COMPILADORES

Prova 2 – 05/12/2009, Prof. Marcus Ramos

1. (2 pontos) Responda:

   - Em que consiste a subfase de identificação durante a fase de análise de contexto?

     Consiste na vinculação entre todas as referências e todas as declarações de nomes. Essa vinculação é feita com base nas regras de escopo da linguagem.

   - Qual a importância da tabela de símbolos durante essa subfase?

     A tabela de símbolos (ou de identificação) é uma estrutura de dados fundamental para permitir que a identificação ocorra em um único passo (no máximo dois, para linguagens que permitem referências para a frente). Ela serve para armazenar todos os nomes declarados e seus respectivos atributos. A busca é sempre feita na tabela.

   - Quais as operações básicas que devem ser prevista pela tabela de símbolos para suportar a identificação de linguagens com estrutura de blocos aninhados?

     Inserção de nomes (enter), busca de nomes (retrieve), início de escopo (open) e fim de escopo (close).

   - Que tipo de erros são reportados durante a subfase de identificação?

     Nomes não declarados e nomes declarados mais de uma vez no mesmo bloco.

2. (1.5 ponto) Responda:

   - Em que consiste a subfase de verificação de tipos durante a fase de análise de contexto?

     Consiste na verificação da adequação dos tipos dos operandos aos tipos requeridos pelos operadores utilizados.

   - De que maneira estão relacionadas as subfases de identificação e de verificação de tipos?

     A subfase de verificação de tipos depende da subfase de identificação, pois é através da vinculação do nome com a respectiva declaração que se obtém o tipo do operando e se pode efetuar a verificação da assinatura da operação.

   - Que tipo de erros são reportados durante a subfase de verificação de tipos?

     Tipos incompatíveis para a operação em questão.

3. (1.5 ponto) Sejam *address (A)=1000*, *size (integer)=2*, *size (boolean)=1* e *size (real)=4*. Considere o seguinte trecho de programa Pascal:

```
var A: array [1..10] of
       record
             B: integer;
             C: array [2..5,-3..8] of
                   record
                         E: real;
                         F: integer;
                   end;
             D: boolean;
       end;
```

Determine:

- *size (A)*;

  *size (A) = 10 * size (record B,C,D)*
  size (record B,C,D) = size (B) + size (C) + size (D) = 2 + size (C) + 1
  *size (C) = 4 * 12 * size (record E,F) = 48 * 6 = 288*
  *Portanto, size (A) = 10 * 291 = 2910.*

- *address (A[i].D)*;

  *address (A[i].D) = 1000 + (i-1) * 291 + 290 =*
  *999 + 291*i*

- *address (A[i].C[j,k].F).*

  *address (A[i].C[j,k].F) = 1000 + (i-1) * 291 + 2 + (j-2) * 12 * 6 + (k+3) * 6 + 4 =*
  *1000 + 291*i - 291 +2 + 72*j - 144 + 6*k + 18 + 4 =*
  *589 + 291*i + 72*j + 6*k*

4. (1 ponto) Compare os modelos de máquina de pilha e máquina de registradores com relação às vantagens e desvantagens que cada um oferece quando se trata de gerar código para avaliação de expressões.

   A geração de código para máquinas de registradores geralmente produz um código mais rápido e compacto do que a geração de código para máquina de pilha. No entanto, ela demanda um gerenciamento rigoroso sobre a alocação e o uso de cada registrador durante a compilação do programa-fonte, o que torna essa fase mais complexa do ponto de vista da implementação. O uso de registradores, no entanto, facilita a otimização de código dependente de máquina, como por exemplo o reaproveitamento do conteúdo previamente carregado nos registradores. A geração de código para máquina de pilha é mais simples mas produz código de qualidade inferior.

5. (1 ponto) Descreva a estrutura de um frame típico, utilizado no ambiente de execução de uma linguagem de programação estruturada em blocos. Explique o que é e para que serve cada um dos campos que fazem parte dele.

   O frame comporta as variáveis locais de cada bloco (acessadas através de offsets positivos em relação ao LB) e também as seguintes informações:
   - *Link dinâmico: serve para armezenar o LB do frame anterior;*
   - *Link estático: serve para armazenar o LB do frame mais recentemente ativado e que envolve, estaticamente, o bloco correspondente ao frame corrente. Serve para acessar variáveis não-locais.*
   - *Endereço de retorno: serve para armazenar o endereço da instrução que deve ser executada no retorno no bloco correspondente ao frame corrente.*
   - *Caso o bloco corrente seja um procedimento ou função parametrizado, os parâmetros são acessados através de offeset negativo em relação ao LB.*

   A base do frame é referenciada por um ponteiro fixo durante toda a execução do bloco, denominado LB. O topo do frame é apontado por um registrador denominado SB, que flutua durante a avaliação de expressões.

6. (2 pontos) Suponha que um endereço ocupe dois bytes e sejam *size (integer)=2, size (boolean)=1* e *size (real)=4* e a seguinte estrutura de um programa Pascal:

```
program P;
      var A,B: integer;
      function Q (N: boolean): real;
            var E,F,G: boolean;
            begin
            ...
```

```
      end;
procedure R (M: real);
      var C,D: real;
      procedure S ():
            var E: real;
            begin
            ...
            end;
      begin
      ...
      end;
begin
...
end.
```

- Considere o fluxo de execução P → R → S e determine o endereço de cada uma das variáveis e parâmetros visíveis (deslocamento + registrador) nessa situação;

```
E: 6[LB]
C: 6[L1]
D: 10[L1]
M: -4[L1]
A: 0[SB]
B: 2[SB]
```

- Considere o fluxo de execução P → R → S e determine a configuração da pilha de execução nessa situação.

```
A(2)
B(2)
** Frame R **
M(4)
LD (vazio ou SB)
LE (vazio ou SB)
ER
C(4)
D(4)
** Frame S **
LD(R)
LE(R)
ER
E(4)
```

- Considere o fluxo de execução P → R → S → S → Q e determine o endereço de cada uma das variáveis e parâmetros visíveis (deslocamento + registrador) nessa situação;

```
E: 6[LB]
F: 7[LB]
G: 8[LB]
N: -1[LB]
A: 0[SB]
B: 2[SB]
```

- Considere o fluxo de execução P → R → S → S → Q e determine a configuração da pilha de execução nessa situação.

```
A(2)
B(2)
** Frame R **
```

```
M(4)
LD (vazio ou SB)
LE (vazio ou SB)
ER
C(4)
D(4)
** Frame S1 **
LD(R)
LE(R)
ER
E(4)
** Frame (S2) **
LD(S1)
LE(R)
ER
E(4)
** Frame (Q) **
N(1)
LD(S2)
LE(vazio ou SB)
ER
E(1)
F(1)
G(1)
```

7. (1 ponto) Considere o conjunto de instruções da máquina virtual TAM. Com base nas funções e nos templates de código estudados em sala de aula, mostre o código que seria gerado para o seguinte trecho de programa:

```
if a>b then
      while c<=(d+1) do
            begin
            c:=c+3;
            d:=e+f*2
            end
else
      repeat
      a:=b+a+c
      until a>(2*b)
```

execute (

```
      if a>b then
            while c<=(d+1) do
                  begin
                  c:=c+3;
                  d:=e+f*2
                  end
      else
            repeat
            a:=b+a+c
            until a>(2*b)
```

) =

evaluate (a>b)
JUMPIF (0) L1
execute (

```
            while c<=(d+1) do
                  begin
                  c:=c+3;
                  d:=e+f*2
```

```
                    end
)
JUMP L2
L1: execute (
            repeat
            a:=b+a+c
            until a>(2*b)
)
L2:

=

LOAD a
LOAD b
MAIOR
JUMPIF (0) L1
evaluate (c<=(d+1))
JUMPIF (0) L3
execute (
                    c:=c+3;
                    d:=e+f*2
)
L3:
JUMP L2
L1:
L4:
execute (
            a:=b+a+c
)
evaluate (a>(2*b))
JUMPIF (0) L4
L2:

=

LOAD a
LOAD b
MAIOR
JUMPIF (0) L1
LOAD C
LOAD D
LOADL 1
CALL ADD
MENORIGUAL
JUMPIF (0) L3
LOAD c
LOADL 3
CALL ADD
STORE c
LOAD e
LOAD f
LOADL 2
CALL MULT
CALL ADD
STORE d
L3:
JUMP L2
L1:
L4:
LOAD b
LOAD a
```

```
CALL ADD
LOAD c
CALL ADD
STORE a
LOAD a
LOADL 2
LOAD b
CALL MULT
CALL MAIOR
JUMPIF (0) L4
L2:
```