Compiladores

Atenção: A resolução de todos os exercícios deverá ser enviada ao email do professor (macedo@dei.uc.pt), com um subject no seguinte formato: Compiladores-Ficha <#numero da ficha> - <Nome do aluno> até à véspera da PL. Dentro desta mensagem, deverá estar um ficheiro ".zip" cujo nome corresponde ao *login* do aluno e o número da ficha e conterá os ficheiros com a resolução dos problemas. Evidentemente, nestes ficheiros não deve deixar de se identificar nos cabeçalhos.

Por exemplo, a ficha 3, do aluno Carlos Meireles (meiras@student.dei.uc.pt), o subject seria:
Compiladores-Ficha 3 - Carlos Meireles
O ficheiro ".zip" chamar-se-ia meiras03.zip

Ficha prática 10 – Código final (parte II). Definição e chamada de procedimentos e funções. Apoio ao projecto

O redireccionamento de código é um dos instrumentos fundamentais de inúmeras linguagens de programação, nomeadamente através da chamada de procedimentos e funções. Vamos debruçar-nos agora sobre este aspecto, focando, mais uma vez, a pequena linguagem de exemplo que temos utilizado nas últimas aulas.

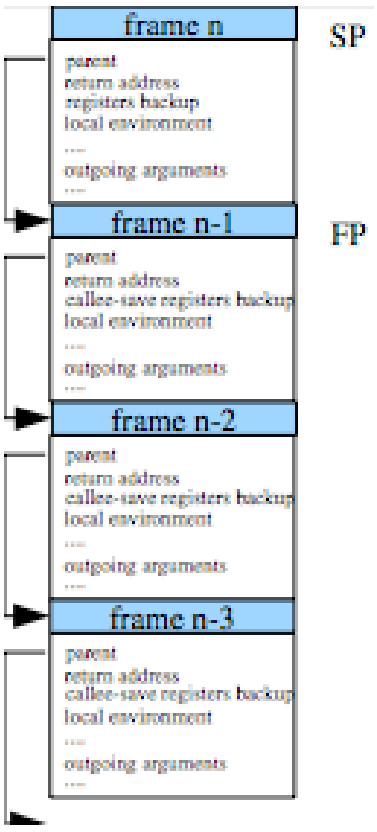

Figura 1 – Pilha de Frames

Geração do código de procedimentos e funções

Em Assembly (tal como no nosso C restringido), é comum colocar todo o conjunto de instruções de um programa num bloco contíguo. Na prática, será o mesmo que ter um programa único grande, que contém vários blocos mais pequenos, cada um associado ao seu procedimento/função. É assim também no nosso caso, em que, de acordo com o enunciado, teremos que colocar todo o código dentro da função *main*.

A geração do código de um procedimento (tal como de uma função) tem que seguir três etapas (ver acetatos teóricas):

1. Geração do *prólogo*, contendo
   a) pseudo-instruções (se necessário - depende do código final em causa)
   b) *label* (etiqueta) para nome de função
   c) instrução para ajuste do ponteiro de pilha de *frames* (frame pointer, fp)
   d) instruções para guardar informação na *frame* e carregar informação em registos
   e) instruções para guardar conteúdos de registos callee-free
2. Geração do código propriamente dito (corpo do procedimento)
3. Geração do *epílogo*
   a) instrução para mover endereço de retorno para registo adequado
   b) instruções de carga para restaurar registos callee-free
   c) instrução para restaurar ponteiro de pilha de molduras
   d) instrução de retorno (JUMP)
   e) pseudo-instruções (eventualmente)

Caso tratemos de uma função, podemos acrescentar ao epílogo também a actualização de um registo, para onde se depositará o valor de retorno da função.

Retomemos então o nosso compilador de exemplo. Existem blocos de procedimentos. A cada procedimento, está associado um nome único, e o seu próprio ambiente (conjunto das suas variáveis locais).

Podemos ver abaixo a geração de código das três etapas para a nossa linguagem de exemplo (veja também os comentários no ficheiro):

```
void translate_procedure(FILE* dest, is_procedure* ip, prog_env* pe)
{

        environment_list* localenv=plookup(pe->procs, ip->name);

        //Prólogo
        fprintf(dest, "goto %sskip;\n", ip->name);
        fprintf(dest, "%s:\n",ip->name);
        fprintf(dest, "fp=sp;\n");
        fprintf(dest, "sp=(frame*)malloc(sizeof(frame));\n");
        fprintf(dest, "sp->parent=fp;\n");
        fprintf(dest, "sp->return_address=_ra;\n");

        //Corpo do procedimento
        translate_vardecs(dest, ip->vlist, localenv, pe);
        translate_statements(dest, ip->slist, localenv, pe);

        //Epilogo
        fprintf(dest, "_ra=sp->return_address;\n");
        fprintf(dest, "sp=sp->parent;\n");
        fprintf(dest, "fp=sp->parent;\n");
```

```
        fprintf(dest, "goto redirector;\n");
        fprintf(dest, "%sskip:\n", ip->name);
}
```

Analisemos então as instruções. A primeira instrução do prólogo ("goto %sskip") serve apenas para evitar que o código seja executado sem o procedimento ser chamado (isso aconteceria da primeira vez que se corresse o código do programa completo, pois todo o código está numa função *main*). Note que a label "%sskip" é a última do epílogo.
A segunda instrução corresponde à alínea 1.*b)* da explicação dada acima. Na terceira linha (correspondente à alínea 1.*c)*), guardamos o ponteiro correspondente à frame em que é feita a chamada para o procedimento. Se não o fizéssemos, perdíamos o acesso a esta frame e, quando se voltasse ao código chamante (quando o procedimento acabasse), não existiria forma de aceder ao seu ambiente. De seguida, é criada a frame actual. Note-se que, sendo C, teremos que alocar memória (com *malloc*), mas caso se tratasse de assembly, esta operação poderia ser tão simples como somar o tamanho previsto da frame ao valor do registo SP actual.
A instrução "sp->parent=fp" é menos trivial de se compreender. A figura 1 permitirá visualizar a ideia: é necessário manter uma lista com a sequência de frames existente na pilha. Na realidade, é esta lista que permite o funcionamento da pilha!
A instrução "sp->return_address=_ra" serve para guardar, na frame, a valor do registo RA. Este registo (que existe nalgumas arquitecturas) contém o endereço de retorno mais recente.
Uma vez que não existem argumentos nos nossos procedimentos, não é necessário ir buscá-los à área "outgoing" da frame chamante ou aos registos do processador. No seu projecto, podemos assumir que todos os argumentos serão passados pela área "outgoing".

Quanto ao *epílogo*, as operações serão essencialmente as inversas do *prólogo*: pretende-se repôr o contexto correcto de forma a voltar para a função/procedimento chamante (repôr endereço de retorno, fazer "pop" da stack, ou seja recolocar o SP e FP nos valores anteriores.

Quanto à acção "goto redirector", corresponde à operação JUMP para o endereço de retorno (que seria feita em Assembly)[1]. No nosso caso, relembramos que em C não existe o conceito de "linha de código". Ou seja, não dá para "ordenar" o processador para ir para uma linha de código específica. Dá, no entanto, para saltar para uma *label*. É isso que se pretende aqui. Cada endereço de retorno estará associado a uma *label*. Essa associação é feita por um "redirector", cujo código poderá ver no ficheiro "result.c" (gerado pelo compilador). Tome muita atenção a esse ficheiro.

Geração do código das chamadas a procedimentos e funções

O código final para uma chamada (CALL) será bastante simples. As tarefas a fazer são:

1. Guardar os argumentos de acordo com o definido nos registos de activação (ou nos registos do processador, ou na zona "outgoing" da frame)
2. Actualizar o registo RA (return address) com o endereço da linha seguinte à chamada.
3. Guardar os registos caller-save.

[1] Este é claramente um dos pontos em que a geração para código Assembly seria mais simples.

4. Executar o procedimento/função (fazer o JUMP para o endereço de código respectivo).
5. Restaurar os registos caller-save

No nosso exemplo, dado que não temos argumentos nos procedimentos, nem registos caller-save para guardar, a função é extremamente simples:

```
void translate_call_stat(FILE* dest, is_call_stat* ias, prog_env* pe)
{
        fprintf(dest, "_ra=%d;\n",returncounter);
        fprintf(dest, "goto %s;\n", ias->proc);
        fprintf(dest, "return%d:\n", returncounter);
        returncounter++;

}
```

Repare que existe um "returncounter" (iniciado a 0), que vai ser usado para criar endereços de retorno. Portanto, no nosso caso, existem apenas duas operações: guarda do endereço de retorno; salto para o procedimento. A linha "return%d:" serve apenas para criar uma label correspondente ao endereço de retorno.

Repare que, caso houvessem argumentos e se tratasse de uma função, existiriam várias outras tarefas a executar!

Veja o código "translate.c" com muita atenção para perceber os conceitos de que tratamos.

Exercícios:

1. Acrescente a possibilidade de enviar argumentos para os procedimentos da nossa linguagem.
2. Acrescente a possibilidade dos procedimentos retornarem um valor (passando a ser funções).
3. Aplique estes conceitos ao seu projecto.

Bibliografia recomendada:
. Manuais da linguagem C (por exemplo, *Linguagem C*, Luís Damas, FCA. 1999)