FEUP Universidade do Porto
Faculdade de Engenharia

Mestrado em Engenharia Informática
Paradigmas de Programação
2004/2005

Exame de Avaliação

27 de Janeiro de 2005

Resolução do Exame

1. Descrição sintáctica de linguagens [10 pontos]

Considere a linguagem de expressões de números inteiros não negativos, com os seguintes operadores, como na linguagem Pascal:

```
< <= = <> >= > in
+ - or
* / div mod and
not
```

Os operadores na mesma linha têm a mesma precedência e as linhas estão ordenadas por ordem de precedência. Os operadores apresentados são associativos à esquerda.

a) Escreva uma gramática EBNF (Extended Backus-Naur Form) para esta linguagem.

Resposta:

```
      <expr> ::= <expr-arith> { <op-rel> <expr-arith> }
<expr-arith> ::= <termo> { <op-sum> <termo> }
     <termo> ::= <factor> { <op-mult> <factor> }
    <factor> ::= <int> | ( <expr> ) | not <factor>
    <op-rel> ::= < | <= | = | <> | >= | > | in
    <op-sum> ::= + | - | or
   <op-mult> ::= * | / | div | mod | and
       <int> ::= <digit> { <digit> }
     <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

b) Estenda a sua gramática para tratar expressões precedidas por sinal menos. Serão permitidas expressões -23 ou -(a-b), mas não 26+-4.

Resposta:

```
<expr-arith> ::= [-] <termo> { <op-sum> <termo> }
```

2. Instruções e Programação Estruturada [15 pontos]

Considere a construção para valores inteiros:

for i:= *E1* **to** *E2* **do** *SL* **end**

a) Usando condicionais e gotos em vez de `for`, reescreva a construção assegurando-se que i não tome valores maiores do que o valor inicial de E2.

Resposta:

```
    i := E1
L1: if i <= E2 then
        SL
        i := i + 1
        goto L1
```

b) Usando atribuição, novas variáveis e condicionais `while` e `repeat` em vez de `for`, reescreva a construção assegurando-se que i não tome valores maiores do que o valor inicial de E2.

Resposta:

```
i := E1
while i <= E2
    repeat
        SL
        i := i + 1
    until i > E2
```

3. Tipos e Representação de Dados [15 pontos]

Considere a discussão sobre tipos (por exemplo em [Sethi, p. 142]) e escolha uma linguagem de programação imperativa (C ou Pascal).

a) Explique porque é que "static typing" é diferente de "strong typing".

Resposta:

Um sistema de tipos é "forte" se aceita apenas expressões "seguras", i.e., livres de erros de tipos.

Um erro de tipo ocorre quando uma expressão tem um tipo que não é equivalente ao tipo esperado.

Verificação estática ocorre em tempo de compilação por oposição a verificação de tipos dinâmica só ocorre em tempo de execução.

b) Determine se a linguagem da sua escolha usa equivalência de nomes ou estrutural ou um híbrido das duas.

Resposta:

Por exemplo em C [Sethi p. 141] usa equivalência estrutural para todos os tipos com excepção das estruturas (record).

As estruturas têm um nome que é tratado como tipo (para evitar tipos circulares).

c) Analise o sistema de tipos dessa linguagem e determine se é forte ("strong") ou não.

Resposta:

Em ANSI C o sistema de tipos só aceita expressões seguras, i.e., que avaliam sem erros de tipos, logo é "forte".

Por exemplo, quando uma função espera um argumento do tipo T1 e é aplicada com um argumento que não é equivalente a T1, o sistema de tipos do C reporta um erro.

4. Procedimentos e activação [15 pontos]

Considere o procedimento my_swap declarado da seguinte forma:

```
procedure my_swap (x, y : integer);
   procedure f ( ) : integer;
      var z : integer;
   begin (* f *)
      z := x; x := y; return z
   end f;
begin (* my_swap *)
   y := f ( );
end my_swap;
```

Descreva o efeito da chamada

```
my_swap (i, A [i])
```

se forem usados os métodos de passagem de parâmetros indicados.

a) Call-by-value

Resposta:

Em "call-by-value" o valor do parâmetro actual é copiado para o parâmetro formal.

```
x fica com o valor de i; y fica com o valor de A[i]
z fica com o valor de x (logo i);  x fica com o valor de y (logo A[i])
y fica com o valor de i
```

Após terminar `my_swap i` **e** `A[i]` **retêm os valores originais.**

b) Call-by-reference

Resposta:

Em "call-by-reference" o parâmetro torna-se um sinónimo para a localização do parâmetro actual. parâmetro formal.

```
x fica com a localização de i; y fica com a localização de A[i]
z fica com o valor de x (logo i); x (logo i) fica com o valor de y (logo A[i])
y fica com o valor de i
```

Após terminar `my_swap A[i]` **retém o valor original e** `i` **fica com o valor de** `A[i]`

c) Call-by-value-result

Resposta:

Em "call-by-value-result" os actuais são inicialmente copiados para os formais, as suas localizações são guardadas e os formais são copiados de volta para os actuais.

Neste caso, como não há sinónimos (aliases) não há diferença em relação ao caso anterior e `i` **fica com o valor de** `A[i]` **quando** `f` **termina.**

5. Programação orientada por objectos [15 pontos]

Considere a seguinte implementação de uma stack em C++:

```
(1)    struct  Stack {
(2)       int  top;
(3)       char elements[101];
(4)
(5)       char pop();
(6)       void push(char);
(7)            Stack()     { top = 0 }
(8)    };
(9)    char Stack::pop() {
(11)      top = top - 1;
(12)      return elements[top+1];
(13)   }
(14)   void Stack::push(char c) {
(16)      top = top + 1;
(17)      elements[top] = c;
(18)   }
(19)   #include <stdio.h>
(20)   main() {
(21)      Stack s;
(22)      s.push('F'); s.push('C'); s.push('P');
(23)      printf(``%c %c %c\n'', s.pop(), s.pop(), s.pop());
(24)   }
```

a) Complete o programa por forma a verificar os limites do array usado na implementação.

Resposta:

Pré-condições: só pode fazer pop se top¿0; só pode fazer push se top¡101

```
(0)     #define LIMIT 101
(3)         char elements[LIMIT];
(10)    if (top == 0) then throw LimitException();
(15)    if (top == LIMIT) then throw LimitException();
```

b) Escreva e junte à classe `Stack` o método `empty()` para avaliar se a stack está ou não vazia.

Resposta:

```
(19)   bool Stack::empty () {
(20)      return top == 0;
(21)   }
```

6. **Programação funcional tipada [15 pontos]**

Considere a seguinte gramática simplificada para expressões aritméticas: $\langle expr \rangle$ é o não-terminal; +, *, x, y, z, (,) são os terminais; +, * são operadores; e x, y, z são variáveis.

```
<expr>  ::=  ( +  <expr>  <expr> )
           |  ( *  <expr>  <expr> )
           |  x | y | z
```

a) Escreva uma função `counter` em Scheme (ou ML) que determina o número total de ocorrências de um operador numa expressão gerada pela gramática apresentada. Pode assumir que `counter` é invocado sempre com expressões válidas.

Por exemplo

```
(counter '(+ (* x y) (+ z y) ) )
```

devolve 3

Resposta:

```
(define (counter e)
  (cond ((eq? 'x e) 0)
        ((eq? 'y e) 0)
        ((eq? 'z e) 0)
        (else (+ 1 (counter (cadr e)) (counter (caddr e))))
  )
)
(counter '(+ (* x y) (+ z y)))
```

b) Escreva uma função `evaluate` em Scheme (ou ML) que determina o valor de uma expressão válida.

Por exemplo, considerando um contexto em que que x, y e z são 1, 2 e 0, respectivamente:

```
(evaluate '(+ (* x y) (+ z y) ) )
```

devolve 4

Resposta:

```
(define (evaluate e)
  (cond ((eq? 'x e) 1)
        ((eq? 'y e) 2)
        ((eq? 'z e) 0)
        ((eq? '+ (car e)) (+ (evaluate (cadr e)) (evaluate (caddr e))))
        ((eq? '* (car e)) (* (evaluate (cadr e)) (evaluate (caddr e))))
        (else 0)
        )
  )
(evaluate '(+ (* x y) (+ z y)))
```

7. **Programação lógica [15 pontos]**

Considere a relação `member` definida pelas regras:

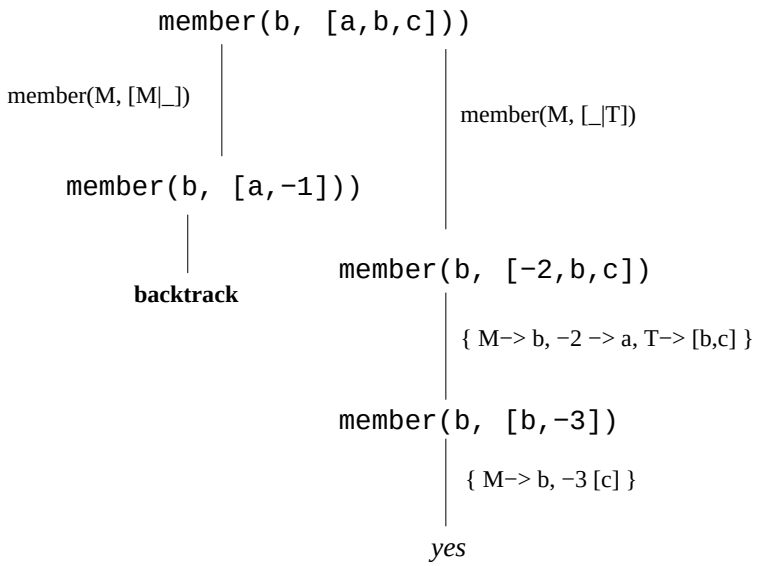
```
member(M, [M|_]).
member(M, [_,T]) :- member(M, T).
```

a) Desenhe a porção da árvore de pesquisa do Prolog que corresponde à seguintes interacção:

```
member(b, [a,b,c]).
```

yes

Resposta:



b) Indique qual a resposta do interpretador de Prolog quando se procuram todas as soluções de `member(b, X)`.

Resposta:

```
X = [b|_] ;
X = [_,b|_] ;
X = [_,_,b|_] ;
...
```

FIM.