para o comando `leia`

Exemplo

- Considere o seguinte comando:
  `leia(x);`
- O código correspondente em MIPS:

```
li $v0, 5 # código de chamada para leitura de um inteiro
syscall #executa a chamda ao sistema
# armazene o valor em $v0 no endereço da pilha correspondente a x
```