Lista de Exercícios 16
Semântica Formal

Exercício 01) Na linguagem Three, para qualquer variável `x` e quaisquer expressões E_1 e E_2, a expressão `let val x = E1 in E2 end` é equivalente a expressão `(fn x => e2) e1`.

a) Usando a versão com escopo dinâmico da semântica natural de Three, mostre que as expressões mencionadas são de fato equivalentes. Dica: mostre que as condições para avaliar a expressão $\langle apply(fn(x, e_2), e_1), C\rangle \rightarrow v$ são equivalentes às condições necessárias para avaliar a expressão $\langle let(x, e_1, e_2), C\rangle \rightarrow v$.
b) Mostre a mesma coisa, mas desta vez usando o interpretador que implementa o escopo estático.

Note que os resultados anteriores mostram que é possível implementar a construção `let` em termos de definição de funções e aplicações. Portanto, o nó que representa `let` na AST poderia ser dispensado desta linguagem tornando assim os interpretadores de Prolog e semântica natural mais simples.

Exercício 02) O objetivo deste exercício é definir a linguagem Four que é uma extensão da linguagem Three. Um exemplo de programa nesta nova linguagem é dado a seguir.

```
let
  val fact = fn x => if x < 2 then x else x * fact (x - 1)
in
  fact 5
end
```

Observe que a linguagem Four adiciona três novas construções: o operador `<` para comparações, o operador `-` para subtrações e a expressão condicional `if-then-else`. O programa de exemplo define `fact` como uma função anônima recursiva e a utiliza para o calcular o fatorial de 5.

a) Define a sintaxe da linguagem Four a partir da linguagem Three, isto é, estenda sua sintaxe para incorporar estas três novas construções. Faça a nova gramática em formato BNF e cuide para que ela não seja ambígua.
b) Defina três novos tipos de nós AST para incorporar a nova sintaxe. Estenda a implementação do interpretador em Prolog para lidar com esses novos nós. Use a versão da implementação com escopo dinâmico, já que a implementação com escopo dinâmico não consegue lidar com definições recursivas. Note que o programa de exemplo não é válido em ML. Verifique que a implementação avalie programas corretamente.
c) Dê uma semântica natural para a linguagem Four.

Exercício 03) A semântica com escopo estático da linguagem Three não suporta recursão porque o escopo de uma definição de uma variável não inclui o corpo da função. ML funciona do mesmo jeito. Já o escopo de uma definição de `f`, que é produzido por `fun f ...`, inclui o corpo da função sendo definida. Assim, em ML, apenas funções declaradas com `fun` podem ser recursivas.

a) Estenda a sintaxe da linguagem Four do exercício anterior para que esta permita definições simples de funções como por exemplo a definição de `f` a seguir: `let fun f x = x+1 in f 1 end`.
b) Defina os novos tipos de nós da AST para incorporar a sintaxe estendida. Estenda o interpretador em Prolog da linguagem Four para lidar com eles. Use agora a implementação com escopo estático para ambos tipos de funções (`fun f ...` e `val f = fn ...`). Funções declaradas com a construção `fun` devem suportar recursão. Dica: use um termo diferente para representar funções criadas com `fun`, um termo que armazene o nome da função e uma nova cláusula para `apply`. Teste a implementação usando uma versão recursiva da função fatorial definida como `fun`.
c) Dê uma semântica natural para esta linguagem estendida.

Exercício 04) Ambas as versões da linguagem Three, com escopo dinâmico e escopo estático, usam passagem de parâmetros por valor. Elas avaliam cada parâmetro antes da chamada da função.

a) Implemente um interpretador em Prolog para esta linguagem que use escopo estático e passagem de parâmetros por nome. Dica: embora seja difícil de testar isto em linguagens sem efeito colateral, ao menos é possível verificar experimentalmente que seu interpretador não avalia o parâmetro atual se o parâmetro formal correspondente nunca for usado.
b) Dê uma semântica natural para esta linguagem.

Exercício 05) Considere a linguagem After que sua sintaxe definida à esquerda e uma descrição de suas operações à direita.

```
<exp> ::= <exp> > <difexp> | <exp>
<difexp> ::= <difexp> - <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= after <stmt> get <exp>
                  | ( <exp> )
                  | <variable>
                  | <constant>
<stmt> ::= <compound-stmt>
               | <while-stmt>
               | <assignment-stmt>
<compound-stmt> ::= { <stmt-sequence> }
<stmt-sequence> ::= <stmt-sequence> <stmt>
                        | <stmt>
<while-stmt> ::= while ( <exp> ) stmt;
<assignment-stmt> ::= <variable> := <exp> ;
```

- Todos os valores manipulados são inteiros
- O valor de `a>b` é `1` se `a` for maior que `b`, `0` caso contrário
- O valor de `a-b` é a diferença de `a` menos `b`
- O valor de `a*b` é a multiplicação de `a` vezes `b`
- O valor de `after s get e end` é dado pela expressão e avaliada no ambiente final após executar `s` no ambiente atual; mudanças feitas por `s` não são visíveis fora de `after`
- Em comandos compostos, mudanças no ambiente são cumulativas; cada comando composto vê a alteração do ambiente do comando anterior
- O comando `while (e) s` avalia a guarda `e`, caso ela seja `0` nada acontece, caso ela não seja `0`, o corpo `s` é executado e toda a ação é repetida
- Uma atribuição `v := e` altera a variável `v` no ambiente atual ou a introduz se for a primeira vez vista

Por exemplo, a expressão a seguir em After calcula o fatorial de 5.

```
01: after {                  05:     sofar = sofar * n;
02:   n = 5;                 06:     n = n - 1;
03:   sofar = 1;             07:   }
04:   while (n > 0) {        08: } get sofar end
```

a) Defina os tipos de nós da AST necessários. Dica: para dois comandos compostos crie um nó `seq(X,Y)`; para cadeias maiores pode-se ser usar `seq(X,seq(Y,Z))`. Implemente um interpretador em Prolog para ela.
b) Dê uma semântica natural para After.