Linguagens e Ambientes de Programação (2018/2019)

Teórica 24 (03/jun/2019)

Execução de programas
Modelos de execução para diversos tipos de linguagens de programação.
Modelo de execução detalhado para linguagens com escopo estático e aninhamento de funções.

Modelo de execução para diversos tipos de linguagens de programação

Na implementação da maioria das linguagens de programação, é necessário considerar diversos aspetos para garantir que os efeitos da execução dos programas são os desejados. Algumas das questões mais importantes a considerar são as seguintes:

1. Avaliação de expressões
2. Acesso a variáveis
3. Chamada e retorno de funções
4. Passagem de funções como argumento
5. Funções retornadas por outras funções

Vamos estudar em sucessão estas questões com a ajuda dum modelo de execução. No nosso modelo fazemos algumas escolhas, o que significa que também se podia ter usado um modelo um pouco diferente. **Em todo o caso, o objetivo de qualquer modelo é simplificar o estudo uma realidade, e omitir deliberadamente pormenores não essenciais e outras complicações.** Por exemplo, assim, no nosso modelo, resolvemos não usar registos do CPU para avaliar expressões, apesar das implementações reais e eficientes das linguagem de programação normalmente os usarem.

1. Avaliação de expressões

No nosso modelo usamos uma pilha para avaliar expressões. Para cada operação a executar, primeiro os seus argumentos são empilhados (pela ordem em que aparecem escritos) e depois a operação é efetuada. A operação retira os argumentos do topo da pilha, calcula o resultado e deixa esse resultado no topo da pilha.

Por exemplo, para avaliar a expressão `1+2` seguindo as regras atrás descritas, o código gerado pelo compilador seria algo do género:

```
PushConst 1   // empilha a constante 1
PushConst 2   // empilha a constante 2
Add           // desempilha dois valores do topo e empilha o resultado
```

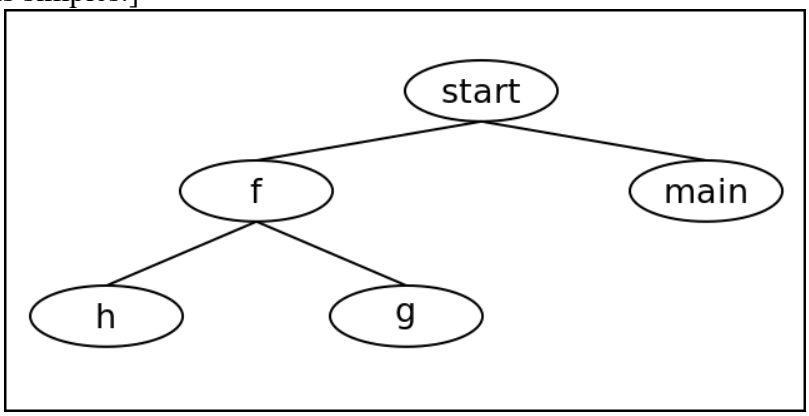
Não há mais nada a dizer sobre o primeiro dos cinco itens.

2. Acesso a variáveis

1 - Numa **linguagem sem recursividade nem aninhamento de funções**, como é o caso do Fortran 77, cada procedimento é dono duma zona de memória própria onde as suas variáveis locais são guardadas. Todas as variáveis são estáticas, ou seja têm localizações de memória fixas. Diversas chamadas do mesmo procedimento reutilizam a mesma zona de memória... não há problema nisto porque, como foi dito, a recursividade não é permitida.

2 - Numa **linguagem com recursividade mas sem aninhamento de funções**, como é o caso do C, há dois casos a considerar. As **variáveis globais**, ou seja as variáveis declaradas fora das funções, são consideradas variáveis estáticas e têm localizações de memória fixas. Já as **variáveis locais** são criadas dinamicamente no topo duma pilha sempre que a sua função-dona é ativada; são removidas do topo da pilha quando a execução da função-dona termina. A partir do interior duma função em C, as variáveis acessíveis são as seguintes: as variáveis locais da ativação mais recente dessa função e as variáveis globais que não tenham sido redefinidas dentro da função. Mais nenhumas variáveis são acessíveis de forma direta.

3 - Numa **linguagem com recursividade e com aninhamento de funções**, como são os casos do Pascal, do OCaml e do GCC (uma variante do C enriquecida), a implementação fica mais complicada porque pretendemos usar a semântica do **escopo estático**. Agora surgem variáveis declaradas nas funções envolventes, as chamadas **variáveis intermédias**. Para aceder às **variáveis intermédias** é preciso guardar na ativação de cada função o respetivo ambiente de definição (que é a função envolvente) representado por um apontador (**static link**) para a ativação mais recente da função envolvente na pilha. Para aceder a uma variável intermédia é preciso percorrer parte duma lista ligada constituída por static links. Foi Dijkstra quem, em 1960, inventou a técnica dos static links. [Numa linguagem com escopo dinâmico, não é necessário usar static links, pelo que a implementação fica mais simples.]



Para perceber melhor a estrutura dum programa, pode valer a pena desenhar a chamada **árvore do programa**, que mostra como as várias funções se relacionam. A imagem à direita corresponde ao programa que vai aparecer já a seguir. Uma árvore de programa tem sempre na raiz uma função imaginária chamada **start**, que contém todo o programa, incluindo as variáveis globais. Por influência desta árvore, muitas vezes usam-se as relações familiares humanas para referenciar as funções. Diz-se, por exemplo, que a função **g** é irmã de **h**, filha de **f**, sobrinha de **main** e neta de **start**.

Considere o exemplo abaixo, em GCC, com aninhamento de funções. Neste programa encontram-se muitos exemplos de variáveis intermédias. Por exemplo, do ponto de vista do interior da função **h**, as variáveis locais da função **f** que forem acessíveis são consideradas variáveis intermédias. Vamos ser mais completos: do ponto de vista do interior da função **h**, esta tem acesso a uma variável local chamada **i**, a uma variável intermédia **j** na função-mãe e duas variáveis globais **k** e **l** na função avó. Em tempo de execução, para a função aceder por exemplo à variável global **k** da avó, precisa de dar dois saltos na lista de static links para alcançar o registo de ativação que contém a variável pretendida. É um pouco complicado e este exemplo mostra um dos preços a pagar por se usar escopo estático.

```
#include <stdio.h>

int i = 1, j = 1, k = 1, l = 1;    // variáveis globais

void f(void) {
    int i = 2, j = 2;
    void h(void) {        // função aninhada
        int i = 3;
        l = i + j + k;
    }
    void g(void) {        // outra função aninhada
        int i = 7, j = 7, k = 7, l = 7;
        h();
    }
```

```
    g();
}

int main(void)
{
    f();
    printf("%d\n", l);
    return 0;
}
```

Nota importante: GCC é um nome duma linguagem um pouco mais vasta do que o C de base. Ao contrário do ANSI-C, o GCC suporta aninhamento de funções.

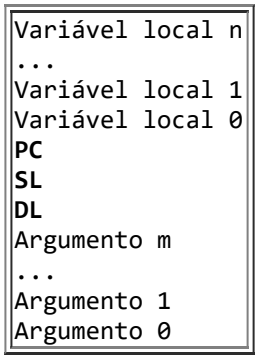
3. Chamada e retorno de funções

Nas linguagem que suportam recursividade, o estado da invocação duma função é capturado numa estrutura chamada **registo de ativação (frame)**. Num registo de ativação guarda-se a seguinte informação: os argumentos e as variáveis locais da função, o endereço de retorno da função, e um apontador para o registo de ativação da função que fez a chamada (**dynamic link**).

Se a linguagem também permitir aninhamento de funções e se pretendermos usar escopo estático, ainda é preciso guardar no registo de ativação o ambiente de definição da função ou seja um apontador para o registo de ativação mais recente da função envolvente (**static link**). Repare que a função envolvente pode não ser a mesma função que fez a chamada.

Os registos de ativação organizam-se de forma natural numa pilha, a chamada **pilha de execução**. A chamada duma função causa o empilhamento dum novo registo de ativação; o retorno duma função causa o desempilhar do registo de ativação mais recente.

Eis um possível formato para os registos de ativação:



4. Passagem de funções como argumento - closures

No caso duma linguagem sem aninhamento de funções, como o C, a passagem duma função como argumento é trivial: basta passar o endereço da função.

No caso duma linguagem com aninhamento de funções e com escopo estático, a passagem duma função como argumento é mais complicada. A função passada pode ter necessidade de aceder a variáveis intermédias, e por isso, juntamente com o endereço da função tem também de ser passado o respetivo ambiente de definição (static link).

Portanto a implementação da passagem duma função como argumento envolve tecnicamente a passagem dum par ordenado a que se chama **closure** (ou **fecho** em Português):

closure = (endereço da função, static link)

A razão de ser do nome closure é a seguinte: Em geral, uma função é uma entidade "aberta" (incompleta) pois o seu corpo pode referir entidades desconhecidas localmente. Mas o par *função + respetivo ambiente de*

definição já é uma entidade fechada pois o ambiente de definição "fecha" (completa) o significado da função.

No seguinte exemplo, em GCC, ocorre uma passagem de função como argumento. Medite no código para confirmar a necessidade das closures.

```
#include <stdio.h>

int i = 1, j = 1, k = 1, l = 1;

void f(void) {
    int i = 2, j = 2;
    void h(void) {        // função aninhada
        int i = 3;
        l = i + j + k;
    }
    void g(void p(void)) {     // outra função aninhada com par. funcional
        int i = 7, j = 7, k = 7, l = 7;
        p();
    }
    g(h);
}

int main(void)
{
    f();
    printf("%d\n", l);
    return 0;
}
```

5. Funções retornadas por outras funções

No caso duma linguagem sem aninhamento de funções, como o C, implementar o retorno duma função a partir de outra função é trivial: basta retornar o endereço da função-resultado.

No caso duma linguagem com aninhamento de funções e com escopo estático, como o OCaml, a implementação desse mecanismo é mais complicada. Quando se retorna uma função, o que tecnicamente precisa de ser retornado é uma closure.

Mas aqui pode haver problema. A closure retornada pode depender de entidades locais à função que produz o resultado. Assim quando a função retorna a closure e termina, o seu registo de ativação não pode ser destruído. Uma solução é transferir o registo de ativação para o heap, ou seja para a zona das variáveis dinâmicas. Esta é a solução adotada na linguagem funcional Lua. Outra solução, mais radical, consiste em abandonar completamente o modelo da pilha de execução e passar a criar todos os registos de ativação no heap (deixando que seja o gestor de memória a descobrir o momento em que eles podem ser removidos com segurança).

Esta complicação explica o facto do retorno de funções ser um mecanismo não suportado pela maioria das linguagens de programação. Praticamente só as linguagens funcionais é que suportam este mecanismo.

No seguinte exemplo, em OCaml, a função f retorna uma closure que depende do x local ao f. Quando a closure retornada é depois chamada, o x tem de continuar disponível, apesar da execução de f já ter terminado.

```
let f x =
    let g y = x + y in
        g
;;

(f 5) 4 ;;
```

Modelo de execução detalhado para linguagem com escopo estático e aninhamento de funções

As explicações da secção anterior já são suficientes para se perceber como funciona a pilha de execução numa linguagem com recursividade, aninhamento de funções e escopo estático. Inclusivamente, são suficientes para conseguirmos fazer uma execução manual dum programa, desenhando a pilha de execução usando papel e lápis.

Mas se pretendermos automatizar a execução dum programa, precisamos de introduzir mais detalhe e mais um ou dois conceitos novos. Esta nova secção lida com o problema de automatizar a execução dum programa. Apresenta uma implementação parcial em C dos mecanismos de execução estudados. O código C terá grandes semelhanças com o código duma máquina virtual típica.

A grande diferença entre uma execução manual e uma execução automática concretiza-se no tratamento dos acessos a entidades. Numa execução manual não temos problema em perceber qual é a entidade referida referida por um nome - basta procurar no texto do programa (tendo em conta a regra de escopo estático) e também podemos olhar para a árvore do programa.

Mas numa execução automática, já não é prático trabalhar com o texto do programa. Contém usar valores numéricos que capturem os aspetos desses entidades que são relevantes para a execução - por exemplo no caso duma variável, precisamos de dois valores numéricos: (1) o offset da variável dentro do seu registo de ativação; (2) a diferença de nível na árvore do programa entre o ponto do uso e o ponto da definição.

Áreas de memória e registos

O nosso modelo tem três áreas de memória:

- **stack** - Guarda os registos de ativação e também como memória de trabalho para avaliar expressões.
- **code** - Guarda o código do programa que está a correr.
- **heap** - Guarda as variáveis dinâmicas do programa.

Pouco será dito sobre a zona de código e sobre o heap. O stack vai receber quase toda a nossa atenção.

Os registos do modelo são os seguintes:

- **SP** - Indica o topo da pilha de execução.
- **FP** - Indica o registo de execução correntemente ativo, ou seja para o registo de execução que se encontra no topo da pilha de execução.
- **PC** - Indica qual a instrução corrente, na zona de código.

Eis uma concretização em C destas ideias:

```
#define STACK_SIZE   20000
#define CODE_SIZE    20000
#define HEAP_SIZE    20000

typedef unsigned Word;
typedef Word *Pt;

#define Push(v)      (*SP++ = (Word)(v))
#define Pop()        (*--SP)

Word stack[STACK_SIZE];
Word code[CODE_SIZE];
Word heap[HEAP_SIZE];

Pt SP = stack;       // Stack pointer
```

```
Pt FP = stack;      // Frame pointer
Pt PC = code;       // Program counter
```

Avaliação de expressões

Como já foi dito, usamos a pilha de execução como auxiliar da avaliação de expressões. Os argumentos das operações são empilhados e depois a operação encontra os seus argumentos no topo da pilha.

Eis uma concretização em C destas ideias:

```
void PushConst(Word c) {
    Push(c);              // push the constant
}

void Add(void) {
    Word arg2 = Pop();    // pop the second argument
    Word arg1 = Pop();    // pop the first argument
    Push(arg1 + arg2);    // push the result
}

...
```

Acesso a variáveis

Em tempo de execução os nomes das funções não estão disponíveis. Para referenciar uma variável em tempo de execução basta usar as chamadas **coordenadas da variável**, constituídas por dois valores:

- **Diferença de nível (nesting)** - Diferença de nível entre o ponto da referência e o ponto da declaração.
- **Deslocamento (offset)** - Deslocamento da variável dentro do registo de ativação a que pertence.

A diferença de nível representa o número de static links que é preciso seguir para se atingir o registo de ativação onde se encontra a variável pretendida. No caso do acesso a uma variável local, a diferença de nível é zero.

O compilador da linguagem consegue determinar com muita facilidade as coordenadas que permitem referenciar cada variável. Um ser humano também o faz facilmente. Por exemplo, no primeiro exemplo desta aula, qual é a diferença de nível das utilizações das variáveis i, j, k e l dentro da função f?

Numa execução, numa atribuição, primeiro determina-se o valor da expressão da direita (o valor é empilhado), depois determina-se o endereço representado pela expressão da esquerda (o endereço também é empilhado), e no final guarda-se o valor calculado na posição de memória indicada pelo endereço (os dois elementos empilhados são desempilhados).

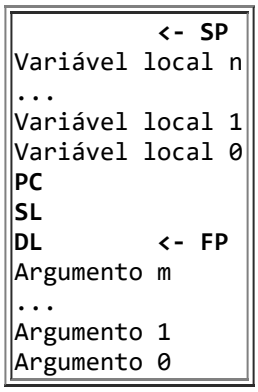
Concretização em C:

```
void LoadVarAddr(int nesting, int offset) {
    for( Pt pt = FP ; nesting-- ; pt = SL(pt) );  // find the frame of the var
    Push(pt + offset);            // push the var address
}

void Value() {
    Pt pt = Pop();        // pop the var address
    Push(*pt);            // push the value of the variable
}

void Assign() {
    Pt pt = Pop();        // pop the var address
    Pt value = Pop();     // pop the value to store
    *pt = value;          // do the assignement
}
```

Chamada e retorno de funções

No nosso modelo, o formato dum registo de ativação é o que se apresenta abaixo. Convenciona-se que a posição do DL é o ponto de referência oficial de qualquer registo de ativação. Quem aponta para um registo de ativação aponta sempre para o respetivo DL.

O registo FP aponta para a célula onde é guardado o dynamic link do registo de ativação mais recente. Nos acessos aos argumentos e variáveis da função, usa-se o registo FP como ponto de referência. Assim, os argumentos da função têm offset negativo, a começar em -1, e as variáveis locais têm offset positivo, a começar em 3. No diagrama assume-se que a pilha cresce para cima.



A principal complicação envolvida na ativação duma função e na criação do seu registo de ativação é o cálculo do static link (que representa o ambiente de definição dessa função). O static link tem de ser guardado no registo de ativação da função para permitir o subsequente acesso a variáveis não-locais. Para determinar o static link é preciso conhecer a **diferença de nível (nesting)** entre o ponto da declaração da função e o ponto da sua chamada. A diferença de nível representa o número de static links para é necessário percorrer para se atingir o registo de ativação onde se encontra o static link a copiar. Numa chamada recursiva, a diferença de nível é zero. Na chamada duma função filha, a diferença de nível é -1.

No nosso modelo a chamada e retorno de funções estão organizados da seguinte forma:

- Antes da função ser chamada, os seus argumentos são empilhados;
- No momento da chamada são empilhados o dynamic link, o static link e o endereço de retorno;
- À entrada da função, esta reserva espaço para as suas variáveis locais.
- Quando a função termina ela desempilha o registo de ativação completo, incluindo os argumentos.
- A função deixa no topo da pilha o seu resultado.

Concretização em C:

```
#define DL(fp)     ((fp)[0])
#define SL(fp)     ((fp)[1])
#define RET(fp)    ((fp)[2])

void Call(int nesting, Pt addr) {
    Push(FP);          // push dynamic link
    if( nesting == -1 )
        Push(FP);      // push static link
    else {
        for( Pt pt = FP ; nesting-- ; pt = SL(pt) );
        Push(SL(pt));  // push static link
    }
    Push(PC);          // push return address
    PC = addr;         // jump to the function
}

void Enter(int locSpace) {
    if( STACK_SIZE - (SP - stack) < locSpace + 20 ) {
        fprintf(stderr, "Stack overflow at PC = %d\n", PC);
        exit(1);
```

```
    }
    FP = SP - 3;    // FP points to the dynamic link inside the frame
    SP += locSpace;
}

void Exit(int argSpace) {
    Word res = Pop();     // save the result here
    SP = FP - paramSpace; // discard the entire frame, including the arguments
    PC = RET(FP);         // restore the program counter
    FP = DL(FP);          // restore the previous frame pointer
    Push(res);            // push the result
}
```

Passagem de funções como argumento - closures

Já sabemos que a passagem duma função como argumento requer efetivamente a passagem duma closure como argumento. A principal complicação envolvida na criação da closure é o calculo do static link. Mas agora não há qualquer novidade: ele calcula-se exatamente da mesma forma que na chamada duma função normal.

A ativação duma função através duma closure é mais simples do que a ativação direta duma função pois o static link já se encontra calculado na closure.

Concretização em C:

```
void PushClosureArg(int nesting, Pt addr) {
    if( nesting == -1 )
        Push(FP);      // push static link
    else {
        for( Pt pt = FP ; nesting-- ; pt = SL(pt) );
        Push(SL(pt));  // push static link
    }
    Push(addr);
}

void CallClosure(int nesting, int offset) {
    for( Pt pt = FP ; nesting-- ; pt = SL(pt) );  // find the frame of the closure-argument
    Pt sl = pt[offset];       // get the static link from the closure
    Pt addr = pt[offset + 1]; // get the function address from the closure
    Push(FP);                 // push dynamic link
    Push(sl);                 // push static link
    Push(PC);                 // push return address
    PC = addr;                // jump to the function
}
```

Funções retornadas por outras funções

O estudo mais detalhado desta questão é um tópico avançado que está fora do âmbito da cadeira de LAP.

Exercícios

- Relativamente ao primeiro exemplo desta aula, mostre o estado da pilha de execução à entrada da função h.
- Relativamente ao segundo exemplo desta aula, mostre o estado da pilha de execução à entrada da função h. (Será feito numa aula prática.)

Para efeitos da criação do registo de ativação inicial, convém imaginar que cada programa em C está embebido numa função chamada start, sem argumentos. É importante tratar todas as entidades globais do

programa como sendo locais à função imaginária start.

Assuma também que a primeira célula da pilha de execução é identificada como posição 0, a segunda célula da pilha de execução é identificada como posição 1, etc.

```
Resolução do primeiro exercício:

              <- SP
25: i  3 --h--
    PC ?
    SL 10
    DL 15     <- FP
    l  7 --g--
20: k  7
    j  7
    i  7
    PC ?
    SL 10
15: DL 10
    j  2 --f--
    i  2
    PC ?
    SL 0
10: DL 7
    PC ? --main--
    SL 0
    DL 0
    l  1 --start--
 5: k  1
    j  1
    i  1
    PC ?
    SL ?
 0: DL ?
```

#60