Universidade de Évora

Compiladores

2ª Frequência - 23 de Maio de 2016

(1) 1. Qual é a importância da Symbol Table durante o desenho do Registo de Activação? (escolha a hipótese mais correcta)

A. A Symbol Table contém informação sobre os tipos das variáveis;

B. A Symbol Table contém informação sobre os tipos das variáveis e das funções;

C. A Symbol Table contém o tamanho e offset de cada variável;

D. A Symbol Table contém o endereço do frame pointer;

E. A Symbol Table não tem importância no desenho do Registo de Activação.

Solução: Embora os items (A) e (B) sejam verdade, não têm qualquer relevância para o desenho do Registo de Activação. O (D) é completamente falso, bem como o (E).

2. Considere o seguinte programa em *Ya!*:

```
1  da(x : int[10], y : int[10]) : void {
2    t : int = 2;
3    i : int = 0;
4
5    while i < 10 do {
6      y[i] = x[i] * t;
7      i = i + 1;
8    };
9  };
10
11 main() : void {
12   i : int = 0;
13   a, b : int[10];
14
15   while i < 10 do {
16     a[i] = i;
17     i = i + 1;
18   };
19
20   da(a, b);
21
22   print(b[3]);
23 };
```

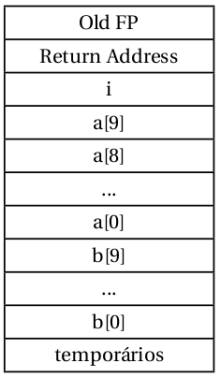
(1) (a) Resumidamente, o que faz a função `da()`?

Solução: A função `da()` recebe dois arrays de 10 inteiros, `x` e `y`, e altera `y` para conter os dobros dos elementos correspondentes em `x`.

(2,5) (b) Desenhe o registo de activação da função `main()`.

Solução: A função `main()` não recebe argumentos, mas tem três variáveis locais: `i`, `a` e `b`.

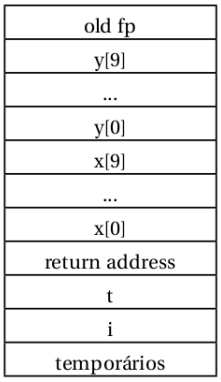
O seu registo de activação ficará então parecido com o seguinte:

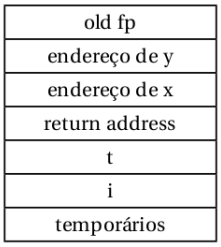


Atenção que os arrays ficam ao contrário, para que os índices possam ser somados à base (os endereços da stack "sobem para cima", e os dos arrays também).

(2,5) (c) Desenhe o registo de activação da função `da()`.

Solução: Aqui podemos assumir que os arrays são passados por valor (representação da esquerda) ou por referência (representação da direita). A diferença prende-se depois com o código gerado para aceder a coisas do tipo `x[i]`. Se passarmos por valor, `x[i]` estará em `mem(fp + 1 + i)`. Se passarmos por referência, `x[i]` estará em `mem(mem(fp + 1) + i)`.





(2) (d) Usando o esquema de geração de código para máquina de pilha com instruções MIPS estudado nas aulas, proponha um padrão para geração de código para ciclos `while`.
(pretende-se a definição para a função `codegen()` de um nó `while` da APT)

Solução: Podemos pegar no exemplo do `if exp1 then exp2 else exp3`, visto nas aulas:

```
1    codegen(exp1)
2    sw $t0, 0($sp)
```

```
    addiu $sp, $sp, -4
    codegen(exp2)
    lw $t1, 4($sp)
    addiu $sp, $sp, 4
    beq $t0, $t1, lbl_iftrue
lbl_iffalse:
    codegen(exp4)
    j lbl_endif
lbl_iftrue:
    codegen(exp3)
lbl_endif:
```

As diferenças básicas são as seguintes:

1. Não há `else`;
2. Não temos uma comparação entre `exp1` e `exp2`, mas sim uma expressão booleana qualquer, como condição do while;
3. O ciclo salta sempre para o início, ao contrário do `if` que só é executado uma vez.

Para o (1), basta não incluir a parte do `iffalse`. Para o (2), podemos assumir que a expressão condicional do while é gerada normalmente, e o seu resultado fica em `$t0` (vamos pensar que se a expressão dá `true`, `$t0` fica com o valor `1`, e `0` caso dê `false`.

O `codegen` para `while exp1 do exp2` ficaria então assim:

```
lbl_while:
    codegen(exp1)           # gerar o código para exp1
    beq $t0, $0, lbl_false  # se falso, sair do while
    codegen(exp2)           # gerar o corpo do while
    j lbl_while             # repetir tudo (recalculando a exp1)
lbl_false:
```

(2,5) (e) Como seria o código gerado para a linha 20 (`da(a, b)`)?

Solução: Aqui seria simplesmente uma chamada de função, tal como está nos slides das teóricas:

```
    sw $fp, 0($sp)
    addiu $sp, $sp, -4
    codegen(b)
    sw $t0, 0($sp)
    addiu $sp, $sp, -4
    codegen(a)
```

```
7     sw $t0, 0($sp)
8     addiu $sp, $sp, -4
9     jal func_da
10
```

Na linha 1, guardamos o `fp` antigo na stack (oldfp). Baixamos o `sp` e geramos o código para a variável `b`, que fica em `$t0`. Colocamos `b` na stack e fazemos o mesmo para `a`. Tendo construído a parte do registo de activação acima do `fp`, chamamos a função, que se encarregará de preencher o resto do registo de activação (a parte das variáveis locais).

(1,5) (f) Quantos temporários são necessários para o ciclo `while` da função `da()`? (apresente os cálculos que efectuar)

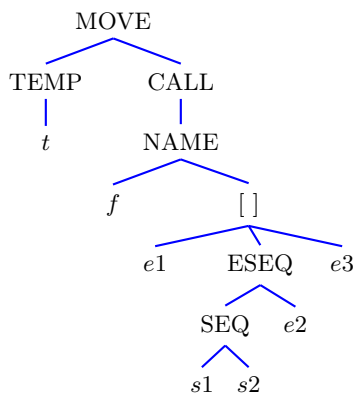
Solução: Aqui temos de calcular o máximo de temporários necessários para as linhas 5, 6 e 7. Para a linha 5 só precisamos de 1 temporário. Para a linha 7 também basta um temporário. Resta saber quantos temporários precisamos para a linha 6. Se for mais do que um, esse será o máximo que procuramos.

Vamos tentar perceber quantos temporários são necessários para fazer a afectação:

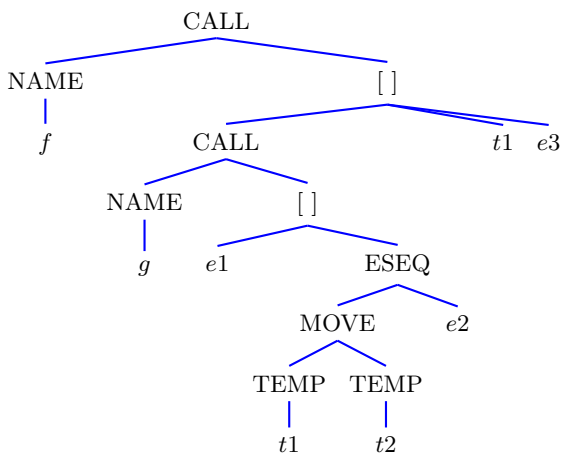
- Precisamos de um temporário para o valor que está em x[i];
- Esse temporário pode ser reutilizado para colocar x[i] * t;
- Precisamos de um temporário para saber o endereço de memória onde está y[i];
- No final temos um `sw` do valor de x[i] * t para o endereço de y[i], ou seja, precisamos de dois temporários.

(3) 3. Utilizando as regras de reescrita para árvores canónicas, proponha formas optimizadas para as seguintes árvores de Representação Intermédia:

(a)



(b)



Solução:

(a):

Queremos tirar o ESEQ de dentro do CALL. Para isso, temos de salvaguardar e1 antes de executar s1 e s2. Como temos um MOVE, basta usarmos SEQs. O nosso código fica assim:

$$SEQ(MOVE(TEMP(auxe1), e1, SEQ(s1, SEQ(s2, MOVE(TEMP(t),$$

$$CALL(f, [TEMP(auxe1), e2, e3])))$$

.

(b):

Temos de retirar o ESEQ de dentro da chamada a `g()`, e também temos de retirar a chamada a `g()` de dentro da chamada a `f()`.

Para ser mais simples, vejamos em código "normal":

`f(g(e1, (t1=t2;e2), t1, e3)`

Vamos colocar a sequência de nós, omitindo, para já, os SEQs.

Como não sabemos se e1 usa t1, temos de salvaguardar e1:

(1) `MOVE(TEMP(auxe1), e1)`

Agora já podemos chamar t1 = t2:

(2) `MOVE(TEMP(t1), TEMP(t2))`

O primeiro argumento de f() é uma chamada a g(), logo temos de fazer essa chamada cá fora:

(3) `MOVE(TEMP(auxg, CALL(g, [auxe1, e2])))`

Tendo todos os argumentos calculados e salvaguardados, basta-nos chamar f():

(4) `CALL(f, [TEMP(auxg), t1, e3])`

E pronto! Agora temos uma sequência de 4 instruções, mas queremos o resultado de f(), ou seja, temos um ESEQ com 3 instruções e a chamada a f():

`ESEQ(SEQ(`*(1)*`, SEQ(`*(2),(3)*`)),` *(4)*`)`

A árvore dentro do ESEQ é canónica, e o ESEQ poderá ser enviado mais para cima na árvore, quando conhecermos o resto do código.

(2) 4. Para o programa do exercício (**2**), proponha uma Representação Intermédia para a linha 6. (`y[i] = x[i] * t`)

Solução:

5. Considere, numa determinada representação intermédia, o seguinte excerto de código:

```
1  L1:
2    MOVE t1 a
3    MOVE a t1
4    MOVE t1 CONST(1)
5    JUMPIFZERO t1 L1
6  L2:
7    GOTO L3
8    MOVE t2 MEM(a)
9  L3:
10   CALL L4 t1 t2
11   ADD t3 t3 CONST(0)
12   SUB t1 t3 t2
13   JUMP L2
```

(2) (a) Proponha optimizações para o código apresentado.

Solução:

- Nas linhas 2 e 3 temos dois MOVEs redundantes, pelo que podemos apagar o segundo;
- Nas linhas 4 e 5 temos um registo t1 com valor 1, e um salto se o valor for 0 (nunca vai acontecer), podemos eliminar o salto que nunca ocorre;
- Na linha 8 temos código inacessível (como não tem um LABEL atrás, não há nenhuma forma de o código chegar a essa linha), por isso podemos remover;
- Na linha 11 temos uma soma com 0 (elemento neutro), ou seja, não faz nada;
- Na linha 13 temos um salto para outro salto, podemos simplificar para JUMP L3.