

Trabalho Final de Linguagens de Programação 2

2005-2006

6 de Dezembro de 2005

versão 1.0 — 22 de Novembro de 2005.

versão 1.1 — 25 de Novembro de 2005: Adicionada a syntax para a linguagem tipificada.

versão 1.2 — 6 de Dezembro de 2005: Adicionadas declarações globais múltiplas, definição das comparações de valores, e alguns exemplos

Resumo

Este documento contém as especificações técnicas, os pormenores de avaliação e as datas do trabalho final da cadeira de Linguagens de Programação 2 (LP2), edição 2005 - 2006.

Conteúdo

1	Introdução	2
2	Avaliação	2
3	Descrição técnica	3
4	Compilação	5
5	Sistema de tipos	5
6	Estrutura do relatório	6
7	Entregáveis	6
8	Datas importantes	6
A	Exemplos de teste	7

1 Introdução

O objectivo base deste trabalho é construir um interpretador e um compilador de uma linguagem imperativa com funções, procedimentos e registos. Os dados para o trabalho, apresentados neste enunciado, são a sintaxe concreta da linguagem e uma breve descrição da semântica das construções. São também especificados neste documento, ou em versões posteriores do mesmo, os prazos para entrega e avaliação do trabalho, quais os elementos que devem ser entregues, qual a estrutura esperada do relatório, e alguns exemplos que devem ser aceites ou rejeitados.

Nesta edição da cadeira admitem-se várias hipóteses para uma avaliação positiva do trabalho final. A realização de uma vertente base é mandatória, sendo opcional, mediante uma redução considerável do limite máximo da nota, a realização de duas partes avançadas. Estas duas partes podem ser realizadas em conjunto, para a classificação máxima total, ou em separado, para uma classificação máxima intermédia.

Na vertente base exige-se a implementação do **interpretador** da linguagem completa apresentada neste enunciado. O programa entregue deve ser capaz de interpretar **todos** os exemplos dados correctamente, bem como responder correctamente a outros requisitos definidos neste enunciado. Em relação aos trabalhos já desenvolvidos nas aulas práticas, esta vertente exige a implementação de novos tipos de dados e novas construções da linguagem como por exemplo, registos.

Numa segunda vertente, avançada, a linguagem deve ser compilada para a máquina virtual CLR (por tradução na linguagem CIL). Para a implementação do **compilador** devem ser utilizadas as ferramentas fornecidas na implementação Mono da CLR. O desafio que se põe neste caso é a compilação das “closures” resultantes do facto de termos funções e procedimentos como valores da linguagem.

Ainda numa outra extensão avançada, a interpretação e/ou compilação da linguagem podem ser extendidas com um **sistema de tipos** que assegure a boa execução dos programas. Neste caso é a compreensão de novos conceitos que está em jogo.

2 Avaliação

O requisito mínimo para a aprovação na parte prática da cadeira resume-se à correcta implementação do interpretador da linguagem. No entanto, um trabalho que apresente apenas a linguagem base interpretada correctamente tem como classificação máxima de 14 valores, cada uma das opções acima indicadas dá o acesso, de acordo com a tabela abaixo, a uma nota maior, até prefazer 20 valores no seu conjunto.

Trabalho Base:	14 valores
Compilação:	3 valores
Sistema de tipos:	3 valores

É possível portanto fazer a interpretação da linguagem base com qualquer uma das opções sem qualquer prioridade entre elas obtendo assim uma classificação

máxima de 17 valores, ou então realizar as duas sendo o trabalho classificado numa escala de 20 valores.

3 Descrição técnica

3.1 Linguagem Base

A sintaxe concreta da linguagem a ser implementada é a seguinte:

```
C ::= C;C
    | decl id1=E1 ... idn=En in C end
    | E := E
    | E(E1,...,En)
    | while E do C end
    | if E then C else C end
    | if E then C end
    | print(E)
    | println()

E ::= id | num | string | true | false
    | ( E ) | E+E | E-E | E*E | E/E | E%E | -E
    | E and E | E or E | not E
    | E > E | E <= E | E = E | E <> E
    | var(E) | !E
    | [id1=E1, id2=E2, ..., idn=En] | E.id
    | if E then E else E end
    | fun x1,x2,...,xn => E end
    | proc x1,x2,...,xn => C end
    | E(E1,...,En)
    | decl id1=E1 ... idn=En in E end
    | do C return E end
    | readInt()
    | readString()
```

Nota: As funções e procedimentos podem não ter parâmetros. Os registos podem não ter quaisquer campos.

Por questões de implementação do parser, a sintaxe concreta acima pode ser restringida até um certo ponto. No entanto, os exemplos fornecidos em apêndice deverão ser aceites sem qualquer modificação. Por exemplo, tal como no quarto trabalho prático, o comando `f(0)(1)(2)(3)`, entendido como uma chamada a um procedimento resultante das chamadas sucessivas `f(0)(1)(2)`, pode não ser aceite, sendo substituída pelo comando `(f(0)(1)(2))(3)`. O mesmo se passa para uma afectação, onde `f(2) := 3` pode ser substituída por `(f(2)):=3`.

A semântica das construções da linguagem já conhecidas (dos trabalhos realizados) é aquela que foi apresentada quer nas aulas teóricas como nas aulas práticas. No entanto registam-se as seguintes construções novas ou alteradas: o comando `print` deve imprimir o resultado da avaliação da sua expressão interna e não deve mudar de linha, já o comando `println`, deve fazê-lo; A avaliação da expressão `readInt` deve produzir um inteiro a partir da leitura do teclado (`stdin`) e a da expressão `readString` deve produzir um valor tipo `string` a partir da

leitura do teclado (stdin); As declarações nos comandos e expressões `decl` devem ser mutuamente recursivas entre si. Os registos, da forma `[id1=E1, ...]` associam a cada etiqueta um valor que depois pode ser seleccionado pela operação `e.id`.

Os operadores de comparação devem ser válidos para todos os tipos de valores: inteiros, booleanos e cadeias de caracteres. Na sintaxe acima apresentada entende-se por `num` qualquer número inteiro positivo, por `id` qualquer sequência de letras e algarismos começados por uma letra (maiúsculas ou minúsculas); Por `string` entende-se qualquer sequência de caracteres (excepto mudança de linha) delimitados por aspas (""); As constantes `true` e `false` são novos elementos atómicos da linguagem.

Para a construção da árvore de sintaxe abstracta devem ser utilizados dois tipos de nós diferenciados, um para comandos e outro para expressões.

A comparação entre valores (`=` e `<>`) deve ser definida para todos os tipos de valores excepto para funções e procedimentos. A relação de maior deve estar definida apenas para os tipos primitivos: `int`, `bool` e `string`.

3.2 Comandos do interpretador

O interpretador é um programa que deve apresentar ao utilizador uma “prompt” para aceitação de vários comandos de gestão e de programas da linguagem. Os programas podem ser inseridos em várias linhas e devem ser terminados por um símbolo finalizador `;;`. Depois da interpretação de cada comando, o interpretador deve continuar a sua execução com uma “nova prompt”. Para além disso o interpretador deve responder a outros comandos para carregar um ficheiro, para sair, ou para definir uma variável no ambiente global.

Dados de entrada	Descrição
<code>comando</code>	executa o <code>comando</code> , e volta a apresentar a “prompt”.
<code>define id=expressão</code> <code>id=expressão</code> <code>...</code>	Define um conjunto de variáveis globais associando <code>id</code> ao valor dado pela avaliação da <code>expressão</code> . Esta associação é válida em todas as avaliações futuras e as definições são mutuamente recursivas.
<code>load “nome”</code>	Carrega comandos do ficheiro <code>nome</code> , interpreta-os, e volta a apresentar a ”prompt”.
<code>quit</code>	Sai do interpretador.

O interpretador deve ainda aceitar como argumento de linha o nome de um ficheiro para ser interpretado. Neste caso, não apresentará qualquer “prompt”, e quando a execução dos comandos presentes no ficheiro terminar, o interpretador acabará a sua execução.

4 Compilação

A compilação da linguagem apresentada deve ser feita pela emissão de código CIL e utilizando as ferramentas da framework Mono para produzir um ficheiro executável na máquina virtual.

A compilação das funções de *input/output* pode e deve ser feita recorrendo às bibliotecas da plataforma de suporte (concretamente à classe `System.Console`).

Os valores booleanos codificam-se em valores inteiros e as closures devem ser codificadas em novos métodos CIL. A sua implementação deve ser feita recorrendo a uma pilha de execução em memória (heap) onde valores são associados a identificadores declarados e onde argumentos são associados a parâmetros. Pode ser utilizado código C#, compilado para a Mono, para desenhar um pequeno suporte de execução para os programas compilados.

4.1 Opções do compilador

O compilador deve ter opções de linha de comando que permitam a definição do nome do ficheiro de saída e nome do ficheiro a compilar. Estas opções devem ser claramente indicadas no relatório. Deve ser apresentado um script de compilação para programas na linguagem base que inclua a análise sintática, a geração de código e a passagem de CIL para código máquina. Este script deve tomar como entrada o ficheiro a compilar e qual o ficheiro de saída. Deve ainda possuir maneira automática de produzir o suporte de run-time a partir de ficheiros fonte bem identificados.

5 Sistema de tipos

O sistema de tipos deve verificar se um comando ou expressão está bem tipificado/a tendo como base os tipos primitivos para valores inteiros, booleanos e um tipo primitivo `string` e nos tipos compostos para funções, procedimentos, registos e referências.

A sintaxe da linguagem deve ser modificada para acomodar anotações de tipo nas expressões e comandos.

```
C ::= C;C
    | decl id1:T1=E1 ... idn:Tn=En in C end
    | ...

E ::=
    | fun x1:T1,x2:T2,...,xn:T3 => E end
    | proc x1:T1,x2:T2,...,xn:T3 => C end
    | decl id1:T1=E1 ... idn:Tn=En in E end
    | ...
```

Deve ainda construir uma sintaxe para expressões de tipo:

```
T ::= int | string | bool | {id1:T1, Id2:T2, ..., idn:Tn} | Ref(T)
    | Fun(T1,T2,...,Tn)T | Proc(T1,T2,...,Tn)
```

6 Estrutura do relatório

A estrutura esperada do relatório será definida em versão posterior deste documento.

7 Entregáveis

A lista completa de entregáveis será definida em versão posterior deste documento. No entanto como lista provisória ficam os seguintes itens:

- Relatório
- Ficheiros fonte
- Executáveis (jar+scripts de execução)

8 Datas importantes

O trabalho deve ser entregue até ao **dia 31 de Dezembro**.

As discussões dos trabalhos decorrerão na semana de 2 a 6 de janeiro, em horário a ser divulgado oportunamente. A duração prevista é de 20 minutos.

A Exemplos de teste

A.1 Interpretador de cálculo lambda com inteiros

```
define
  Env      = fun => [] end
  Assoc    = fun e,i,v => [id=i, val=v, next=e] end
  Find     = fun e,i => if e.id = i then e.val else Find(e.next, i) end end

  vartag   = "_VAR"
  apptag   = "_APP"
  lamtag   = "_LAM"
  numtag   = "_NUM"

  ASTVar   = fun s => [type=vartag, name = s] end
  IsVar    = fun a => a.type = vartag end

  ASTApp   = fun e1, e2 => [type=apptag, fn = e1, arg = e2 ] end
  IsAppl   = fun a => a.type = apptag end

  ASTLam   = fun s, e => [type=lamtag, par = s, body = e] end
  IsLambda = fun a => a.type = lamtag end

  ASTNum   = fun n => [type=numtag, val=n] end
  IsNum    = fun a => a.type = numtag end

  PrettyP = decl
    Pretty = proc e =>
      if IsNum(e) then
        print(e.val)
      else if IsVar(e) then
        print(e.name)
      else if IsLambda(e) then
        print("(lambda ");
        print(e.par);
        print("."); Pretty(e.body); print(")")
      else if IsAppl(e) then
        print("(");
        Pretty(e.fn); print(" "); Pretty(e.arg);
        print(")")
      end end end end
    end
    in proc e => Pretty(e); println() end end

  MkClosure = fun i,b,e => [par=i, body=b, env=e] end

  Error = fun => do print("error!"); println() return [] end end

  Evalenv = fun expr, env =>
    if IsNum(expr) then
      expr else
    if IsVar(expr) then
      Find(env,expr.name)
    else if IsAppl(expr) then
      decl
        vf = Evalenv(expr.fn, env)
        va = Evalenv(expr.arg, env)
      in decl
        newenv = Assoc(vf.env, vf.par, va)
      in
        Evalenv(vf.body,newenv)
    end end
```

```

                else if IsLambda(expr) then
                    MkClosure(expr.par,expr.body,env)
                else Error() end
            end
        end
    end
end
end
end
Eval = fun e => Evalenv(e, Env()) end
;;

define Id = ASTLam("x",ASTVar("x"));
define P1 = ASTApp(Id, ASTApp(Id,ASTNum(1)));
define P2 = ASTApp(ASTApp(Id,Id),ASTNum(2));

PrettyP(Eval(P1));
PrettyP(Eval(P2));

define True = ASTLam("x",ASTLam("y",ASTVar("x")));
define False = ASTLam("x",ASTLam("y",ASTVar("y")));
define If = ASTLam("x",ASTLam("y",ASTLam("z", ASTApp(ASTApp(ASTVar("x"), ASTVar("y")) , ASTVar("z")))));

define P3 = ASTApp( ASTApp(ASTApp(If,False) , ASTNum(1)), ASTNum(2));

PrettyP(Eval( P3 ));

```

A.2 Árvores binárias

```
define BinTree =
  decl
    Node =
      fun l,r,v =>
        [
          getV = v,
          getL = var(l),
          getR = var(r)
        ]
      end
    root = var([])
    search = fun term, found =>
      fun r, v =>
        if (!r) = [] then
          term(r,v)
        else
          if (!r).getV = v then
            found(r,v) else
            if (!r).getV < v then
              (search(term, found))(!r).getR, v)
            else
              (search(term, found))(!r).getL, v)
            end
          end
        end
      end
    end
  in
    decl
      insertaux = search((fun r,v =>
        do r:= Node([], [],v)
        return v end end),
        (fun r,v => v end) )
      findaux = search((fun r,v=> false end),(fun r,v=> true end))
    in fun =>
      [
        insert = fun v => insertaux(root, v) end,
        find = fun v => findaux(root,v) end
      ]
    end
  end
end
;;

define t = BinTree();;

print(t.insert(1));;

print(t.insert(0));;

print(t.insert(2));;

print(t.find(2));;

print(t.find(3));;
```