Aluno: Carlos Victor Martins da Silva

Matrícula: 476516

Link para o arquivo Atividade1.oz: [PasteBin](https://pastebin.com/5b9NbTya) / [Google Drive](https://drive.google.com/file/d/1IoP3z82L-sEh9rkOyw2MXZstePX-wqho/view?usp=sharing)

| **Questões feitas** | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| a. X  b. X | a. X  b. X |  | X | X | a. X  b. X | X | X |  | a. X  b. X  c. X |

1. A Seção 1.1 usa o sistema como calculadora. Vamos explorar as possibilidades:

(b) … Existem atalhos possíveis neste caso?

Resposta:

Ao evitar o uso de chamadas recursivas na implementação pela limitação de não poder ser definida novas funções e se distanciando um implementação mais direta da noção matemática ao substituir a recursão por um loop condicional for reduzimos o consumo de memória e aceleramos um pouco a execução dos cálculos já que não é necessário mais utilizar a pilha de execução ou executar novas chamadas da função até chegar ao ponto de retorno.

4. O que a Seção 1.7 diz sobre programas cuja complexidade de tempo é um polinômio de alta ordem? Eles são práticos ou não? O que você acha?

Resposta:

“Programs whose time complexity is a low-order polynomial are practical.”. Programas cuja a complexidade é de alta ordem podem ou não ser práticos, dependendo da forma com que o algoritmo será implementado, como exemplificado nesta sessão com o uso das funções Pascal e FastPascal, uma simples mudança, ao invés de fazer cada chamada das funções ShiftLeft e ShiftRight recalcular todos os valores anteriores até o valor chamado, armazenar ele temporariamente em uma variável dentro da recursão

5. A seção 1.8 define a Lazy Function Ints, que calcula “preguiçosamente” uma lista infinita de números inteiros. Vamos definir uma função que calcula a soma de uma lista de inteiros:

fun {SumList L}

case L of X|L1 then X+{SumList L1}

else 0 end

end

➤ O que acontece se chamarmos {SumList {Ints 0}}? Isso é uma boa ideia?

Resposta:

Não, essa não é uma boa ideia, pois ao chamar essa função, o programa entrará num loop infinito já que a função Lazy ter sido passada como argumento de uma função não Lazy irá continuar a construir uma lista com todos valores inteiros de 0 ao infinito (∞) para serem somados pela função SumList, inevitavelmente, dado um certo tempo, o interpretador do ambiente Mozart irá estourar o limite de memória alocado para sua máquina virtual com esses valores e irá encerrar a execução.

7. Este exercício compara variáveis e células. Fornecemos dois fragmentos de código. O primeiro usa variáveis:

… Nível 0

local X in Nível 1

X=23

local X in

X=44 Nível 2

end

{Browse X} Nível 1

end

… Nível 0

O segundo usa uma célula:

local X in

X = {NewCell 23}

X := 44

{Browse @X}

end

No primeiro, o identificador X se refere a duas variáveis diferentes. No segundo, X se refere a uma célula. O que Browse exibe em cada fragmento? Explicar.

Resposta:

No 1° fragmento de código o valor exibido é 23 pois o identificador de variável X lida pela função Browse está contida no primeiro nível do escopo onde o identificador de variável X está definido como 23, caso o Browse tivesse sido chamado dentro do bloco de 2° nível o valor escrito no Oz Browser seria igual a 44. Essa diferenciação ocorre pois com o uso do "local X” é criado um novo identificador X cujo o acesso e definição só é possível dentro do bloco definido após ele, caso seja usado novamente (“local X”) dentro do bloco será criado um novo bloco cuja terá um novo identificador X cuja o valor só é possível ser acessado e definido dentro desse novo bloco e assim sucessivamente.

Já no 2° fragmento de código o valor exibido é 44 pois a variável X aponta para uma Cell que, diferente dos identificadores de variáveis inicializadas diretamente utilizando valores na linguagem Oz que são imutáveis, uma Cell é mutável e pode ser acessada pela função Browse sem problemas já que estão em um mesmo bloco de código e indentação (local X in … end) junto com a redefinição feita (X := 44).

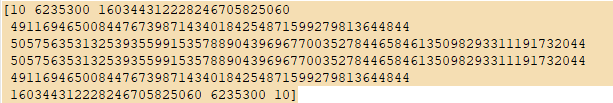
6. A seção 1.9 explica como usar a programação de ordem superior para calcular variações no triângulo de Pascal. O objetivo deste exercício é explorar essas variações.

(a) Calcule linhas individuais usando subtração, multiplicação e outras operações. Por que usar a multiplicação dá um triângulo com zeros? Tente o seguinte tipo de multiplicação:

fun {Mul1 X Y} (X+1)\*(Y+1) end

Qual é a lista da 10ª linha quando calculada com Mul1?

Resposta:



8. Este exercício investiga como usar células junto com funções. Vamos definir uma função {Accumulate N} que acumula todas as suas entradas, ou seja, adiciona todos os argumentos de todas as chamadas. Aqui está um exemplo:

{Browse {Accumulate 5}}

{Browse {Accumulate 100}}

{Browse {Accumulate 45}}

Deve exibir 5, 105 e 150, supondo que o acumulador contenha zero no início. Esta é uma maneira errada de escrever Accumulate:

declare

fun {Accumulate N}

Acc in

Acc={NewCell 0}

Acc:=@Acc+N

@Acc

end

O que há de errado com essa definição? Como você corrigiria isso?

Resposta:

A cada chamada uma nova célula é definida com o valor 0, sempre sendo modificada para receber o valor passado como argumento mas nunca armazenada em nenhum lugar onde poderá ser acessada novamente em uma possível futura chamada e assim garantir que o comportamento esperado aconteça. Para corrigir esse problema bastaria mover a definição do identificador Acc para fora da função ao mesmo tempo que garantisse o acesso dela para a função consiga incrementá-la com o valor passado como argumento. Uma implementação corrigida está incluída no arquivo “Atividade1.oz”.

10. A seção 1.15 dá um exemplo usando uma célula para armazenar um contador que é incrementado por dois threads.

(a) Tente executar este exemplo várias vezes. Que resultados você consegue? Você já obteve o resultado 1? Por que isso pode ser?

Resposta:

O resultado variou entre 0, 1 e 2. Sim. O motivo para isso é devido a forma com que a linguagem Oz funciona, a linguagem tem uma natureza não determinística e com o uso de threads independentes o resultado varia conforme a ordem de execução que for acontecendo, isso é o que gera inconsistências como nesse caso, por algumas vezes a função Browse retornou 0 pois nenhuma das threads havia tido tempo para modificar a célula C, em outras retornou 1 por apenas uma das threads ter conseguido executar suas instruções e modificar a celular indicada por C e retornou 2 quando ambas as threads tiveram tempo para executar suas operações antes de Browse ser chamado.

(b) Modifique o exemplo adicionando chamadas para Delay em cada thread. Isso altera a intercalação do segmento sem alterar os cálculos que o segmento faz. Você pode conceber um esquema que sempre resulte em 1?

Resposta:

Para que o resultado dê 1 basta garantir, utilizando a função Delay, que uma thread seja executada antes da função Browser que deve ser executada antes da outra thread seja executada tbm. Ou seja, basta colocar um atraso em uma das threads e um atraso um pouco maior após a chamada da função Browse.

\* Código exemplo também presente no arquivo Atividade1.oz

declare C = {NewCell 0}

thread I in

{Delay 100}

I=@C

C:=I+1

end

thread J in

J=@C

C:=J+1

end

{Delay 10}

{Browse @C}

(c) A seção 1.16 fornece uma versão do contador que nunca dá o resultado 1. O que acontece se você usar a técnica de retardo para tentar obter um 1 de qualquer maneira?

Resposta:

O resultado continuará a ser 2 porém haverá uma demora maior para que o resultado seja escrito no Oz Browser, o motivo para isso é por conta do uso de travas nas threads garantindo que uma thread não irá executar código antes que a outra finalize seu trabalho e assim evitando inconsistências.