Compiladores
Prof. Marcus Ramos
Prova 2 – 20/06/2024

- *Alunos atrasados só podem entrar na sala enquanto ninguém sair; depois que o primeiro sair ninguém mais entra.*
- *As filas são alternadas na medida do possível.*
- *As carteiras de uma mesma fila são alternadas na medida do possível.*
- *É proibido sair da sala e retornar, qualquer que seja o motivo (banheiro inclusive, que deve ser usado antes da prova).*
- *A prova pode ser feita com lápis ou caneta.*
- *As questões podem ser respondidas em qualquer ordem.*
- *O material (borrachas, apontadores etc) são de uso exclusivo de cada um e não podem ser compartilhados.*
- *Os celulares devem permanecer desligados.*
- *A prova começa às 10:00h e termina às 12:00h.*
- *Cadernos e livros devem ser fechados e guardados.*
- *A folha de enunciados pode ser levada depois da prova, desde que não contenha resposta à qualquer das questões.*

1. (2 pontos) Por que a subfase de identificação é a mais crítica do compilador em termos de tempo de execução? O que pode ser feito para reduzir este tempo?

   Se a Tabela de Símbolos for implementada com busca linear, então um programa com n identificadores demanda tempo $n * O(n)$, ou $O(n^2)$ para a sua completa identificação. Como as outras fase do compilador executam em $O(n)$, segue que a subfase de identificação é a mais crítica em termos de tempo. Para reduzir este tempo pode-se representar a Tabela de Símbolos como uma árvore, uma tabela de hash etc. O tempo de busca pode então ser reduzido para até perto de $O(n)$, torando o tempo de identificação similar aos tempos das demais fases.

2. (1 ponto) Qual é o número mínimo de bits necessário para representar um tipo com n valores? Quantos valores podem ser representados com um tipo que usa n bits?

   - n valores necessitam de pelo menos $log_2 n$ bits para a sua representação.
   - Com n bits é possível representar 2^n valores distintos.

3. (2 pontos) O que é, quando pode ser usado e quais as vantagens do uso da origem virtual de um agregado homogêneo?

   (i) Origem virtual (diferentemente da origem real) é um endereço de memória onde o agregado teria início caso o índice do elemento inicial fosse zero.
   (ii) Pode ser usado quando o endereço $base$ do agregado é conhecido em tempo de compilação.

(iii) A vantagem está no fato de que ao invés de ter que fazer uma adição, uma multiplicação e um subtração para cada indexação, pode-se fazer apenas a adição e a multiplicação. É útil em linguagens onde o índice inicial pode ser diferente de zero (ex: Pascal).

(iv) Exemplo: `var v [2..10] of integer;`

$address\ (v[i]) = base + size(integer) * (i - 2)$, ou

$address\ (v[i]) = base + size(integer) * i + size(integer) * (-2)$, ou

$address\ (v[i]) = base' + size(integer) * i$

onde $base' = base - 2 * size(integer)$

$base$ é a origem real,

$base'$ é a origem virtual.

4. (1 ponto) Suponha que um subprograma (procedimento ou função) está sendo executado enquanto transcorre o tempo de vida de uma variável. Esta variável pode ser acessada pela função?

   Depende. Tempo de vida não tem nada a ver com escopo. Se a variável for visível (conforme as regras de escopo da linguagem), então ela pode ser acessada. Caso contrário, não.

5. (2 pontos) Quando acontece a alocação estática, a alocação automática e a alocação dinâmica? Como são endereçadas as variáveis em cada uma destas formas de alocação?

   - Alocação estática: do início da execução do programa até o término da execução do programa. Acesso via deslocamento + SB.
   - Alocação automática: do início da execução do bloco (procedimento ou função) até o término da execução do bloco. Acesso via deslocamento + LB (se local) ou deslocamento + L_i (com i>=1) (se não local).
   - Alocação dinâmica: do momento em que o comando de alocação é emitido até o momento em que o comando de desalocação é emitido (ou o *garbage collector* entre em ação). Por meio de ponteiro alocado estaticamente, automaticamente ou dinamicamente.

6. (2 pontos) Mostre o código gerado para o trecho de programa apresentado abaixo na linguagem de baixo nível TAM (suponha que todas as variáveis são inteiras).

```
while (a+1!=b) do {
    if (c>d*2) then c=d; else d=c;
    if (e==f) then e=e+f*3-4;
    }
```

   Funções de código:

   execute [while <exp> do <com>] =

   JUMP h

```
        g:      execute [<com>]
        h:      evaluate [<exp>]
                JUMPIF (1) g
```

execute [if <exp> then <com1> else <com2>] =

```
                evaluate [<exp>]
                JUMP (0) g
                execute [<com1>]
                JUMP h
        g:      execute [<com2>]
        h:
```

execute [if <exp> then <com>] =

```
                evaluate [<exp>]
                JUMP (0) g
                execute [<com>]
        g:
```

Resultado parcial:

```
             execute [while <exp> do {<com1>;<com2>}] =
             JUMP      L_1
        L_2: execute   [{<com1>;<com2>;}]
        L_1: evaluate  [a+1!=b]
             JUMPIF(1) L_2

             execute   [{<com1>;<com2>;}] =
             execute   [<com1>]
             execute   [<com2>]

             execute   [<com1>] =
             evaluate  [c>d*2]
             JUMPIF(0) L_3
             execute   [c=d]
             JUMP      L_4
        L_3: execute   [d=c]
        L_4:

             execute   [<com2>] =
             evaluate  [e==f]
             JUMPIF(0) L_5
             execute   [e=e+f*3-4]
        L_5:

             evaluate  [a+1!=b] =
             LOAD      a
             LOADL     1
             CALL      add
             LOAD      b
```

```
        CALL    neq

        evaluate  [c>d*2] =
        LOAD    c
        LOAD    d
        LOADL   2
        CALL    mult
        CALL    gt

        evaluate  [e==f] =
        LOAD    e
        LOAD    f
        CALL    eq

        execute   [c=d] =
        LOAD    d
        STORE   c

        execute   [d=c] =
        LOAD    c
        STORE   d

        execute   [e=e+f*3-4] =
        LOAD    e
        LOAD    f
        LOADL   3
        CALL    mult
        CALL    add
        LOADL   4
        CALL    sub
```

Resultado final:

```
        JUMP    L_1
L_2:    LOAD    c
        LOAD    d
        LOADL   2
        CALL    mult
        CALL    gt
        JUMPIF(0) L_3
        LOAD    d
        STORE   c
        JUMP    L_4
L_3:    LOAD    c
        STORE   d
L_4:    LOAD    e
        LOAD    f
        CALL    eq
        JUMPIF(0) L_5
        LOAD    e
        LOAD    f
```

```
        LOADL     3
        CALL      mult
        CALL      add
        LOADL     4
        CALL      sub
L_5:
L_1:    LOAD      a
        LOADL     1
        CALL      add
        LOAD      b
        CALL      neq
        JUMPIF(1) L_2
```