Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2010-2011

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

"If you don't understand interpreters, you can still write programs; you can even be a competent programmer. But you can't be a master."

(Hal Abelson, prefácio de *Essentials of Programming Languages* de Friedman et al.)

Objectivos da Disciplina: Saber

- Quais são as **técnicas** usadas na implementação de LPs ?
- Quais são os **conceitos básicos** de construção das LPs ?
- Como se descrevem, analisam e justificam as **características** das várias linguagens de programação com base nos conceitos básicos apresentados ?
- Como se desenham **interpretadores e compiladores** para linguagens de programação ?
- Como se consegue **prever o comportamento dos programas** escritos numa linguagem de programação de forma precisa ?
- Como se expressam **propriedades sobre programas** como segurança de tipos ? O que significa segurança de tipos ?
- Como se descrevem e implementam **algoritmos de verificação** de linguagens de programação ?
- Como funcionam os ambientes de **suporte à execução** modernos (Java, .NET) ?

Objectivos da Disciplina: Fazer

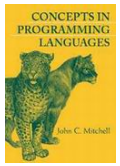
- Como se constroem **analisadores sintácticos** usando geradores ?
- Como se **representam programas** como dados para programas ?
- Como se **exprime a semântica** de uma linguagem ?
- Como se **constrói um interpretador** para linguagens funcionais ?
- E para uma linguagem orientada por **objectos** ?
- E se a máquina destino for uma **plataforma industrial** (.NET) ?
- Como se desenham **algoritmos** simples de **análise de programas** (sistemas de tipos) ?

"Mapa da Estrada"

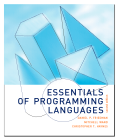
Em ICLP estudamos os conceitos fundamentais das linguagens de programação, abordando os seus aspectos sintácticos e semânticos tanto do ponto de vista dos fundamentos teóricos, como do ponto de vista da sua implementação:

- Expressões e valores
- Ligação e mecanismos de nomeação
- Estado (memória)
- Abstracção funcional e procedimental
- Tipos e sistemas de tipos
- Abstracção de dados
- Objectos, Classes e Módulos
- Técnicas de Compilação

Bibliografia



"*Concepts in Programming Languages*",
John C. Mitchell,
Cambridge University Press.
ISBN 0 521 78098 5



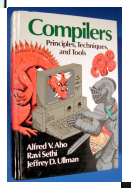
"Essentials of Programming Languages",
Daniel Friedman, Mitchell Wand, Christopher Haynes,
MIT Press.

"The Study of Programming Languages",
Ryan Stansifer,
Prentice Hall International Edition.

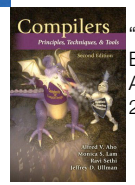
"Modern Compiler Implementation in Java"
Andrew W. Appel
Cambridge University Press

Outros Livros: "Clássicos"

"Principles of Compiler Design"
Alfred V. Aho, Jeffrey D. Ullman, 1977



"Compilers: Principles, Techniques, and Tools"
Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, 1986



"Compilers: Principles, Techniques, and Tools, Second Edition"
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, 2006

Regras de Avaliação

- Obtenção de frequência (P):
  - Trabalho Prático: Um programa interpretador e compilador para uma linguagem de programação dada. Ex: uma linguagem orientada por objectos. O trabalho prático é feito com base nos trabalhos das aulas práticas.
  - Grupos de 1 ou 2 elementos
  - Entrega em fases:
    - ex: linguagem funcional, linguagem orientada por objectos
  - Nota: 0 a 20 (por escalões, <=14, <=20)
- Exame final (E)
  - Com consulta
  - Nota mínima: 9,5
- Nota final: 70% Exame, 30% Trabalho Prático
- Datas importantes:
  - Apresentação do Enunciado: 05/11
  - Entrega primeira fase: 26/11
  - Entrega do trabalho: 10/12
  - Apresentações dos trabalhos: 15/12 a 17/12

Trabalhos Práticos

Trabalhos de Laboratório de Interpretação e Compilação de Linguagens de Programação

20 de Novembro de 2008

Os trabalhos práticos da disciplina de Interpretação e Compilação de Linguagens, incluindo o trabalho final, têm por objectivo a implementação de um compilador e um interpretador para uma linguagem de programação. Os trabalhos das aulas práticas são dirigidos de modo a que o trabalho final seja apenas uma extensão do trabalho realizado nas aulas práticas. As aulas práticas têm como material de apoio o conteúdo dos ficheiros disponibilizados na página da disciplina.

Trabalho Prático nº1 - Construção de um Analisador sintáctico

Neste trabalho pretende-se construir uma calculadora simples, ou seja, um interpretador para a linguagem das expressões aritméticas. A primeira peça deste programa é um analisador sintáctico para a linguagem. Dada a sintaxe das expressões aritméticas definida por uma gramática construa o analisador sintático recorrendo a um gerador de analisadores sintáticos para Java.

Para implementar o reconhecedor sintático utiliza-se um gerador de analisadores sintácticos (LL(k)) para a linguagem Java, chamado JavaCC (Java Compiler Compiler). Pode encontrar um tutorial de JavaCC em apêndice a este documento. Considere a gramática com o não terminal E e os terminais "+", "-", "*", "/" e num representando literais inteiros:

```
E ::= E "+" E | E "-" E | E "*" E | E "/" E | "("E")" | num
```

Note que esta gramática é ambígua, ou seja, para uma mesma expressão é possível desenhar mais do que uma árvore de derivação sintáctica. É portanto necessário desambiguar a gramática tornando explícitas as prioridades dos operadores. Pode-se proceder a modificações na gramática para conseguir este fim. Considere a gramática equivalente não ambígua com os não terminais E, F, e T:

```
E ::= T ( "+" | "-" ) E | T
T ::= F ( "*" | "/" ) T | F
F ::= num | "(" E ")"
```

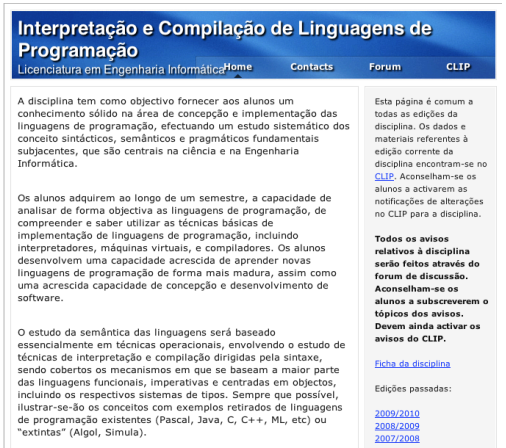

Trabalho Prático (exemplo)

- Um programa interpretador, compilador e sistema de tipos para uma linguagem de programação orientada por objectos :

```
decl
  Counter = class
    var count = 0
    proc inc => count := !count + 1
    fun get => !count
  end
in
decl
  o = new Counter
in
decl
  i = var(0)
in
  while !i < 10 do o.inc(); i := o.get() end;
  println(o.get())
end
end
end
```


Página da Disciplina

- O público em http://ctp.di.fct.unl.pt/lei/icl
- o resto no CLIP



Unidade 1: Introdução

Esta unidade fala sobre conceitos gerais associados ao desenho de linguagens de programação e ambientes de programação.

- Expressividade das linguagens de programação
- Computabilidade e completude de Turing
- Sintaxe e semântica das linguagens de programação
- Sintaxe abstracta e sintaxe concreta
- Interpretação e compilação de programas
- Ambientes de execução - Máquinas Virtuais, Linguagens intermédias, portabilidade

Expressividade das Linguagens de Programação

1. O que é uma linguagem de programação?

2. Quais são as **possibilidades** e as **limitações** teóricas das linguagens de programação?

Expressividade das Linguagens de Programação

1. O que é uma linguagem de programação?
   - É uma linguagem para descrever ***processos computacionais***.
   - E o que são "processos computacionais"?

Expressividade das Linguagens de Programação

1. O que é uma linguagem de programação?
   - É uma linguagem para descrever ***processos computacionais***.
   - E o que são "processos computacionais"?

Podemos considerar duas grandes categorias:

Processos "transformadores"
transformam uma ***entrada*** num ***saída (input /output)***, e terminam ***(morrem)***

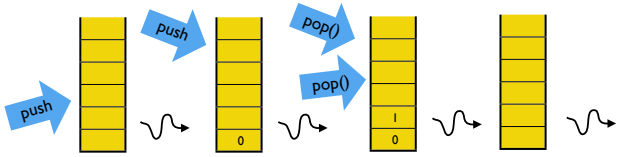
São os processos mais elementares que se podem imaginar.

Expressividade das Linguagens de Programação

1. O que é uma linguagem de programação?
   - É uma linguagem para descrever ***processos computacionais***.
   - E o que são "processos computacionais"?

Podemos considerar duas grandes categorias:

Processos "transformadores"
Processos "interactivos"



Interagem com o ambiente, mudando de estado em consequência. Podem nunca terminar (ex: webserver).

Expressividade das Linguagens de Programação

1. O que é uma linguagem de programação?
   - É uma linguagem para descrever ***processos computacionais***.
   - E o que são "processos computacionais"?

Vamos apenas considerar os processos transformadores:

Processos "transformadores"
transformam uma ***entrada*** num ***saída (input /output)***, e terminam ***(morrem)***

São os processos mais elementares que se podem imaginar. Aparentemente estes processos são apenas funções, no sentido matemático usual do termo.

Qual a diferença entre uma função e um "processo computacional"?

- Uma função atribui um valor no conjunto contradomínio para cada valor do conjunto domínio.
- O resultado de uma computação pode ser indefinido:
  - Operação não definida (ex: divisão por zero)
    - A expressão "3/0" não tem valor (mesmo!)
    - A implementação pode decidir abortar a execução
  - Não-Terminação
    - f(x) ≜ if x=0 then 1 else f(x-2)
    - f é uma função ***parcial*** : não está definida em todos os argumentos
- Estas duas situações são "Matematicamente" equivalentes (porquê?), mas **Operacionalmente** diferentes

Qual a diferença entre uma função e um "processo computacional"?

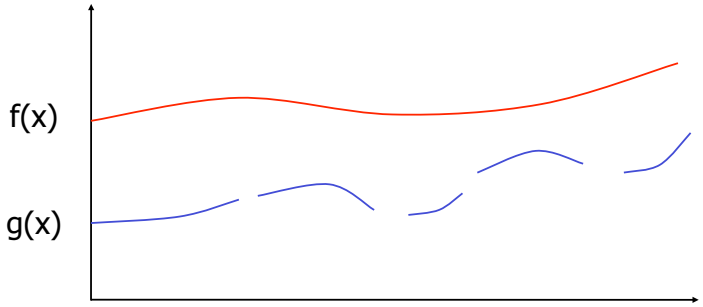
- Uma função atribui um valor no conjunto contradomínio para cada valor do conjunto domínio.
- O resultado de uma computação pode ser indefinido:
  - Operação não definida (ex: divisão por zero)
    - A expressão "3/0" não tem valor (mesmo!)
    - A implementação pode decidir abortar a execução
  - Não-Terminação
    - f(x) ≜ if x=0 then 1 else f(x-2)
    - f é uma função ***parcial*** : não está definida em todos os argumentos
- A terminação não pode ser detectada em tempo de compilação (uma instância do chamado "halting problem")

Funções Parciais vs. Totais

Função total: f(x) é definido para todo o valor x.

Função parcial: g(x) é indefinido para alguns valores x.

(imagens retiradas de [J.Mitchell,2002])

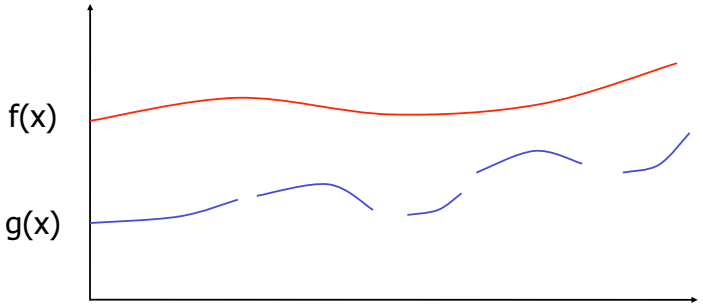


Funções vistas como "gráficos"

Gráfico de f = { <x,y> | y = f(x) }

Gráfico de g = { <x,y> | y = g(x) }

Uma função é um conjunto de pares ordenados (gráfico da função)



Funções Parciais vs Totais

- Uma função total f:A→B é um subconjunto f ⊆ A×B tal que:
  - Para todo x ∈ A, existe y ∈ B tal que ⟨x,y⟩ ∈ f (totalidade)
  - Se ⟨x,y⟩ ∈ f e ⟨x,z⟩ ∈ f então y=z (unicidade)
- Uma função parcial f:A→B é um subconjunto f ⊆ A×B tal que:
  - Se ⟨x,y⟩ ∈ f e ⟨x,z⟩ ∈ f então y=z (unicidade)
- Os programas definem funções parciais por duas razões
  - Existem operações parciais:
    - divisão, desempilhar uma pilha, etc...
  - Não-terminação; por exemplo
    - f(1) com f(x) = if x=0 then 1 else f(x-2)

Computabilidade

- Definição:

  Uma função f diz-se computável
  se existe um programa P que calcula f.

  Ou seja,

  Para todo o ***input*** x, se ***f(x)*** está definido, então
  a execução de ***P(x)*** termina e fornece o ***output f(x)***
- Terminologia:
  "Funções recursivas parciais" = funções parciais computáveis

Será que todas as funções parciais são computáveis?

- Por outras palavras:
  será que é possível programar toda e qualquer função parcial?
- "ser possível programar" depende da linguagem!
- Será possível definir uma linguagem de programação em que seja possível exprimir qualquer função parcial?

"Halting Problem"

"Dado um programa P que recebe uma string x, determinar se P termina com a entrada x."

- Considere-se a função Halt com dois parâmetros P e x que decide se um programa termina com um determinado input.

Dado um programa P e um ***input*** x, seja

$$\mathit{Halt}(P, x) \triangleq \begin{cases} \textbf{true} & \text{se P}(x) \text{ termina} \\ \textbf{false} & \text{caso contrário} \end{cases}$$

A função ***Halt*** aceita um programa P como *input*. Não se baralhe com isto, existem muitas funções que aceitam programas como input (por exemplo, os compiladores, os browsers da web, etc).

"Halting Problem"

"Dado um programa P que recebe uma string x, determinar se P termina com a entrada x."

- Considere-se a função Halt com dois parâmetros P e x que decide se um programa termina com um determinado input.

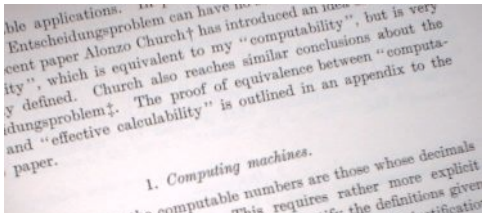
Dado u

Teorema (Turing, 1931):
Não existe um programa para a função *Halt*

(leia-se: não é possível implementar a função *Halt*)

A função ***Halt*** aceita um programa P como *input*. Não se baralhe com isto, existem muitas funções que aceitam programas como input (por exemplo, os compiladores, os browsers da web, etc).

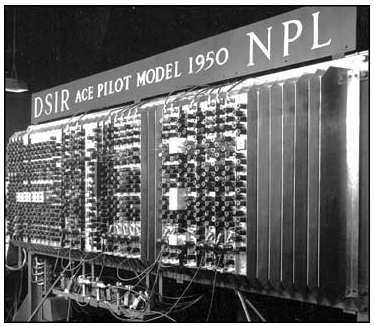
Alan Turing (1912-1954)



Entscheidungsproblem can have
cent paper Alonzo Church† has introduced an
ity", which is equivalent to my "computability", but is very
y defined. Church also reaches similar conclusions about the
dungsproblem‡. The proof of equivalence between "computa-
and "effective calculability" is outlined in an appendix to the
paper.

1. *Computing machines.*

computable numbers are those whose decimals
This requires rather more explicit
the definitions given
ification



EXECUTIVE COMMITTEE

'ACE' MACHINE PROJECT

Memorandum by Mr. J. R. Womersley, Superintendent, Mathematics Division

The research programme of the Mathematics Division contains an item "To explore the application of switching methods (mechanical, electrical and electronic) to computations of all kinds." The first step in this exploration was the visit of the newly-appointed Superintendent to the U.S.A. in February, 1945, to gain acquaintence of recent American developments in this field, and later Dr. A. M. Turing was appointed to the staff of the Division, and began to consider the possibilities of electronic methods, in the light of recent progress in the U.S.A. Dr. Turing has now completed a long report, which makes definite proposals for the construction of a machine, capable of solving a wide variety of problems at speeds hitherto unattainable.

Indecidibilidade da função Halt

1. Assuma que o programa P implementa (uma variante de) ***Halt***

   Dado um input Q, seja

$$P(Q) \triangleq \begin{cases} \textbf{true} & \text{se Q(Q) termina} \\ \textbf{false} & \text{caso contrário} \end{cases}$$

2. Defina um programa D assim

$$D(Q) \triangleq \begin{cases} \textbf{if } P(Q) \textbf{ then while(true)\{\}} \\ \qquad \textbf{else stop} \end{cases}$$

Indecidibilidade da função Halt

1. Assuma que o programa P implementa (uma variante de) ***Halt***

   Dado um input Q, seja

$$P(Q) \triangleq \begin{cases} \textbf{true} & \text{se Q(Q) termina} \\ \textbf{false} & \text{caso contrário} \end{cases}$$

2. Defina um programa D assim

$$D(Q) \triangleq \begin{cases} \textbf{if } P(Q) \textbf{ then while(true)\{\}} \\ \qquad\qquad \textbf{else stop} \end{cases}$$

Que pode a chamada D(D) fazer ?

Indecidibilidade da função Halt

1. Assuma que o programa P implementa (uma variante de) ***Halt***

   Dado um input Q, seja

$$P(Q) \triangleq \begin{cases} \textbf{true} & \text{se Q(Q) termina} \\ \textbf{false} & \text{caso contrário} \end{cases}$$

2. Defina um programa D assim

$$D(Q) \triangleq \begin{cases} \textbf{if } P(Q) \textbf{ then while(true)\{\}} \\ \qquad\qquad \textbf{else stop} \end{cases}$$

Que pode a chamada D(D) fazer ?

Se D(D) termina, então D(D) não termina

Se D(D) não termina, então D(D) termina!

CONTRADIÇÃO!

Onde está o "erro" ?

Limitações Essenciais das Linguagens de Programação

- Algumas funções parciais são computáveis, outras não …
  - Halting problem (e muitos outros)
- Impacto na implementação de processadores de linguagens
  - É possível detectar e assinalar um erro no caso de uma operação estar indefinida (pilha vazia, overflow, divisão por zero)
  - Nem sempre é possível determinar se um programa vai terminar ou não, se vai eventualmente executar uma operação indefinida, qual a memória que um programa vai necessitar para correr, etc, etc, etc …
- No entanto, é possível aproximar algumas destas análises
  - (ex: usando sistemas de tipos).
  - Tentando caracterizar os casos em que não se consegue determinar…

TERMINATOR

proof tools for termination and liveness

http://research.microsoft.com/TERMINATOR/

Microsoft Research News & Highlights

Terminator Tackles an Impossible Task

By Rob Knies

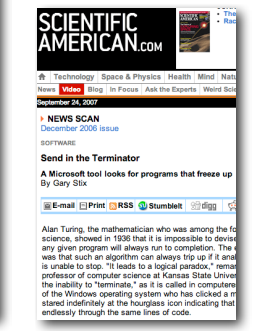
It is May 28, 1936. A paper entitled *On Computable Numbers With an Applic*
Entscheidungsproblem is received by the London Mathematical Society for p

In the paper, the latest salvo in a mathematical debate that extends back to
century, Cambridge academic Alan Turing proves that a general algorithm t
possible combinations of programs and initial inputs will run to completion a
to devise. Computer scientists refer to this as the "halting problem."

Fast forward 70 years. Microsoft researcher Byron Cook, buttressed by his
clutch of European computer scientists, prepares to deliver a series of lectu
automatic methods for proving program termination in most—and perhaps,
—circumstances.

What's going on here? Turing is popularly considered to be the father of mo
The Association for Computing Machinery named its most prestigious techni
Award. Is it possible that one of his enduring proofs is about to be upended?

Not quite. But someday, the impact of Turing's result may be considered str
Terminator project, Cook, a researcher in the Programming Principles and T
Microsoft Research's Cambridge lab, is simply doing what researchers do: p
difficult problems and making scientific advances. In this case, Cook is chall
a sense, you could say he is whittling away at the impossible.



SCIENTIFIC AMERICAN.COM

NEWS SCAN

December 2006 issue

SOFTWARE

Send in the Terminator

A Microsoft tool looks for programs that freeze up

By Gary Stix

Alan Turing, the mathematician who was among the fo
science, showed in 1936 that it is impossible to devis
any given program will always run to completion. The
was that such an algorithm can always trip up if it ena
is unable to stop. "It leads to a logical paradox," rema
professor of computer science at Kansas State Univer
the inability to "terminate," as it is called in computer
of the Windows operating system who has clicked a m
stared indefinitely at the hourglass icon indicating that
endlessly through the same lines of code.

Resumo: Ideias a reter!

- Processo computacional = Função parcial calculável por
  - Máquina de Turing
  - Cálculo Lambda [Church]
  - Sistemas de equações recursivas [Godel]
  - … todos estes modelos são equivalentes!
- Terminologia:

Uma linguagem de programação diz-se

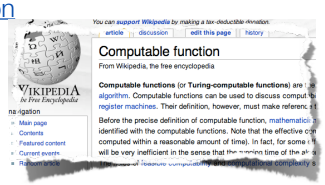
Turing-completa

se permitir programar todas as funções efectivamente computáveis.

Leituras: Ideias a procurar!!

Capítulo 2, *Concepts in programming Languages*. John C. Mitchell

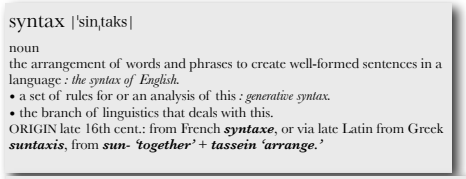
http://en.wikipedia.org/wiki/Computable_function

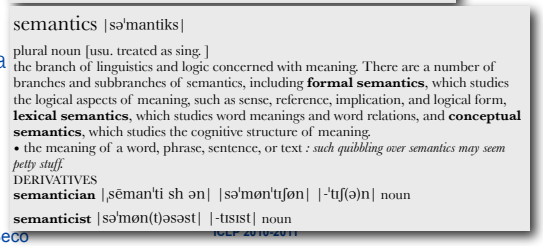


http://research.microsoft.com/TERMINATOR/

Linguagens de Programação

- As linguagens de programação descrevem/especificam processos computacionais (computações).
- As linguagens são compostas por duas partes que devem ser descritas de uma forma precisa e não ambígua:
  - Sintaxe
  - Semântica





Linguagens de Programação

Exemplo de ambiguidade na sintaxe:
Qual o valor da expressão `f(10)` ?

```
int f(int x) {
  if ( x > 0 )
    if ( x < 10 ) return x;
  else return 10;
  return 0;
}
```

```
f(10) = ??
```


Linguagens de Programação

Exemplo de ambiguidade na semântica:
Qual o valor da expressão `f(2)+g(3)` ?

```
#include "stdio.h"

int a = 0;

int f(int x) { a = a + 1; return x; }

int g(int y) { return y+a; }

int sum(int x, int y) { return x+y; }

int main() { printf("%d\n", sum(f(2),g(3))); }
```

```
f(2) + g(3) = ??
```


Linguagens de Programação

Exemplo de ambiguidade na semântica:
Qual o valor da expressão `f(2)+g(3)` ?

```
public class    A {
   static int a = 0;

   static int f(int x) {
          a = a + 1;
          return x;
   }
   static int g(int y) { return y + a; }

   static int sum(int x, int y) { return x + y; }

   public static void main(String[] args) {
          System.out.println(sum(f(2), g(3)));

   }
}
```

```
f(2) + g(3) = ??
```


Linguagens de Programação (Sintaxe)

A sintaxe caracteriza a forma como se escrevem os programas da linguagem, sem atender ao seu significado. A sintaxe é normalmente descrita por um conjunto de palavras ou "tokens", formando um léxico, e a estrutura em que se pode organizar os tokens para formar frases bem formadas.

```
Inteiro:  ("0" | ["1"-"9"]["0"-"9"]*)
Real: (["0"-"9"]) "." (["0"-"9"])* ("E" ...)?
Identificador: [a-z,A-Z,_][a-z,A-Z,_,0-9]*
palavras reservadas: int, float, void, while, class, ...
```

The `if` statement is in the form:

```
if (<expression>)
    <statement1>
else
    <statement2>
```

```
switch (<expression>)
{
    case <label1> :
        <statements 1>
    case <label2> :
        <statements 2>
        break;
    default :
        <statements 3>
}
```

Linguagens de Programação (Sintaxe)

A sintaxe caracteriza a forma como se escrevem os programas da linguagem, sem atender ao seu significado. A sintaxe é normalmente descrita por um conjunto de palavras ou "tokens", formando um léxico, e a estrutura em que se pode organizar os tokens para formar frases bem formadas.

Exemplo de observação sobre "sintaxe":

"Enquanto na linguagem C os blocos são delimitados por chavetas { e }, na linguagem Pascal usam-se os delimitadores `begin end`"

A sintaxe concreta de uma linguagem de programação pode ser descrita precisa e formalmente usando expressões regulares para os tokens e gramáticas para as frases (Teoria da Computação).

A partir gramática da linguagem constroem-se programas que reconhecem e processam os programas bem formados, os Parsers.

Linguagens de Programação (Sintaxe)

Um exemplo para começar

Derivação esquerda numa gramática LL(1)

Uma gramática não-LL(1)

$S \rightarrow E;$
$E \rightarrow T \mid E + T$
$T \rightarrow a \mid (E)$

Eliminando a recursividade à esquerda

$S \rightarrow E;$
$E \rightarrow TX$
$X \rightarrow \lambda \mid +TX$
$T \rightarrow a \mid (E)$

É uma gramática LL(1)!

Derivação esquerda de $a + a$;

Derivação	Palavra
S	$a + a;$
$\Rightarrow E;$	$a + a;$
$\Rightarrow TX;$	$a + a;$
$\Rightarrow aX;$	$a + a;$
$\Rightarrow a + TX;$	$a + a;$
$\Rightarrow a + aX;$	$a + a;$
$\Rightarrow a + a;$	$a + a;$

As produções são determinadas pelo próximo símbolo da palavra ainda não produzido: símbolo director da produção.

Linguagens de Programação (Sintaxe)

yacc

```
%token  NAME
%token  NUMBER
%token  EQ
%token PLUS MINUS TIMES DIV
%left MINUS PLUS
%left TIMES DIV
%nonassoc UMINUS

%%

statement_list
        : statement
        | statement statement_list

statement
        : NAME EQ expression ';' {vbltable[$1] = $3; }

expression
        : expression PLUS expression {$$ = $1 + $3;}
        | expression MINUS expression {$$ = $1 - $3;}
        | expression TIMES expression {$$ = $1 * $3;}
        | expression DIV   expression {$$ = $1 / $3;}
        | MINUS expression %prec UMINUS {$$ = - $2;}
        | '(' expression ')' { $$ = $2; }
        | NUMBER
        | NAME   { $$ = vbltable[$1]; }
```


Linguagens de Programação (Sintaxe)

javacc

```
void Start() :
{ }
{
    exp() <EOL>
}

void exp() :
{ }
{
    term() [ <PLUS> exp() ]
}

void term() :
{ }
{
    factor() [ <MULTIPLY> term() ]
}

void factor() :
{  }
{
    <CONSTANT>
|   <LPAR> exp() <RPAR>
}
```

Linguagens de Programação (Semântica)

A semântica descreve o significado das frases sintacticamente válidas de uma linguagem. A semântica das linguagens deve ser definida de maneira precisa.

10.4 Array Access

A component of an array is accessed by an array access expression (§15.13) that consists of an expression whose value is an array reference followed by an indexing expression enclosed by `[` and `]`, as in `A[i]`. All arrays are `0`-origin. An array with length *n* can be indexed by the integers `0` to *n*-1.

Arrays must be indexed by `int` values; `short`, `byte`, or `char` values may also be used as index values because they are subjected to unary numeric promotion (§) and become `int` values. An attempt to access an array component with a `long` index value results in a compile-time error.

All array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an `ArrayIndexOutOfBoundsException` to be thrown.

in *The Java Language Specification*

Linguagens de Programação (Semântica)

A semântica descreve o significado das frases sintacticamente válidas de uma linguagem. A semântica das linguagens deve ser definida de maneira precisa.

Exemplos de observações sobre "semântica":

"Em C, um vector é modelado por um apontador, mas em Pascal um vector é um valor primitivo"

"A linguagem ML (OCaml) é uma linguagem imperativa, mas centrada no uso de funções"

"Na linguagem Java os argumentos são sempre passados por valor, mas em Pascal também podem ser passados por referência"

Linguagens de Programação (Semântica)

A semântica descreve o significado das frases sintacticamente válidas de uma linguagem. A semântica das linguagens deve ser definida de maneira precisa.

A semântica de uma linguagem pode ser definida precisamente por uma função computável I que atribui um significado a cada programa (ou fragmento de programa)

$$\mathrm{I} : \mathrm{PROG} \rightarrow \mathrm{DENOT}$$

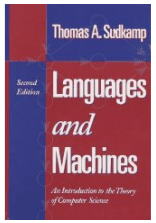
PROG = conjunto dos programas
DENOT = conjunto dos significados possíveis (denotações)

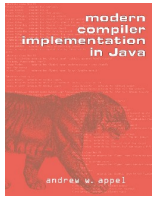
A função semântica I pode ser vista como um **algoritmo** que "sabe como interpretar" todos os programas (sintacticamente correctos) de uma linguagem, determinando o seu **valor** ou **efeito**

Resumo: Ideias a reter!

- Uma linguagem de programação é definida por sintaxe e semântica
- A sintaxe de uma linguagem é definida por uma gramática e é reconhecida e manipulada por uma ferramenta, um analisador sintáctico (*Parser*).
- A semântica de uma linguagem é definida por um algoritmo interpretador que dado um programa dá uma denotação (valor, efeito, programa noutra linguagem, etc)

Leituras: Ideias a procurar!!





Aulas de Teoria da Computação sobre gramáticas - LL(1)

Capítulos 1, 2 e 3: "Modern Compiler Implementation in Java"

"Languages and Machines", Thomas A. Sudkamp, Part II.

http://en.wikipedia.org/wiki/Syntax_of_programming_language

http://en.wikipedia.org/wiki/Backus–Naur_Form

http://en.wikipedia.org/wiki/Semantics#Computer_science

Interpretação e compilação (software translators)

Qual a diferença entre um compilador e um interpretador?

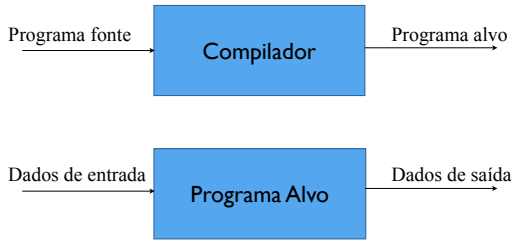
O que é uma linguagem intermédia?

O que é mais eficiente? O que é um JIT?

Qual é a arquitectura de um compilador?

Compiladores

"Um compilador é um programa que lê um programa numa linguagem (fonte) e o traduz para um programa equivalente noutra linguagem (alvo). Um papel importante do compilador é detectar erros no programa fonte. Se a linguagem alvo for uma linguagem máquina (executável) então o programa pode ser chamado para processar dados de entrada e produzir dados de saída." (in Aho et al)



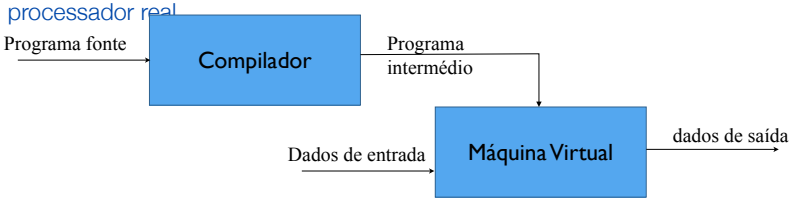
Interpretadores

Um interpretador é um programa que lê um programa numa linguagem (fonte) e produz um valor ou um efeito no seu próprio estado. Um interpretador é normalmente mais lento na produção dos dados de saída.

Interpretadores

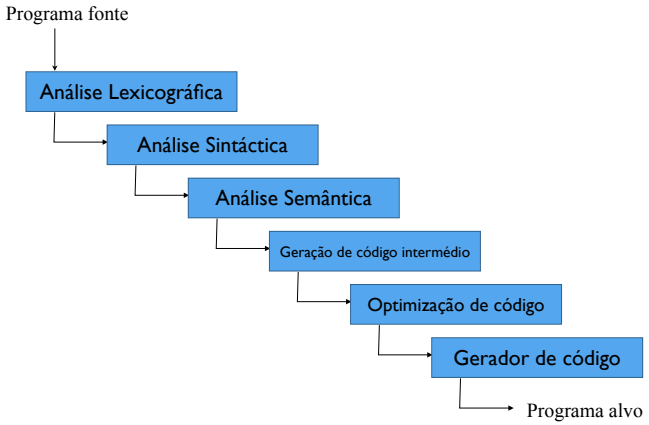
Um interpretador é um programa que lê um programa numa linguagem (fonte) e produz um valor ou um efeito no seu próprio estado. Um interpretador é normalmente mais lento na produção dos dados de saída.

No caso da linguagem Java é utilizada uma linguagem intermédia que é interpretada por uma máquina virtual (a JVM). Para além disso, o código intermédio é compilado em tempo de execução (JIT) para código máquina e corrido directamente no processador real.



Arquitectura de um compilador

- Um compilador divide-se tradicionalmente nas seguintes fases:

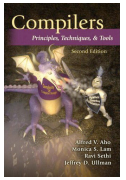


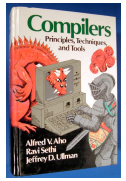
Leituras: Ideias a procurar!!

Secção 4.1: “Concepts in Programming Languages”

Capítulo 1: “Compilers: Principles, Techniques, and Tools”

http://en.wikipedia.org/wiki/Compiler

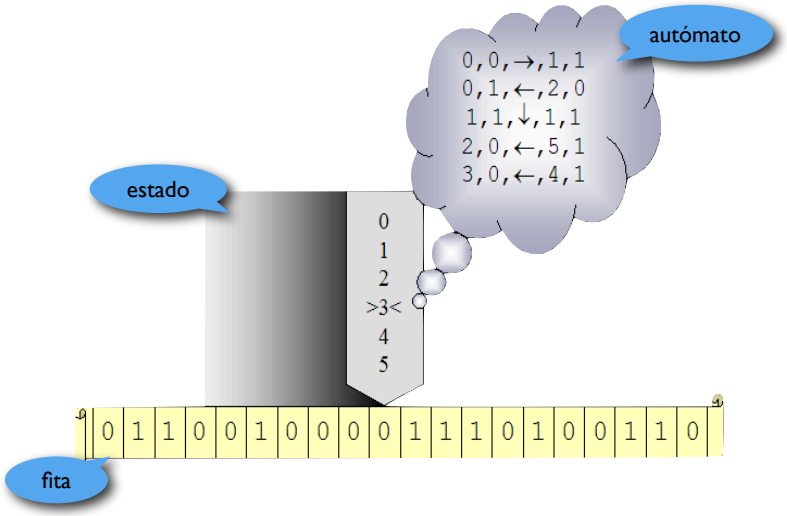




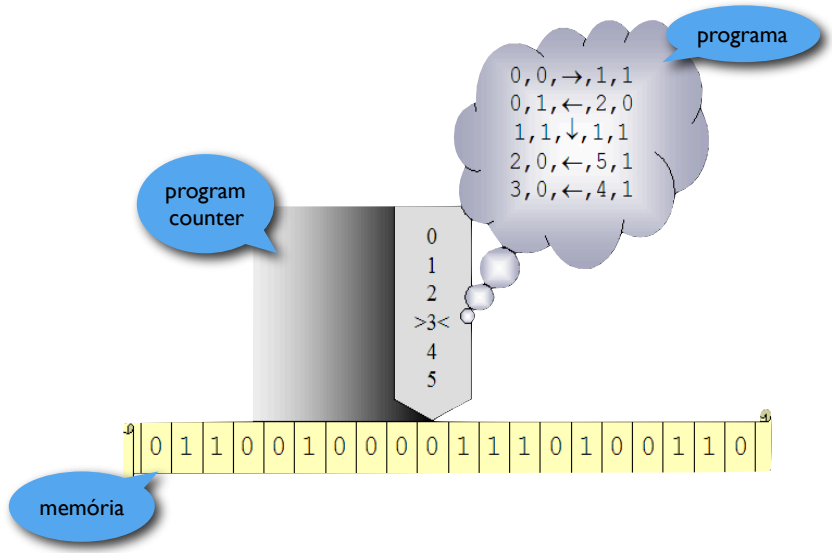
Ambientes de execução: Máquinas Virtuais (software processors)

- Máquina de Turing (Turing, 1931)
  - A primeira …
- SECD (Landin, 1962)
  - Stack, Environment, Code, Dump
- P-code machine (Wirth 1972)
  - Máquina de Pilha, Pascal p-code compiler
- JVM (Sun, 1995)
  - Multi-threaded, tipificada, dedicada à linguagem Java
- CLR (Microsoft, 2000)
  - Multi-threaded, tipificada, Multi linguagem

Máquina de Turing (Turing, 1931)



Máquina de Turing (Turing, 1931)



SECD - machine (Landin, 1962)

- A primeira desenhada para implementar o cálculo lambda
- Tem quatro registos a apontar para listas ligadas
  - S : stack
  - E : Environment
  - C : Code
  - D : Dump
- Cada posição da memória pode ter um átomo (12) ou uma lista (um par com dois endereços (o do primeiro elemento e o da lista que se segue)
- A lista (1 2 3) pode ser representada por:

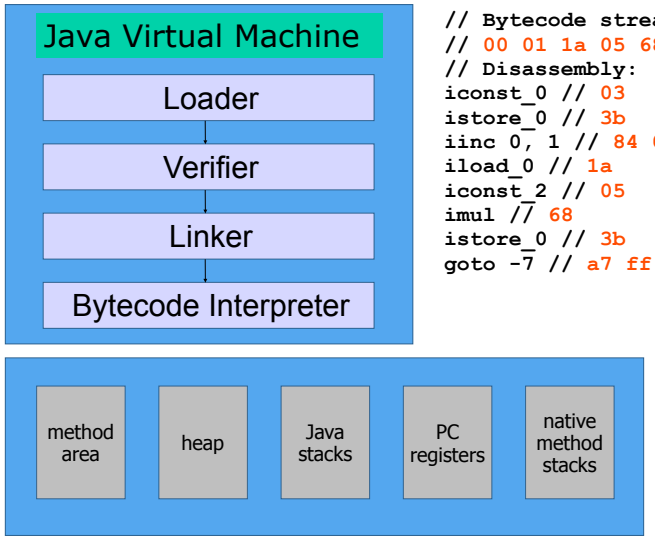
- As instruções: nil, ldc, ld, sel, join, ap, ret, dum, rap

P-code machine (Wirth 1972)

- É uma máquina de pilha o que quer dizer que as instruções da máquina vão buscar os operandos à pilha e colocam o seu resultado de volta na pilha.
- Máquina simples de implementar apenas com uma pilha partilhada entre informação de controlo e dados.

```
procedure interpret;
   const stacksize = 500;

   var
      p,b,t: integer; {program-, base-, topstack-registers}
      i: instruction; {instruction register}
      s: array [1..stacksize] of integer; {datastore}

   function base(l: integer): integer;
      var b1: integer;
   begin
      b1 := b; {find base l levels down}
      while l > 0 do
         begin b1 := s[b1]; l := l - 1
         end;
      base := b1
   end {base};

begin
   writeln(' start pl/0');
   t := 0; b := 1; p := 0;
   s[1] := 0; s[2] := 0; s[3] := 0;
   repeat
      i := code[p]; p := p + 1;
      with i do
      case f of
      lit: begin t := t + 1; s[t] := a end;
      opr: case a of {operator}
           0: begin {return}
                t := b - 1; p := s[t + 3]; b := s[t + 2];
              end;
           1: s[t] := -s[t];
           2: begin t := t - 1; s[t] := s[t] + s[t + 1] end;
```

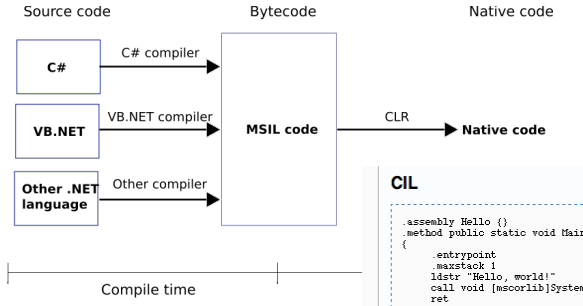

Java Virtual Machine (Sun, 1995)




```
// Bytecode stream: 03 3b 84
// 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
iconst_0 // 03
istore_0 // 3b
iinc 0, 1 // 84 00 01
iload_0 // 1a
iconst_2 // 05
imul // 68
istore_0 // 3b
goto -7 // a7 ff f9
```

Common Language Runtime (Microsoft, 2000)

- Máquina de pilha, base para a plataforma Microsoft .NET
- Desenhada para executar programas de várias linguagens.
- CIL - Common Intermediate Language



CIL

```
.assembly Hello {}
.method public static void Main() cil managed
{
      .entrypoint
      .maxstack 1
      ldstr "Hello, world!"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
}
```

Leituras: Ideias a procurar!!

http://en.wikipedia.org/wiki/Turing_machine

http://en.wikipedia.org/wiki/SECD_machine

http://en.wikipedia.org/wiki/P-code_machine

http://en.wikipedia.org/wiki/Common_Language_Runtime

http://en.wikipedia.org/wiki/JVM

Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2010-2011

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Unidade 2: Sintaxe Abstracta

Os interpretadores e compiladores são algoritmos que aceitam programas sintaticamente válidos e produzem denotações (valores, efeitos, ou outros programas). Os programas são os dados de entrada para esta classe de algoritmos.

Como podemos representar os programas para que sejam processados?

Como podemos definir o processamento de programas.

- Definição inductiva de tipos de dados
- Algoritmos recursivos sobre tipos de dados inductivos
- Representação da sintaxe abstracta como tipo inductivo
- Sintaxe concreta vs sintaxe abstracta (parsing)
- Um programa como dados de input para outro programa

Sintaxe e Semântica

- **Conceito chave**: definição da semântica de uma linguagem por indução na estrutura sintática da linguagem
- **Conceito chave**: definição da semântica de uma linguagem por meio de um algoritmo interpretador
- Como definir a sintaxe de uma linguagem de programação como um tipo de dados ?
- Como definir a semântica de uma linguagem de programação como um algoritmo interpretador ?

Tipos de Dados

- Um tipo de dados pode ser representado como um conjunto de valores decidível.
- Tal conjunto de valores pode ser infinito, mas cada valor é “finito”.
- Exemplos:
  - Booleanos (**true** / **false**)
  - Números inteiros (…, -2, -1, 0, 1, 2, …)
  - Listas
  - Árvores binárias
  - etc ...

Definição de tipos de dados

- Quando o conjunto de valores possíveis é finito, podemos definir os valores do tipo por enumeração :
  - Boolean ≜ { **true**, **false** }
  - BasicColor ≜ { **red**, **green**, **blue** }
- Como definir o conjunto dos valores de um tipo quando o número de valores possível é infinito ?
  - NaturalNumber ≜ ?
  - List ≜ ?
  - Tree ≜ ?

Definição indutiva de dados

- O mecanismo de definição "universalmente" usado em informática é a definição **indutiva**.

O tipo *NaturalNumber* é definido pelas seguintes regras :

1. ***0*** é um número natural (caso base)
2. se ***n*** é um número natural, então ***succ(n)*** é também um número natural
3. não existem mais números naturais, excepto os construídos de acordo com as regras 1 e 2 acima.

Definição indutiva de dados

- O mecanismo de definição "universalmente" usado em informática é a definição **indutiva**.

O tipo *List* (de elementos de tipo T) é definido por:

1. ***nil*** é uma lista (a lista vazia)
2. se ***x*** é um valor de tipo T e ***L*** é uma lista, então ***cons***(***x***, ***L***) é também uma lista
3. não existem mais listas, excepto as construídas de acordo com as regras 1 e 2 acima.

```
type list = Nil | Cons of int * list
```


Definição indutiva de dados

- Todas as definições indutivas de tipos de dados obedecem ao mesmo padrão:

O tipo *LabeledTree* (de valores de tipo T) é definido por:

1. ***empty*** é uma árvore (a árvore vazia)
2. se x é um valor de tipo T, e T1 e T2 são árvores, então ***node***(***T1***, ***x***, ***T2***) também é uma árvore, com raiz etiquetada por ***x***, cuja subárvore esquerda é ***T1*** e cuja subárvore direita é ***T2***.
3. Não existem mais … 1. e 2.

```
type tree = Empty | Node of tree * int * tree
```

Definição indutiva de dados

- Todas as definições indutivas de tipos de dados obedecem ao mesmo padrão:

O tipo *Stack* (de valores de tipo T)

1. ***empty*** é uma pilha (a pilha vazia)
2. se ***x*** é um valor de tipo T, e ***S*** é uma pilha, então ***push***(***x***,***S***) também é uma pilha, cujo topo contém o valor x, por trás do qual está a pilha S.
3. Não existem mais … 1. e 2.

```
type stack = Empty | Push of int * stack
```


Definição indutiva de dados

- Os ingredientes da definição indutiva de um tipo de dados são:

1. o **nome** do tipo T
2. um conjunto de **construtores**

- Um construtor é uma função que permite construir novos valores do tipo T a partir de valores já construídos do mesmo tipo ou de outros tipos.
- Cada construtor tem uma assinatura, que indica os tipos dos seus parâmetros.
- O tipo resultado de cada construtor do tipo T é o tipo T

Definição indutiva de dados

- Elementos da definição indutiva do tipo ***lista*** de valores de tipo **integer**

- o nome do tipo: ***ListInt***
- um conjunto de construtores: **nil**, **cons**

nil: () → ListInt

cons: Integer × ListInt → ListInt

Definição indutiva de dados

- Elementos da definição indutiva do tipo ***lista*** de valores de tipo **integer**

- o nome do tipo: ***ListInt***
- um conjunto de construtores: **nil**, **cons**

nil: () → ListInt

cons: Integer × ListInt → ListInt

```
/* ListInt.h */
typedef struct ListInt ListInt;
ListInt* nil();
ListInt* cons(int elem, ListInt *tail);
```

Definição indutiva de dados

- Elementos da definição indutiva do tipo ***lista*** de valores de tipo **integer**
- o nome do tipo: ***ListInt***
- um conjunto de construtores: **nil**, **cons**

nil: () → ListInt

cons: Integer × ListInt → ListInt

```
class ListInt {
    static ListInt nil();
    static ListInt cons(int elem, ListInt tail);
};
```


Definição indutiva de dados

- Elementos da definição indutiva do tipo ***árvore etiquetada*** de valores de tipo **integer**
- o nome do tipo: **TreeInt**
- um conjunto de construtores: **empty**, **node**

empty: () → TreeInt

node: TreeInt × Integer × TreeInt → TreeInt

Definição indutiva de dados

- Elementos da definição indutiva do tipo ***árvore etiquetada*** de valores de tipo **integer**
- o nome do tipo: **Treeint**
- um conjunto de construtores: **empty**, **node**

empty: () → TreeInt

node: TreeInt × Integer × TreeInt → TreeInt

```
interface TreeInt {
    TreeInt();
    TreeInt(TreeInt l, int elem, TreeInt r);
};
```


Definição indutiva de dados

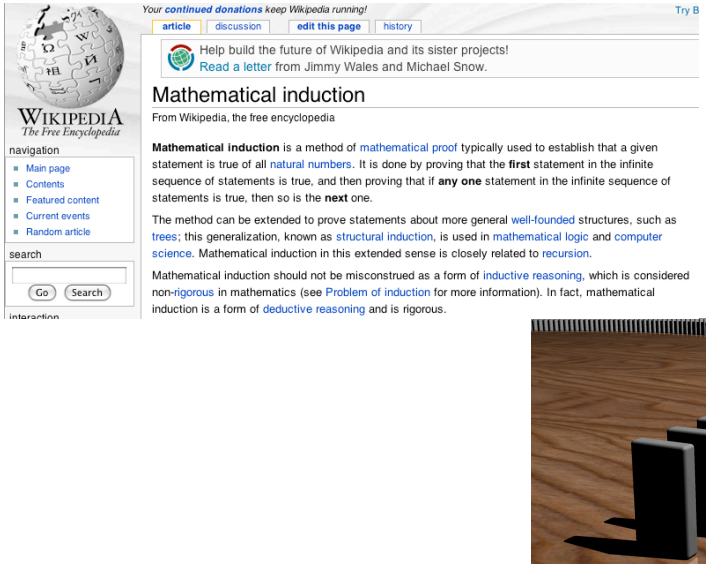
- Elementos da definição indutiva do tipo ***árvore etiquetada*** de valores de tipo **integer**
- o nome do tipo: **Treeint**
- um conjunto de construtores: **empty**, **node**

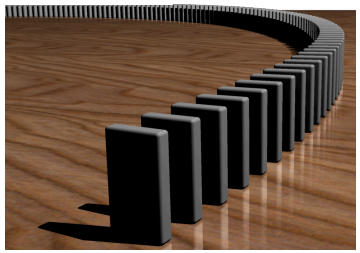
empty: () → TreeInt

node: TreeInt × Integer × TreeInt → TreeInt

```
/* TreeInt.h */
typedef struct TreeInt TreeInt;
TreeInt* empty();
TreeInt* node(TreeInt *l, int elem, TreeInt *r);
```

Indução Matemática





Definição indutiva de algoritmos

The Smaller-Subproblem Principle

If we can reduce a problem to a smaller subproblem, we can call the procedure that solves the problem to solve the subproblem.

- A definição de algoritmos que operam sobre tipos de dados de tipo indutivo, funcionam por análise de casos no construtores e decomposição do problema em problemas mais pequenos (valores mais pequenos).
- Dado um valor v qualquer de um tipo T, o algoritmo
  - verifica-se qual é o último construtor aplicado na construção de v (por exemplo, v é cons(3, cons(2,nil)));
  - extraem-se os componentes aos quais o construtor foi aplicado (neste caso, o inteiro 3 e a lista cons(2,nil));
  - aplica-se recursivamente aos componentes de tipo T. Assim, obtêm-se os resultados intermédios que permitem produzir o resultado final.

Definição indutiva de algoritmos

- Tipo **ListInt**
- Construtores: **nil**, **cons**

- Algoritmo length(L) para calcular o **comprimento** de uma lista qualquer L:

se L é da forma **nil**
  length(L) ≜ 0
se L é da forma **cons**(x,L')
  length(L) ≜ 1+length(L')

```
type listInt = Nil
             | cons of int * listInt

let rec length l = match l with
            Nil -> 0
          | Cons(x,l) -> 1+(length l)
```


Definição indutiva de algoritmos

- Tipo **ListInt**
- Construtores: **nil**, **cons**

- Algoritmo sum(L) para calcular a **soma** dos valores numa lista qualquer L:

se L é da forma **nil**
  sum(L) ≜ 0
se L é da forma **cons**(x,L')
  sum(L) ≜ x+sum(L')

```
type listInt = Nil
             | cons of int * listInt

let rec sum l = match l with
            Nil -> 0
          | Cons(x,l) -> 1+(sum l)
```

Definição indutiva de algoritmos

- Tipo **TreeInt**
- Construtores: **empty**, **node**
- Algoritmo numnodes(T) para calcular o **número de nós** numa árvore qualquer T:

se T é da forma **empty**

numnodes(L) ≜ 0

se T é da forma **node**(L', x, L'')

numnodes(L) ≜ 1+numnodes(L')+numnodes(L'')

```
type treeInt = Empty
  | Node of treeInt * int * treeInt
let rec numnodes t = match t with
        Empty -> 0
      | Node(l,x,r) -> 1+(numnodes l)+(numnodes r)
```


Definição indutiva de algoritmos

- Tipo **TreeInt**
- Construtores: **empty**, **node**
- Algoritmo depth(T) para calcular a altura de uma árvore qualquer T:

se T é da forma **empty**

depth(T) ≜ 0

se T é da forma **node**(L',x,L'')

depth(T) ≜ 1+**max**(depth(L'), depth(L''))

```
type treeInt = Empty
  | Node of treeInt * int * treeInt
let rec depth t = match t with
        Empty -> 0
      | Node(l,x,r) -> 1+(max (depth l) (depth r))
```


Análise inductiva de algoritmos

- Propriedades sobre os algoritmos definidos inductivamente na estrutura dos dados podem ser analisadas usando indução matemática.
- Ex: Será que uma árvore binária construída pela função insert, é uma árvore binária de pesquisa?

```
type treeInt = Empty
  | Node of treeInt * int * treeInt
let rec depth t = match t with
        Empty -> 0
      | Node(l,x,r) -> 1+(max (depth l) (depth r))
let rec insert t y = match t with
        Empty -> Node(Empty,y,Empty)
      | Node(l,x,r) -> if x <= y then Node(insert l y,x,r)
                       else Node(l,x,insert r y)
```


Definição indutiva de programas

- O conjunto de todos os programas de uma linguagem de programação pode ser visto como um tipo de dados
- É fácil definir indutivamente o conjunto de todos os programas de uma linguagem de programação
- A definição indutiva de linguagens melhorou o desenho das mesmas (Fortran/spaghetti code vs. Pascal/estrutura em blocos)

- Discussão:
  definição indutiva de algoritmos versus definição indutiva de programas
  (a que se referem as duas ocorrências do adjectivo "indutivo"?)

Definição indutiva de programas

Copyright Notice

The following manuscript

EWD 215: A Case against the GO TO Statement

was published as a letter entitled

Go-to statement considered harmful

in *Commun. ACM 11* (1968), 3: permission.

A Case against the GO TO Statement.

by Edsger W.Dijkstra
Technological University
Eindhoven, The Netherlands

Since a number of years I am familiar with the obse
quality of programmers is a decreasing function of the d
statements in the programs they produce. Later I discove
the go to statement has such disastrous effects and did
that the go to statement should be abolished from all "h
programming languages (i.e. everything

Estruturação hierárquica de linguagens

```
10 for I=1 to 10 do
20 if I<10 goto 40
25 for J=I to 20 do
30 next I
40 next J
```

```
for i=1 to 10 do
   begin
      if i<10 then
         begin

         end
   end;
```


Exemplo: A Linguagem CALC

- CALC é uma linguagem simples de expressões aritméticas.
- Cada programa da linguagem CALC é uma expressão algébrica construída com base em numerais inteiros, nos quatro operadores aritméticos básicos (+, −, ×, ÷) e nos parêntesis.
- Exemplos:

```
(21+32) * 42
2/(7-2)
```

- A semântica pretendida para a linguagem CALC é a esperada para uma linguagem de expressões aritméticas:

  a denotação de cada expressão da linguagem CALC é o seu valor.

Semântica da linguagem CALC

Em geral, a semântica de uma linguagem pode ser caracterizada por uma função I que atribui um significado (ou denotação) a cada programa (ou fragmento de programa) sintacticamente correcto.

- No caso da linguagem CALC:

$$I : CALC \rightarrow Integer$$

CALC = conjunto dos programas válidos

Integer = conjunto dos significados (denotações)

Como representar a linguagem CALC como um tipo de dados?

A Linguagem CALC (como tipo indutivo)

- Tipo de dados CALC com os construtores: **num**, **add**, **mul**, **div**, **sub**

num: Integer → CALC
add: CALC × CALC → CALC
mul: CALC × CALC → CALC
div: CALC × CALC → CALC
sub: CALC × CALC → CALC

- Cada valor do tipo indutivo CALC representa uma expressão na linguagem CALC. Diz-se que o tipo indutivo CALC define a sintaxe abstracta da linguagem CALC
- A sintaxe concreta de uma linguagem:
  - Caracteriza a forma como as suas expressões e programas são efectivamente escritos em termos de sequências de caracteres, formatação, etc …
- A sintaxe abstracta de uma linguagem:
  - Caracteriza a estrutura das suas expressões e programas, apresentando

Sintaxe abstracta vs Sintaxe concreta

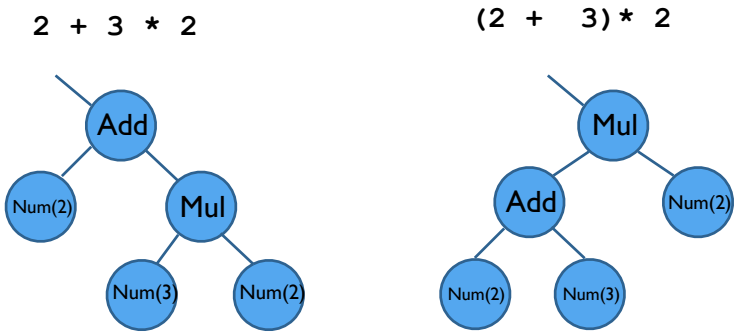
- Constantes inteiras
  - em decimal: 12
  - em hexadecimal: 0x0C
  - sintaxe abstracta: num(12)
- Variáveis
  - em Pascal: A
  - em bash: %A
  - sintaxe abstracta: var("A")
- Afectação
  - em C: x = 2
  - em Pascal: x := 2
  - sintaxe abstracta: assign(var("x"),num(2))

Sintaxe abstracta vs Sintaxe concreta

- Expressões algébricas
  - em C: 2*3+2
  - em RPN: 2 3 * 2 +
  - em Lisp: (+ (* 2 3) 2)
  - sintaxe abstracta: add(mul(num(2),num(3)),num(2))
- Blocos
  - em C: {S1 S2 … Sn }
  - em Pascal: begin S1; S2; …; Sn end
  - sintaxe abstracta: block(S1,S2,…,Sn)
- Ciclo while:
  - em C: while (C) S
  - em Pascal: while C do S
  - sintaxe abstracta: while(C,S)

Árvore Sintáctica Abstracta

- Representa a estrutura de uma frase da linguagem em termos dos seus constructores abstractos.
- Abstract Syntax Tree (AST)



Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2010-2011

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Unidade 3: Semântica Operacional

Os interpretadores e compiladores são algoritmos que aceitam programas sintacticamente válidos e produzem denotações (valores, efeitos, ou outros programas). Os interpretadores têm como resultado um valor ou efeito, os compiladores têm como resultado um programa numa linguagem de uma máquina.

Como podemos definir o processamento de programas?

- Definição composicional de Semântica operacional.
- Algoritmo interpretador de CALC: implementação usando uma linguagem orientada por objectos (Java)
- Algoritmo compilador de CALC: introdução à máquina virtual CLR

Semântica de CALC

- A função semântica I pode ser definida por um algoritmo que "sabe como interpretar" todos os programas sintacticamente correctos de uma linguagem, determinando o seu valor ou efeito

$$I : CALC \rightarrow Integer$$

CALC = conjunto dos programas válidos

Integer = conjunto dos significados (denotações)

Interpretadores

- Um interpretador para uma linguagem de programação é um algoritmo que atribui um valor ou efeito a cada programa legítimo da linguagem
- Os programas fornecidos como dados de entrada a um interpretador são representados por valores da sintaxe abstracta da linguagem a que pertencem
- Um algoritmo interpretador pode ser definido indutivamente na sintaxe abstracta da linguagem

Interpretadores

- Um interpretador para uma linguagem de programação é um algoritmo que atribui um valor ou efeito a cada programa legítimo da linguagem
- Os programas fornecidos como dados de entrada a um interpretador são representados por valores da sintaxe abstracta da linguagem a que pertencem
- Um algoritmo interpretador pode ser definido indutivamente na sintaxe abstracta da linguagem

Resumindo:

- Programas exprimem algoritmos que processam dados
- Programas também podem ser vistos como dados
- Um interpretador é um programa que processa programas

A Linguagem CALC (como tipo indutivo)

- Tipo de dados CALC com os construtores: num, add, mul, div, sub

num: Integer → CALC
add: CALC × CALC → CALC
mul: CALC × CALC → CALC
div: CALC × CALC → CALC
sub: CALC × CALC → CALC

A Linguagem CALC (como tipo indutivo)

- Tipo de dados CALC com os construtores: num, add, mul, div, sub

num: Integer → CALC
add: CALC × CALC → CALC
mul: CALC × CALC → CALC
div: CALC × CALC → CALC
sub: CALC × CALC → CALC

```
type calc =  Num of int
           | Add of calc * calc
           | Mul of calc * calc
           | Div of calc * calc
           | Sub of calc * calc
```


Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

$$eval : CALC \rightarrow Integer$$

se E é da forma **num**(n):	eval(E) = n
se E é da forma **add**(E',E''):	v1 = eval(E'); v2 = eval(E''); eval(E) ≜ v1+v2
se E é da forma **mul**(E',E''):	v1 = eval(E'); v2 = eval(E''); eval(E) ≜ v1*v2
se E é da forma **sub**(E',E''):	v1 = eval(E'); v2 = eval(E''); eval(E) ≜ v1-v2
se E é da forma **div**(E',E''):	v1 = eval(E'); v2 = eval(E''); eval(E) ≜ v1/v2

Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

$$eval : CALC \rightarrow Integer$$

eval(num(n))	≜ n
eval(add(E',E"))	≜ eval(E') + eval(E")
eval(mul(E',E"))	≜ eval(E') * eval(E")
eval(sub(E',E"))	≜ eval(E') - eval(E")
eval(div(E',E"))	≜ eval(E') / eval(E")

[notação mais legível, usando pattern matching]

Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

$$eval : CALC \rightarrow Integer$$

eval(num(n)) ≜ n

```
let rec eval e = match e with
      Num(n) -> n
    | Add(e1,e2) -> (eval e1)+(eval e2)
    | Mul(e1,e2) -> (eval e1)*(eval e2)
    | Sub(e1,e2) -> (eval e1)-(eval e2)
    | Div(e1,e2) -> (eval e1)/(eval e2)
```


Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

$$eval : CALC \rightarrow Integer$$

- Note bem: a função de interpretação eval(-) é definida recursivamente na estrutura do seu argumento!
- Semântica composicional: o significado do todo só depende do significado das suas partes
- O mesmo não acontece nas linguagens "naturais":
  - time flies like an arrow
  - fruit flies like a banana

Semântica Operacional Estrutural

- Sintaxe (abstracta) definida por um tipo indutivo, apresentada por um conjunto de construtores;
- Semântica definida por um algoritmo interpretador, que atribui um significado (valor) a cada expressão da linguagem, calculando-o composicionalmente a partir do significado das suas subexpressões;
- A esta técnica de definição da semântica de uma linguagem de programação chama-se:

Semântica Operacional Estrutural

Implementação em Java

- Usando uma linguagem baseada em objectos, um tipo de dados indutivo pode ser representado por uma interface (que representa o tipo indutivo), e por um conjunto de classes (em que cada classe representa um construtor do tipo indutivo)
- A interface pode declarar uma ou mais operações sobre o tipo indutivo, por exemplo:

```
public interface CALC {
     int eval();
}
```

Ou seja, eval: CALC → Integer

Implementação em Java

- Cada classe representa um (e um só) dos construtores do tipo indutivo
- Cada classe fornece a implementação relativa ao construtor que representa, para cada operação definida sobre o tipo indutivo

```
public class Num implements CALC {
     private int value ;
     Num(int v) { value = v; }
     int eval() { return value; }
}
```

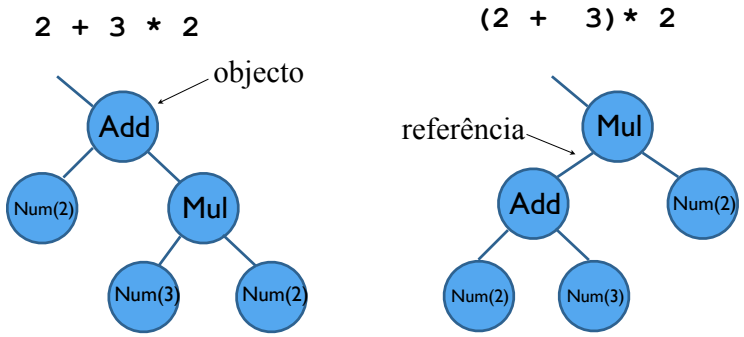

Implementação em Java

- Cada expressão é representada por uma árvore (n-ária) de objectos (uma AST);
- Cada construtor do tipo indutivo é aplicado chamando o construtor da classe respectiva:

```
CALC expr1 = new Add(new Num(2),new Num(3));
int result1 = expr1.eval();
CALC expr2 =
     new Add(new Sub(new Num(2),new Num(3)),new Num(3));
int result2 = expr2.eval();
```

Árvore Sintáctica Abstracta

- Representa a estrutura de uma frase da linguagem em termos dos seus constructores abstractos.
- Abstract Syntax Tree (AST)



Implementação em Java

- No nosso exemplo, temos as classes **`Num`**, **`Add`**, **`Sub`**, **`Mul`** e **`Div`**, implementando os construtores da linguagem **num**, **add**, **sub**, **mul** e **div**.
- A definição das operações fica "dispersa" pelas várias classes (é mais cómodo acrescentar novos construtores a uma linguagem do que novas operações)

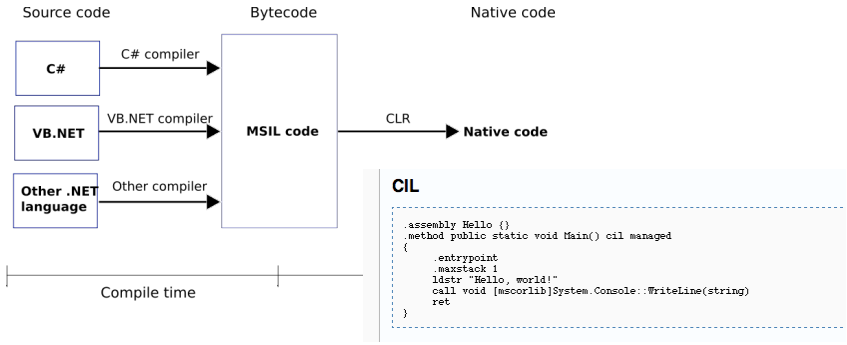
```
public class Add implements CALC {
     private CALC lhs;
     private CALC rhs;
     Add(CALC l, CALC r) { lhs = l; rhs = r; }
     int eval() { return lhs.eval()+rhs.eval(); }
}
```


Compiladores

- Um compilador para uma linguagem de programação é um algoritmo que traduz cada programa legítimo da linguagem numa versão do mesmo program directamente executável por uma "máquina".
- A máquina pode ser física (Pentium) ou virtual (JVM, CLR)
- O destino do programa gerado pode ser outro compilador já existente
  [exemplo: tradutor de Java em C para ser processado pelo gcc]
- A tradução feita por um compilador preserva a semântica do programa.
- Um algoritmo compilador pode ser definido indutivamente na sintaxe abstracta da linguagem

Common Language Runtime (Microsoft, 2000)

- Máquina de pilha, base para a plataforma Microsoft .NET
- Desenhada para executar programas de várias linguagens.
- CIL - Common Intermediate Language

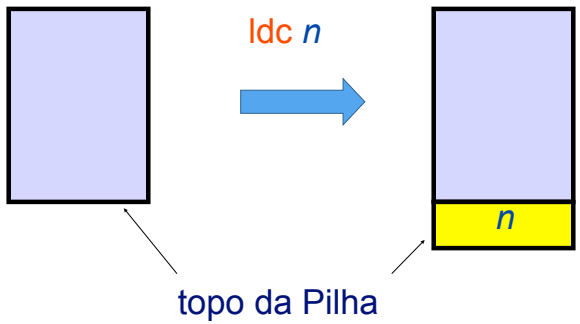


Common Language Runtime

- É uma máquina de pilha: todas as instruções consomem argumentos do topo da Pilha, e deixam um resultado no topo da Pilha (***stack machine***)
- "Primeiras" (5) instruções da linguagem CIL:
  - ldc *n* : Carrega o valor n no topo da pilha
  - add : Retira dois valores inteiros do topo da pilha e coloca na pilha a sua soma
  - mul : idem para a multiplicação
  - div : idem para a divisão
  - sub : idem para a subtracção

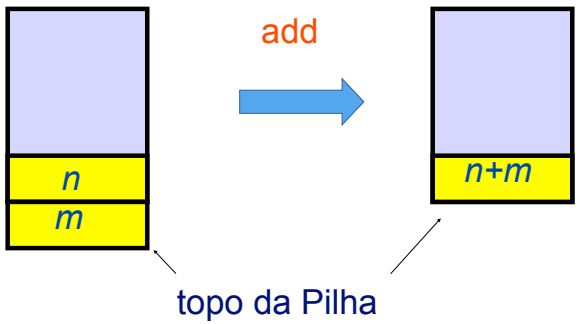
Common Language Runtime

- "Primeiras" (5) instruções: ldc *n*, add, mul, div, sub.
- Load Constant (ldc *n*)



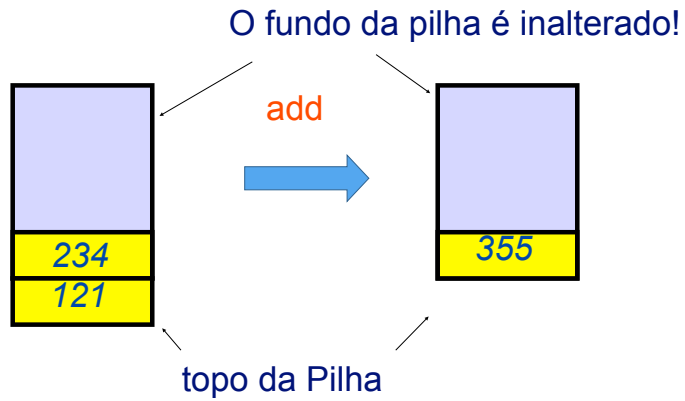
Common Language Runtime

- "Primeiras" (5) instruções: ldc n, add, mul, div, sub.
- Add (add)



Common Language Runtime

- "Primeiras" (5) instruções: ldc n, add, mul, div, sub.
- Add (add)



Compilador de CALC

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALC numa sequência de instruções CLR

comp : CALC → CodeSeq

se E é da forma **num**(n):	comp(E) ≜ < **ldc.i4 n** >
se E é da forma **add**(E',E''):	s1 = comp(E'); s2 = comp(E''); comp(E) ≜ s1 @ s2 @ < **add** >
se E é da forma **mul**(E',E''):	v1 = comp(E'); v2 = comp(E''); comp(E) ≜ s1 @ s2 @ < **mul** >
se E é da forma **sub**(E',E''):	v1 = comp(E'); v2 = comp(E''); comp(E) ≜ s1 @ s2 @ < **sub** >
se E é da forma **div**(E',E''):	v1 = comp(E'); v2 = comp(E''); comp(E) ≜ s1 @ s2 @ < **div** >

Compilador de CALC

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALC numa sequência de instruções CLR

comp : CALC → CodeSeq

comp(**num**(n)) ≜ < **ldc.i4 n** >

comp(**add**(E',E'')) ≜ comp(E') @ comp(E'') @ < **add** >

comp(**mul**(E',E'')) ≜ comp(E') @ comp(E'') @ < **mul** >

comp(**sub**(E',E'')) ≜ comp(E') @ comp(E'') @ < **sub** >

comp(**div**(E',E'')) ≜ comp(E') @ comp(E'') @ < **div** >

Compilador de CALC

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALC numa sequência de instruções CLR

comp : CALC → CodeSeq

comp(**num**(n)) ≜ < **ldc.i4 n** >

```
let rec comp e = match e with
      Num(n) -> ["ldc.i4 "^(string_of_int n)]
    | Add(e1,e2) -> (comp e1)@(comp e2)@["add"]
    | Mul(e1,e2) -> (comp e1)@(comp e2)@["mul"]
    | Sub(e1,e2) -> (comp e1)@(comp e2)@["sub"]
    | Div(e1,e2) -> (comp e1)@(comp e2)@["div"]
```

Correcção do Compilador

- Algoritmo comp(E) para traduzir uma expressão E qualquer da linguagem CALC numa sequência de instruções da linguagem CIL (CLR)

$$comp : CALC \rightarrow CodeSeq$$

- **Propriedade de Correcção:** Quando a sequência de instruções comp(E) é executada num estado da máquina virtual em que a pilha está no estado *p*, quando termina deixa sempre a máquina no estado push(*v*,*p*), em que *v* é o valor da expressão E.

Common Language Runtime

- "Primeiras" (5) instruções: ldc ***n***, add, mul, div, sub.
- Comp("2+2*(7-2)")
- Comp(add(num(2),mul(num(2),sub(num(7),num(2)))))

```
ldc.i4 2
ldc.i4 2
ldc.i4 7
ldc.i4 2
sub
mul
add
```

Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2010-2011

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Unidade 4: Ligação e Âmbito

Os identificadores são a primeira ferramenta básica para criar abstracções numa linguagem de programação. Um identificador usado numa expressão (ou programa) representa uma subexpressão cuja definição é feita separadamente. O significado da expressão contendo identificadores deve ser o mesmo da expressão em que os identificadores são substituídos pelas subexpressões que eles representam.

- Literais e identificadores
- Declaração de identificadores
- Âmbito de uma declaração
- Ocorrências de um identificador (livres, ligadas e ligantes)
- Expressões abertas e fechadas
- Construção fundamental **decl** id=E **in** E **end**.
- Linguagem com identificadores (declaração e ocorrências ligadas): CALCI.
- Algoritmo interpretador com substituição
- Algoritmo interpretador com ambiente
- Algoritmo compilador com ambiente

Constantes e Identificadores

- Constantes (ou literais)
  - Referem entidades ou valores bem determinados em qualquer contexto onde ocorram
  - Nas linguagens "naturais" correspondem aos "nomes próprios".
  - Linguagem ML: `true, false, []`
  - Linguagem C: `1, 1.0, 0xFF, "hello", int`
- Identificadores (ou nomes)
  - Referem entidades ou valores que dependem do contexto
  - Nas linguagens "naturais" correspondem aos "pronomes".
  - Linguagem Java: `x, Count, System.out`
  - Linguagem C: `printf`

Ligação e Âmbito

- Os literais e os identificadores denotam sempre uma entidade bem determinada e inalterável.
- A entidade denotada por um literal (ou o valor de um literal) é determinada pelo próprio literal (`23`,`"hi!"`, etc).
- A associação entre um identificador e a entidade ou valor denotado chama-se ligação (*binding*)
- Em geral, a ligação entre identificador e entidade denotada estabelece-se num certo contexto sintáctico e é introduzida por uma declaração
- Ao contexto sintáctico onde uma ligação tem efeito chama-se o âmbito (*scope*) da ligação

Ligação e Âmbito

- O identificador **x** denota uma variável de estado (célula de memória)

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```


Ligação e Âmbito

- O identificador **x** denota uma variável de estado (célula de memória)

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```

Âmbito da ligação

Ligação e Âmbito

- O identificador **j** denota uma variável de estado (célula de memória)

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```


Ligação e Âmbito

- O identificador **j** denota uma variável de estado (célula de memória)

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```

Âmbito da ligação

Elementos de um âmbito

- A ligação entre um identificador e a respectiva entidade por este denotada (valor, posição de memória, etc) envolve os seguintes ingredientes:
  - Uma (única!) ocorrência ligante:
    em geral, corresponde à declaração do identificador.
  - O âmbito da ligação
    A "parte/região/zona/fragmento" do programa onde a ligação em causa tem efeito
  - Várias (zero ou mais) ocorrências ligadas
    todas as ocorrências do identificador, distintas da ocorrência ligante, que existem dentro do âmbito

Ocorrências ligantes e ligadas

- Ocorrências do identificador **x**

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```

Ocorrências ligantes

Ocorrências ligantes e ligadas

- Ocorrências do identificador **x**

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```

Ocorrências ligadas

Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```

Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```


Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```


Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```


Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```

Ocorrências livres

- Uma ocorrência de identificador que não é ligada nem ligante diz-se **livre**

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j+y;
       z += x;
     }
     return z;
}
```


Expressões Abertas e Fechadas

- Uma subexpressão diz-se **aberta** se contém ocorrências livres de identificadores
- Uma subexpressão diz-se **fechada** se não contém ocorrências livres de identificadores
- Exemplos de expressões abertas:

```
void f(int x)                                   C
{    int i;
     for(int i=0;i<TEN;i++) x+=i;
     printf("%d\n",x);
}
```

```
let x=1 in (f x)                              OCaml
```


Expressões Abertas e Fechadas

- Uma subexpressão diz-se **aberta** se contém ocorrências livres de identificadores
- Uma subexpressão diz-se **fechada** se não contém ocorrências livres de identificadores
- Exemplos de expressões abertas:

```
void f(int x)                  ocorrência livre   C
{    int i;
     for(int i=0;i<TEN;i++) x+=i;
     printf("%d\n",x);
}
```

```
                          ocorrência livre
let x=1 in (f x)                              OCaml
```


Semântica de expressões abertas

- A denotação de uma subexpressão de programa só pode ser calculada se se conhecer a denotação de cada identificador que nela ocorra livre.
- A definição de uma semântica composicional para linguagens com declarações de identificadores tem necessariamente que considerar expressões abertas.
  Por exemplo, a expressão OCaml

```
let x = 2 in (x+x)
```

  é fechada mas contém uma subexpressão aberta.

- Um programa (fragmento fechado) pode conter no seu interior expressões abertas.

  [Dê exemplos de linguagens de programação onde seja possível compilar um programa aberto]

Ambiente

- Um programa fechado fornece necessariamente ligações para todas as ocorrências livres de identificadores que ocorram nas suas subexpressões (através de declarações).

Para cada subexpressão $\mathcal{E}$ de um programa $\mathcal{P}$, ao conjunto de todas as ligações no âmbito das quais $\mathcal{E}$ ocorre chama-se o **ambiente** de $\mathcal{E}$ em $\mathcal{P}$.

Ambiente (Quiz)

- Qual o ambiente da subexpressão "`x+1`"?

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j;
       z+=x;
     }
     return z;
}
```


Ambiente (Quiz)

- Qual o ambiente da subexpressão "`z+=x`"?

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j;
       z+=x;
     }
     return z;
}
```


Ambiente (Quiz)

- Qual o ambiente da subexpressão "`return z`"?

```
int f(int x)
{
     int z = x+1;
     for(int j=0; j<10; j++){
       int x=j;
       z+=x;
     }
     return z;
}
```

Exemplo: A Linguagem CALCI

- A linguagem CALCI estende a linguagem CALC com a possibilidade de se poderem introduzir e usar identificadores usando a construção **declare**:

```
decl Id = Expressão1 in Expressão2 end
```

Numa expressão **decl**, a primeira ocorrência de *Id* é ligante, no âmbito definido pela *Expressão2*

- Definimos os programas CALCI como sendo as expressões fechadas da linguagem CALCI.

```
decl x=2 in decl y=x+2 in (x+y) end end
```


A Linguagem CALCI (como tipo indutivo)

- Tipo de dados CALCI com os construtores: num, add, mul, div, sub, id, decl

num:	Integer → CALCI
id:	String → CALCI
add:	CALCI × CALCI → CALCI
mul:	CALCI × CALCI → CALCI
div:	CALCI × CALCI → CALCI
sub:	CALCI × CALCI → CALCI
decl:	String × CALCI × CALCI → CALCI

A Linguagem CALCI (como tipo indutivo)

- Tipo de dados CALCI com os construtores: num, add, mul, div, sub, id, decl

num:	Integer → CALCI
id:	String → CALCI
add:	CALCI × CALCI → CALCI

```
type calci =  Num of int
            | Id of string
            | Add of calci * calci
            | Mul of calci * calci
            | Div of calci * calci
            | Sub of calci * calci
            | Decl of string * calci * calci
```


Semântica de CALCI (1)

- A função semântica I de CALCI pode ser definida por um **algoritmo** que "sabe como interpretar" **todas** as expressões de CALCI, determinando o seu **valor ou efeito**.

$$I : \text{CALCI} \rightarrow \text{Integer}$$

CALCI = conjunto das expressões fechadas

Integer = conjunto dos significados (denotações)

Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

$$eval : CALCI \rightarrow Integer$$

```
eval( num(n) )          ≙ n
eval( add(E1,E2) )      ≙ eval(E1) + eval(E2)
        …
eval( decl(s, E1, E2) ) ≙ [ G = subst(s, E2,E1);
                            eval(G); ]
```

Intuitivamente: o significado da expressão contendo identificadores deve ser o mesmo da expressão em que os identificadores são substituídos pelas subexpressões que eles representam.

A Função Subst

subst(s, E, F)

Calcula a expressão que resulta de substituir todas as ocorrências livres do identificador s pela expressão F na expressão E.

subst(s, s+s+2, y+z) **=** (y+z)+(y+z)+2

subst(y, decl x=y in decl y=2 in x+y, u) **=** decl x=u in decl y=2 in x+y

Definição da Função Subst

```
subst(s, num(n), F )      ≙ num(n);
subst(s, id(s), F )       ≙ F;
subst(s, add(E1, E2), F ) ≙ add( subst(s, E1, F), subst(s, E2 , F));
…
subst(s, decl(s, E1, E2), F) ≙ [/* caso s = s' */ G = subst(s, E1, F);
                                                    decl(s, G, E2); ]
subst(s, decl(s', E1, E2), F) ≙ [/* caso s ≠ s' */ G = subst(s, E1, F);
                                                    decl(s', G, subst(s, E2, F)); ]
```


Definição da Função Subst

```
subst(s, num(n), F )      ≙ num(n);
subst(s, id(s), F )       ≙ F;
```

```
let rec subst s e f = match e with
        Num(n) -> Num(n)
      | Add(e1,e2) -> Add(subst s e1 f, subst s e2 f)
      | ...
      | Decl(t,e1,e2) -> if s = t then
                           Decl(s,subst s e1 f,e2)
                         else
                           Decl(t,subst s e1 f,subst s e2 f)
```

Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

$$eval : CALCI \rightarrow integer$$

eval(**num**(*n*))	≜ *n*
eval(**add**(E1,E2))	≜ **eval**(E1) + **eval**(E2)
…	
eval(**decl**(s, E1, E2))	≜ **eval**(**subst**(s, E2, E1));]

Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

$$eval : CALCI \rightarrow integer$$

eval(**num**(*n*)) ≜ *n*

```
let rec eval e = match e with
       Num(n) -> n
     | Add(e1,e2) -> (eval e1)+(eval e2)
     | ...
     | Decl(s,e1,e2) -> eval (subst s e2 e1)
```


Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

$$eval : CALCI \rightarrow integer$$

eval(**num**(*n*)) ≜ *n*

```
let rec eval e = match e with
       Num(n) -> n
     | Add(e1,
     | ...
     | Decl(s,e1,e2) -> eval (subst s e2 e1)
```

porque será que não é preciso programar o caso do identificador??

Semântica de CALCI (2)

- A semântica da linguagem CALCI baseada em substituições é muito conveniente do ponto de vista da especificação pois é muito simples.

$$eval : CALCI \rightarrow Integer$$

- Já do ponto de vista operacional, é conveniente definir uma semântica mais concreta, recorrendo à manipulação de ambientes.
- A manipulação de ambientes também é mais conveniente como técnica de implementação de interpretadores (como estruturas de dados auxiliares).

Semântica de CALCI (2)

A função semântica I de CALCI pode ser definida por um **algoritmo** interpretador para expressões de CALCI, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

$$I : \text{CALCI} \times \text{ENV} \rightarrow \text{Integer}$$

CALCI = programas abertos

ENV = ambientes

Integer = significados (denotações)

Semântica de CALCI (2)

A função semântica I de CALCI pode ser definida por um **algoritmo** interpretador para expressões de CALCI, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

$$I : \text{CALCI} \times \text{ENV} \rightarrow \text{integer}$$

CALCI = programas abertos

```
let rec eval e env = match e with
        Num(n) -> n
      | Id(s) -> find s env
      | Add(e1,e2) -> (eval e1 env)+(eval e2 env)
      | ...
      | Decl(s,e1,e2) -> ...
```


Semântica de CALCI (2)

A função semântica I de CALCI pode ser definida por um **algoritmo** interpretador para expressões de CALCI, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

$$I : \text{CALCI} \times \text{ENV} \rightarrow \text{integer}$$

CALCI = programas abertos

```
let rec eval e env = match e with
        Num(n) -> n
      | Id(s) -> find s env
      | Add(e1,e2) -> (eval e1 env)+(eval e2 env)
      | ...
      | Decl(s,e1,e2) -> let v1 = (eval e1 env) in
                         let nenv = (s,v1)::env in
                          eval e2 nenv
```

Ambiente "mutável"

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável ao estilo Object-Oriented.
- Numa linguagem estruturada, com âmbitos encaixados hierarquicamente, a adição e remoção de ligações entre identificadores e valores segue uma disciplina LIFO.
- Um ambiente guarda as associações correspondentes a um determinado âmbito (e todos os âmbitos envolventes). A partir de um ambiente pode criar-se um novo nível, correspondendo a um âmbito encaixado.
- Os ambientes têm duas operações fundamentais:

```
Environ BeginScope()
```

  - que cria um novo nível local vazio, onde serão colocadas as novas ligações.
  - Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível (Porquê?)

```
Environ EndScope()
```

  - que coloca o ambiente no estado anterior à última operação BeginScope().

Ambiente "mutável"

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável ao estilo Object-Oriented.
- Outras operações fundamentais:

```
void Assoc(String id, Value val)
```

  - Adiciona uma nova ligação que associa ao identificador id o valor val indicado.
  - A ligação é adicionada ao último nível (mais recente) do ambiente.
- Devolve o valor associado ao identificador id no ambiente.

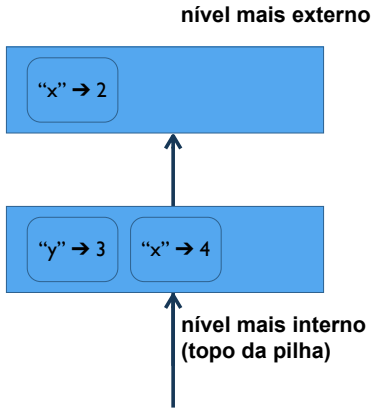
```
Value Find(String id)
```

  - A pesquisa é efectuada do nível mais "recente" para o mais "antigo", de modo a respeitar o encaixe dos âmbitos das declarações.

A "interface" Ambiente

- Simule mentalmente:

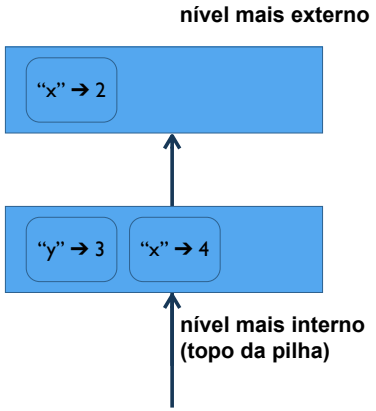
```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");          // devolve 2
env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);
val = env.Find("y");          // devolve 3
val = env.Find("x");          // devolve 4
env=env.EndScope()
val = env.Find("x")           // devolve 2
```



A "interface" Ambiente

- Implementado como pilha de dicionários …

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");          // devolve 2
env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);
val = env.Find("y");          // devolve 3
val = env.Find("x");          // devolve 4
env=env.EndScope()
val = env.Find("x")           // devolve 2
```



Interpretador de CALCI

- Algoritmo eval(E, *env*) para calcular o valor de uma expressão E da linguagem CALCI:

$$eval : CALCI \times ENV \rightarrow Integer$$

```
eval( num(n) , env)        ≜ n
eval( id(s) , env)         ≜ env.Find(s)
eval( add(E1,E2) , env)    ≜ eval(E1, env) + eval(E2, env)
…
eval( decl(s, E1, E2), env) ≜ [ v1 = eval(E1, env);
                                env = env.BeginScope();
                                env.Assoc(s, v1);
                                val = eval(E2, env);
                                env = env.EndScope();
                                val ]
```


Erros de execução

- O que é que acontece se não for encontrado o identificador no ambiente?
  A que corresponde essa falha?
  A um erro de execução do tipo "identificador não declarado".
- A função não está definida para os programas em que há ocorrências de identificadores sem declaração.
- Que outros tipos de erros de execução podem ocorrer?
- São o mesmo tipo de erros? É possível evitar uns e outros não?

Compilador de CALCI

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

$$comp : CALCI \rightarrow CodeSeq$$

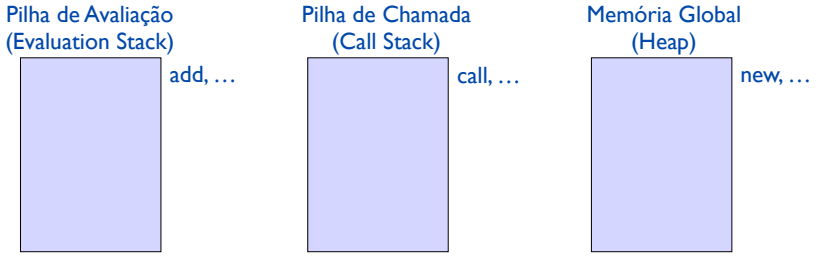
No algoritmo interpretador para expressões abertas é possível saber o valor de um identificador usando o nome.

O processo de compilação deve depender o menos possível de nomes do domínio dos programas. Onde for possível, deve ser estabelecida uma ligação entre os identificadores e um endereço de memória.

Ideia geral: Usar um vector de inteiros na pilha de execução da CLR para guardar os valores dos identificadores declarados por uma expressão.

Common Language Runtime

- Estruturas principais da máquina virtual CLR
  - ES: guarda resultados temporários durante a avaliação de expressões;
  - CS: guarda variáveis locais e endereços de retorno;
  - Heap: guarda objectos e outras entidades dinâmicas



Variáveis locais em CIL

- A máquina virtual CLR prevê uma pilha de execução para as funções e métodos definidas em CIL (call stack) e uma pilha de execução manipulada pelo código de cada método (execution stack).
- Podemos declarar variáveis locais a cada método que são guardadas na pilha de chamada.

```
.assembly 'Calc' { }
.module Calc.exe
.method ... void Main ()  cil managed
{ .entrypoint
  .locals init (int32 v_0)
  ...
```

- Instruções da linguagem CIL para manipular variáveis locais
  - stloc *label* : Retira o valor no topo da pilha e coloca-o na variável *label*
  - ldloc *label*: Carrega o valor da variável *label* no topo da pilha (não altera a variável)

Variáveis locais em CIL

- A máquina virtual CLR prevê uma pilha de execução para as funções e métodos definidas em CIL (call stack) e uma pilha de execução manipulada pelo código de cada método (execution stack).
- As variáveis podem ser referidas pela sua posição relativa nas declarações. A contagem começa em 0 (zero).

```
.assembly 'Calc' { }
.module Calc.exe
.method ... void Main ()  cil managed
{ .entrypoint
  .locals init (int32,int32)
  ...
```

- Instruções da linguagem CIL para manipular variáveis locais
  - stloc *n* : Retira o valor no topo da pilha e coloca-o na variável de posição *n*
  - ldloc *n*: Carrega o valor da variável *n* no topo da pilha (não altera a variável)

Vectores em CIL

- A máquina virtual CLR permite a manipulação primitiva de arrays. Um array é manipulado por referência.
- Instruções da linguagem CIL para manipular vectores primitivos
  - newarr *T*: Retira da pilha um valor inteiro (*n*) e coloca na pilha uma referência para um vector com o tamanho *n*.
  - stelem.i4: Retira da pilha uma referência para um vector (*a*), um inteiro (*n*) e um valor (*v*) e modifica a posição *n* do vector *a*, com o valor *v*.
  - ldelem.i4: Retira da pilha uma referência para um vector (*a*) e um inteiro (*n*) e coloca na pilha o elemento que está na posição *n* do vector *a* (Não altera os conteúdos do vector).

Vectores em CIL

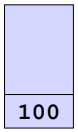
- Exemplo...

```
ldc.i4 100
newarr int32
dup       // array (a)
ldc.i4 0  // index (i)
ldc.i4 1  // value (v)
stelem.i4 // a[i] = v

dup       // array (a)
ldc.i4 99 // index (i)
ldelem.i4 // a[i]
...
```

Vectores em CIL

- Exemplo...

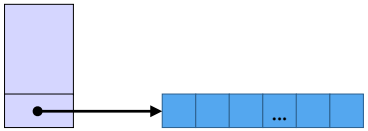
```
ldc.i4 100
newarr int32
dup       // array (a)
ldc.i4 0  // index (i)
ldc.i4 1  // value (v)
stelem.i4 // a[i] = v

dup       // array (a)
ldc.i4 99 // index (i)
ldelem.i4 // a[i]
...
```



Vectores em CIL

- Exemplo...

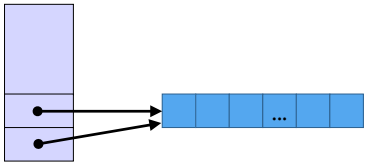
```
ldc.i4 100
newarr int32
dup       // array (a)
ldc.i4 0  // index (i)
ldc.i4 1  // value (v)
stelem.i4 // a[i] = v

dup       // array (a)
ldc.i4 99 // index (i)
ldelem.i4 // a[i]
...
```



Vectores em CIL

- Exemplo...

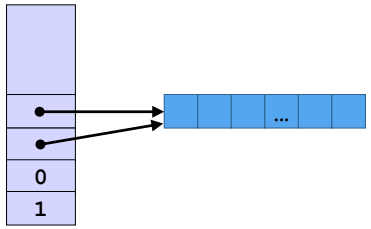
```
ldc.i4 100
newarr int32
dup       // array (a)
ldc.i4 0  // index (i)
ldc.i4 1  // value (v)
stelem.i4 // a[i] = v

dup       // array (a)
ldc.i4 99 // index (i)
ldelem.i4 // a[i]
...
```



Vectores em CIL

- Exemplo...

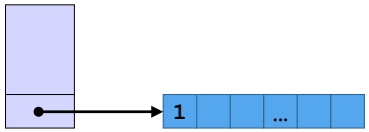
```
ldc.i4 100
newarr int32
dup       // array (a)
ldc.i4 0  // index (i)
ldc.i4 1  // value (v)
stelem.i4 // a[i] = v

dup       // array (a)
ldc.i4 99 // index (i)
ldelem.i4 // a[i]
...
```



Vectores em CIL

- Exemplo...

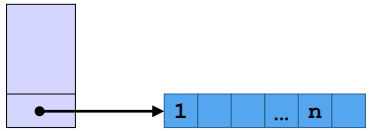
```
ldc.i4 100
newarr int32
dup       // array (a)
ldc.i4 0  // index (i)
ldc.i4 1  // value (v)
stelem.i4 // a[i] = v

dup       // array (a)
ldc.i4 99 // index (i)
ldelem.i4 // a[i]
...
```



Vectores em CIL

- Exemplo...

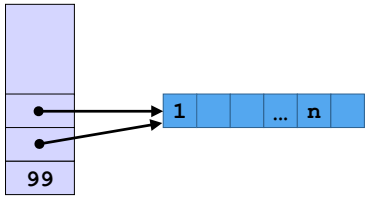
```
ldc.i4 100
newarr int32
dup       // array (a)
ldc.i4 0  // index (i)
ldc.i4 1  // value (v)
stelem.i4 // a[i] = v

dup       // array (a)
ldc.i4 99 // index (i)
ldelem.i4 // a[i]
...
```



Vectores em CIL

- Exemplo...

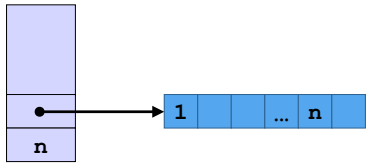
```
ldc.i4 100
newarr int32
dup       // array (a)
ldc.i4 0  // index (i)
ldc.i4 1  // value (v)
stelem.i4 // a[i] = v

dup       // array (a)
ldc.i4 99 // index (i)
ldelem.i4 // a[i]
...
```



Vectores em CIL

- Exemplo...

```
ldc.i4 100
newarr int32
dup       // array (a)
ldc.i4 0  // index (i)
ldc.i4 1  // value (v)
stelem.i4 // a[i] = v

dup       // array (a)
ldc.i4 99 // index (i)
ldelem.i4 // a[i]
...
```



Compilador de CALCI

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

comp : CALCI → CodeSeq

Ideia geral: Usar um vector de inteiros na pilha de execução da CLR para guardar os valores dos identificadores declarados por uma expressão.

```
decl
  x = 1
in
  decl
    y = 2
  in
    x + y
  end
end
```

```
.locals init (int32[] table)
                  // inicialização
ldc.i4 2          // número de slots
newarr int32
stloc table       // guardar o vector

...
```


Compilador de CALCI

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

comp : CALCI → CodeSeq

Ideia geral: Usar um vector de inteiros na pilha de execução da CLR para guardar os valores dos identificadores declarados por uma expressão.

```
decl
  x = 1
in
  decl
    y = 2
  in
    x + y
  end
end
```

```
// x = 1
ldloc table  // vector
ldc.i4 0     // índice
ldc.i4 1     // valor
stelem.i4

...
```


Compilador de CALCI

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

comp : CALCI → CodeSeq

Ideia geral: Usar um vector de inteiros na pilha de execução da CLR para guardar os valores dos identificadores declarados por uma expressão.

```
decl
  x = E
in
  decl
    y = 2
  in
    x + y
  end
end
```

```
// x = 1
ldloc table  // vector
ldc.i4 0     // índice
[[ E ]]      // valor
stelem.i4

...
```

Compilador de CALCI

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

$$comp : CALCI \rightarrow CodeSeq$$

Ideia geral: Usar um vector de inteiros na pilha de execução da CLR para guardar os valores dos identificadores declarados por uma expressão.

```
decl
  x = 1
in
  decl
    y = 2
  in
    x + y
  end
end
```

```
// x = 1
...
// y = 2
ldloc table  // vector
ldc.i4 1     // índice
ldc.i4 2     // valor
stelem.i4
```


Compilador de CALCI

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

$$comp : CALCI \rightarrow CodeSeq$$

Ideia geral: Usar um vector de inteiros na pilha de execução da CLR para guardar os valores dos identificadores declarados por uma expressão.

```
decl
  x = 1
in
  decl
    y = E
  in
    x + y
  end
end
```

```
// x = 1
...
// y = 2
ldloc table  // vector
ldc.i4 1     // índice
[[ E ]]      // valor
stelem.i4
```


Compilador de CALCI

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

$$comp : CALCI \rightarrow CodeSeq$$

Ideia geral: Usar um vector de inteiros na pilha de execução da CLR para guardar os valores dos identificadores declarados por uma expressão.

```
decl
  x = 1
in
  decl
    y = 2
  in
    x + y
  end
end
```

```
...
// x + y
ldloc table // vector
ldc.i4 0    // índice
ldelem.i4   // valor
ldloc table // vector
ldc.i4 1    // índice
ldelem.i4   // valor
add
```


Compilador de CALCI

```
decl
  x = 1
in
  decl
    y = 2
  in
    x + y
  end
end
```

```
.locals init (int32[] table)
// inicialização
ldc.i4 2        // número de slots
newarr int32
stloc table     // guardar o vector
// x = 1
ldloc table  // vector
ldc.i4 0     // índice
ldc.i4 1     // valor
stelem.i4
// y = 2
ldloc table  // vector
ldc.i4 1     // índice
ldc.i4 2     // valor
stelem.i4
// x + y
ldloc table // vector
ldc.i4 0    // índice
ldelem.i4   // valor
ldloc table // vector
ldc.i4 1    // índice
ldelem.i4   // valor
add
```

Compilador de CALCI

```
decl
   x = 1
in
   decl
      y = 2
   in
      x + y
   end
end
```

```
.locals init (int32[] table)
// inicialização
ldc.i4 2          // número de slots
newarr int32
stloc table       // guardar o vector
// x = 1
ldloc table  // vector
ldc.i4 0     // índice
ldc.i4 1     // valor
stelem.i4
// y = 2
ldloc table  // vector
ldc.i4 1     // índice
ldc.i4 2     // valor
stelem.i4
// x + y
ldloc table // vector
ldc.i4 0    // índice
ldelem.i4   // valor
ldloc table // vector
ldc.i4 1    // índice
ldelem.i4   // valor
add
```


Compilador de CALCI

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

$$comp : CALCI \rightarrow CodeSeq$$

Dúvida geral: Como saber o "endereço" para cada identificador?

```
decl
   x = 1
in
   decl
      y = 2
   in
      x + y
   end
end
```


Compilador de CALCI

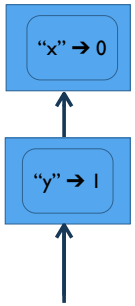
- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

$$comp : CALCI \times ENV \rightarrow CodeSeq$$

Dúvida geral: Como saber o "endereço" para cada identificador?

Usando um ambiente que associa identificadores a

```
decl
   x = 1
in
   decl
      y = 2
   in
      x + y
   end
end
```



Compilador de CALCI

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

$$comp : CALCI \times ENV \rightarrow CodeSeq$$

Dado um ambiente **env**

se E é da forma **num**(n):	comp(E,env) ≜ < **ldc.i4** n >
se E é da forma **add**(E',E''):	s1 = comp(E',env); s2 = comp(E'',env); comp(E,env) ≜ s1 @ s2 @ < **add** >
...	
se E é da forma **id**(s):	[addr = env.Find(s) < **ldloc table** > @ < **ldc.i4** addr > @ < **ldelem.i4** >]

Compilador de CALCI

- Algoritmo comp(E) para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

comp : CALCI x ENV → CodeSeq

Dado um ambiente **env**

...

se E é da forma **decl**(s,E',E"): **[** s1 = comp(E',env);
nenv = env.BeginScope();
addr = env.getNumDecls();
nenv.Assoc(s,addr);
s2 = comp(E",nenv);
env.EndScope();

O próximo endereço livre é igual ao número de declarações já feitas.

< `ldloc table` **>** @ **<** `ldc.i4` `addr` **>** @ s1 @ **<** `stelem.i4` **>** @ s2 **]**

Ambiente "mutável" para compilação

- A disciplina de resolução de nomes é a mesma do algoritmo interpretador, a adição e remoção de ligações entre identificadores e endereços segue a mesma disciplina LIFO.
- O endereço de cada identificador é atribuído em função dos endereços que já foram atribuídos anteriormente.
- Os ambientes continuam a ter as duas operações fundamentais:

  Environ BeginScope()

  – que cria um novo nível local vazio, onde serão colocadas as novas ligações.
  – Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível.

  Environ EndScope()

  – que coloca o ambiente no estado anterior à última operação BeginScope().

Ambiente "mutável" para compilação

- A disciplina de resolução de nomes é a mesma do algoritmo interpretador, a adição e remoção de ligações entre identificadores e endereços segue a mesma disciplina LIFO.
- Outras operações fundamentais:

  void Assoc(String id, int addr)

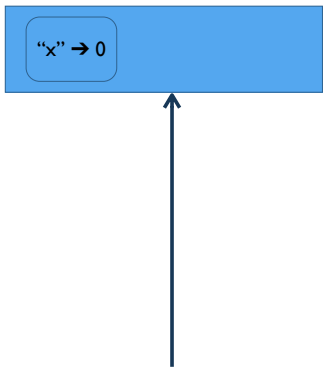
  – Adiciona uma nova ligação que associa ao identificador id ao endereço addr.
  – A ligação é adicionada ao último nível (mais recente) do ambiente.
- Devolve o endereço associado ao identificador id no ambiente.

  int Find(String id)

  – A pesquisa é efectuada do nível mais "recente" para o mais "antigo", de modo a respeitar o encaixe dos âmbitos das declarações.

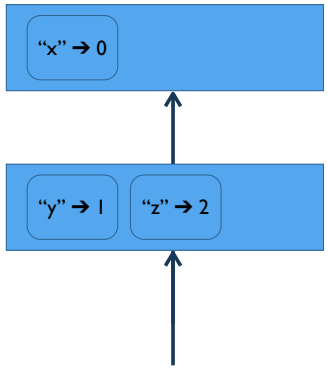
Exemplo

```
decl
  x = 1
in
  decl
    y = 2
    z = 3+x
  in
    x + y
  end
  +
  decl
    x = 3
    w = 2+x
  in
    x + w
  end
end
```



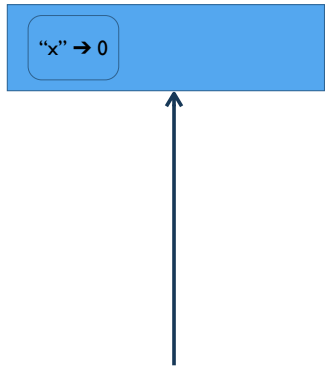
Exemplo

```
decl
  x = 1
in
  decl
    y = 2
    z = 3+x
  in
    x + y
  end
  +
  decl
    x = 3
    w = 2+x
  in
    x + w
  end
end
```



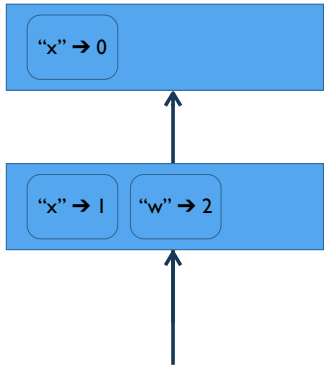
Exemplo

```
decl
  x = 1
in
  decl
    y = 2
    z = 3+x
  in
    x + y
  end
  +
  decl
    x = 3
    w = 2+x
  in
    x + w
  end
end
```



Exemplo

```
decl
  x = 1
in
  decl
    y = 2
    z = 3+x
  in
    x + y
  end
  +
  decl
    x = 3
    w = 2+x
  in
    x + w
  end
end
```



Exemplo

```
decl
  x = 1
in
  decl
    y = 2
    z = 3+x
  in
    x + y
  end
  +
  decl
    x = 3
    w = 2+x
  in
    x + w
  end
end
```

Partilham os mesmos endereços na memória

Implementação em ML

```
let rec comp e env = match e with
          Num(n) -> ["ldc.i4 "^(string_of_int n)]

        | Id(x) -> let addr = find x env in
                           ["ldloc table";
                            "ldc.i4 "^(string_of_int addr);
                            "ldelem.i4"]

        | Add(e1,e2) -> (comp e1 env)@(comp e2 env)@["add"]

        | Mul(e1,e2) -> (comp e1 env)@(comp e2 env)@["mul"]

        | Sub(e1,e2) -> (comp e1 env)@(comp e2 env)@["sub"]

        | Div(e1,e2) -> (comp e1 env)@(comp e2 env)@["div"]

        | Decl(s,e1,e2) -> let addr = free_addr env in
                                          let s1 = comp e1 env in
                                          let s2 = comp e2 ((s,addr)::env) in
                                          ["ldloc table";
                                           "ldc.i4 "^(string_of_int addr)]@
                                          s1@
                                          ["stelem.i4"]@
                                          s2;;
```


Mini Quiz (para entregar)

```
decl
  x = 1
  y = 3
in
  decl
    x = y+x
  in
    3+x
  end
  +
  x
end
```

1. Qual o ambiente de interpretação da expressão “3+x” assinalada?
2. e da expressão “x” assinalada?
3. Qual o ambiente de compilação para as duas expressões?
4. Qual o código resultante da compilação?

Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2010-2011

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Unidade 5: Linguagens Imperativas

As expressões das linguagens consideradas até agora denotam sempre valores puros. Até agora os identificadores denotam sempre o mesmo valor ao longo da execução de um programa. No entanto, o paradigma de programação dominante é o paradigma imperativo, caracterizado pela mutação de estado (C, Java).

As operações fundamentais das linguagens imperativas são:

- A associação de identificadores a localizações de memória (var x:Integer)
- A modificação do conteúdo de localizações de memória (x := 2)

- Modelo de memória (cell: set e get)
- Ambiente versus memória
- Aliasing
- L-value e R-value
- Tempo de vida vs âmbito
- Manipulação de memória por apontadores, referências, etc
- Estrutura das linguagens imperativas. Família Algol vs família ML
- Sintaxe separada de comandos e expressões
- Zonas de memória (stack/heap)
- Representação interna de valores e objectos

Modelo de Memória

- Memória:
  é um conjunto (potencialmente infinito) de células cujo conteúdo é mutável.
- Cada célula de memória tem um designador único (a referência da célula) e pode conter qualquer valor da linguagem.
- As referências são valores de um tipo de dados especial ref que só podem ser usados no contexto da memória a que dizem respeito.
- Operações primitivas sobre uma memória $\mathcal{M}$

new:	$\mathcal{M}$ × **void** → **ref**
set:	$\mathcal{M}$ × **ref** × Value → **void**
get:	$\mathcal{M}$ × **ref** → Value
free:	$\mathcal{M}$ × **ref** → **void**

Modelo de Memória

- Operações sobre uma memória $\mathcal{M}$

new: $\mathcal{M}$ × **void** → **ref**

Devolve uma referência para uma nova célula livre, e define-a como estando "em uso".

set: $\mathcal{M}$ × **ref** × Value → **void**

Altera o conteúdo da célula referida para o valor indicado. O valor "antigo" perde-se irremediavelmente.

get: $\mathcal{M}$ × **ref** → Value

Devolve o valor contido na célula referida.

free: $\mathcal{M}$ × **ref** → **void**

Define a célula referida como estando livre, devolvendo-a ao gestor de memória, para ser reciclada.

Ambiente versus Memória

- Um ambiente indica a denotação de cada identificador declarado num programa e reflecte a estrutura estática do programa.
- A associação estabelecida no ambiente entre um identificador e o seu valor denotado é fixa e imutável dentro do âmbito respectivo.
- A memória agrega o conteúdo das variáveis de estado mutáveis, indicando o valor contido em cada localização (ou referência).
- Uma variável de estado é visível nos programas através de identificadores.
- A associação entre o identificador de uma variável de estado e a sua localização de memória é imutável e é mantida pelo ambiente.

Ambiente versus Memória

Ambiente

Identificador	Valor
PI	3.14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Memória

Localização	Valor
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

Ambiente versus Memória

Ambiente

Identificador	Valor
PI	3.14
x	0x00FF
k	0x0100
j	0x0100
TEN	10

Memória

Endereço	Valor
0x00FF	25
0x0100	12
0x0102	0x0100
...	...
0xFFFF	0

Propriedades do modelo de memória

Ambiente

Identificador	Valor
PI	3.14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Memória

Localização	Valor
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

Uma mesma célula de memória pode ser referida por vários identificadores distintos (aliasing).

Aliasing

- Dois identificadores diferentes que referem a mesma localização de memória.

```
class A {
  int x;
  boolean equals(A a) { return x == a.x}
}
A a = new A(); a.equals(a);


int x = 0;
void f(int* y) { *y = x+1; }
...
f(&x);
// x = ?
```


Propriedades do modelo de memória

Ambiente

Identificador	Valor
PI	3.14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Memória

Localização	Valor
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

Uma célula pode conter uma referência para outra célula, permitindo a construção de estruturas de dados dinâmicas.

Operações Imperativas nas linguagens

- Reserva e inicialização de uma célula nova dada uma expressão E qualquer

var(E)

- Pode ser encontrada de diversas formas:

```
{
  int a;
  MyClass m;
  …
}

new int[10];

malloc(sizeof(int));

new MyClass();
```

em Java tem um significado
em C++ tem outro,
qual a diferença?

Operações Imperativas nas linguagens

- Afectação de um valor a uma variável dadas as expressões E e F

E := F

A expressão E denota uma referência para uma célula, F é uma expressão qualquer

- Pode ser encontrada de diversas formas:

```
a = 1

i := 2

b[x+2][b[x-2]] = 2

*(p+2) = y

myTable(i,j) = myTable(j,i)

Readln(MyLine);
```

Operações Imperativas nas linguagens

- Desreferenciação de uma célula de memória dada uma expressão E que denota uma referência para uma célula.

!E

- Pode ser encontrada de diversas formas:

```
i := !i + 1          (linguagem ML)
*p                   (linguagem C)
i = i + 1            (linguagem C)
i++                  (linguagem C)
```


Operações Imperativas nas linguagens

- Desreferenciação de uma célula de memória dada uma expressão E que denota uma referência para uma célula.

!E

- Pode ser encontrada de diversas formas:

```
i := !i + 1          (linguagem ML)
*p                   (linguagem C)
i = i + 1            (linguagem C)
i++                  (linguagem C)
```

referências

Operações Imperativas nas linguagens

- Desreferenciação de uma célula de memória dada uma expressão E que denota uma referência para uma célula.

!E

- Pode ser encontrada de diversas formas:

```
i := !i + 1          (linguagem ML)
*p                   (linguagem C)
i = i + 1            (linguagem C)
i++                  (linguagem C)
```

valor

L-Value e R-Value

- Se uma expressão E tem por valor uma referência, a maior parte das linguagens de programação interpreta E de forma dependente do contexto

E := 2

- (Left-Value) À "esquerda" do símbolo de afectação, denota o seu valor efectivo (que é uma referência)

E := E + 1

- (Right-Value) À "direita" do símbolo de afectação, denota o conteúdo da célula referida, evitando-se escrever a desreferenciação explícita

E := !E + 1

L-Value e R-Value

- Se uma expressão E tem por valor uma referência, a maior parte das linguagens de programação interpreta E de forma dependente do contexto.

```
A[A[2]] := A[2] + 1
```

- A terminologia “L-Value” e “R-Value” não é muito feliz. Por exemplo, na expressão acima as duas subexpressões da forma A[2], uma à esquerda e outra à direita, são ambas desreferenciadas implicitamente.

Desreferenciação

- A operação de desreferenciação !E torna a interpretação dos programas mais precisa e evita qualquer ambiguidade.

```
A[!A[2]] := !A[2] + 1
```

- Por outro lado, pode argumentar-se que torna os programas mais difíceis de ler.

A desreferenciação implícita pode ser vista como uma operação de coerção (conversão ou cast).

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

  const int k = 2;
  int a = k;
  int b = a + 2;
  …
  b = a * b
  …
}
```

```
decl
  k = 2
  a = newvar(k)
  b = newvar(!a+2)
in
  …
  b := !a * !b
  …
  free(a);
  free(b)
end
```

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

  const int k = 2;
  int a = k;
  int b = a + 2;
  …
  b = a * b
  …
}
```

```
decl
  k = 2
  a = newvar(k)
  b = newvar(!a+2)
in
  …
  b := !a * !b
  …
  free(a);
  free(b)
end
```

libertação implícita (das células atribuídas aos ids a e b)

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

  int k = 2;
  int *a = &k;
  … *a = k+*a …

}
```

```
decl
  k = newvar(2)
  a = newvar(k)
in
  … !a := !k+!!a …
  free(k);
  free(a)
end
```


Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

  int k = 2;
  const int *a = &k;
  int b = *a;
  … *a = k+b …

}
```

```
decl
  k = newvar(2)
  a = k
  b = newvar(!a)
in
  … a := !k+!b …
  free(k);
  free(b)
end
```


Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

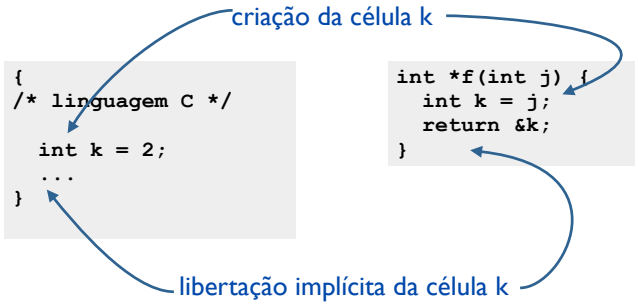
```
{
/* linguagem C */

  int k = 2;
  int *a = &k;
  … k = k+*a …

}
```

```
decl
  k = newvar(2)
  a = newvar(k)
in
  … k := !k+!!a …
  free(k);
  free(a);
end
```

Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **var(_)** e a sua libertação usando **free(_)**.

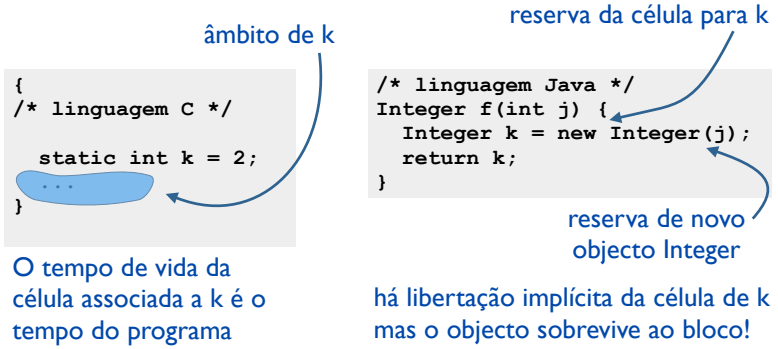
- Em muitas situações, o tempo de vida da célula concide com o âmbito do(s) seu(s) identificador.



Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **var(_)** e a sua libertação usando **free(_)**.

- Noutras situações, o tempo de vida da célula extravasa o âmbito do(s) seu(s) identificador.



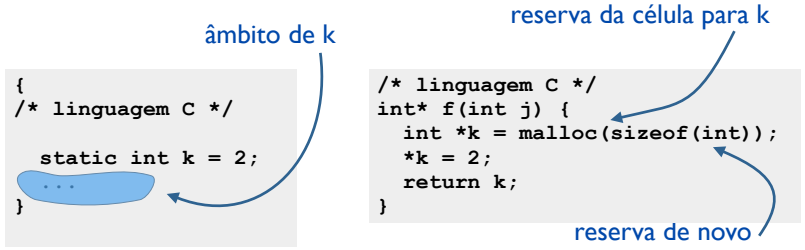
O tempo de vida da célula associada a k é o tempo do programa

há libertação implícita da célula de k mas o objecto sobrevive ao bloco!

Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **var(_)** e a sua libertação usando **free(_)**.

- Noutras situações, o tempo de vida da célula extravasa o âmbito do(s) seu(s) identificador.



O tempo de vida da célula associada a k é o tempo do programa

há libertação implícita da célula de k mas a memória reservada predura.

Linguagens Imperativas

Linguagens da família do ALGOL (Pascal, C, …)

Assumem como princípio de desenho uma separação muito clara, logo ao nível sintáctico, entre expressões e comandos

- Expressões:
  - Denotam valores puros (inteiros, booleanos, funções)
  - A avaliação de expressões não deve ter efeitos (laterais)
- Comandos:
  - Denotam efeitos (na memória)
  - Um comando é executado pelo efeito que produz na memória: representa uma acção.

Uma linguagem de tipo ALGOL

- Definida com base em duas categorias sintácticas: Expressões (EXP) e comandos (COM):

num:	Integer → EXP
bool:	Boolean → EXP
id:	String → EXP
add:	EXP × EXP → EXP
and:	EXP × EXP → EXP
if:	EXP × COM × COM → COM
while:	EXP × COM → COM
assign:	EXP × EXP → COM
seq:	COM × COM → COM
var:	String × EXP × COM → COM
const:	String × EXP × COM → COM

Linguagens Imperativas

Linguagens da família do ML

Todas as construções pertencem a uma única categoria sintáctica, de expressões.

- Nestas linguagens, qualquer expressão pode potencialmente produzir um efeito lateral…
- Por exemplo, em OCAML a afectação x := E é uma expressão (de tipo **unit** (a.k.a. **void**)).
- N.B. Existem linguagens que combinam conceitos! Por exemplo, a linguagem C, contém expressões e comandos: a afectacão (x = y) é uma expressão, e expressões podem produzir efeitos (i++).

Uma linguagem tipo ML (microML)

- Consideramos uma só categoria sintáctica para expressões (EXP):

num:	Integer → EXP	
bool:	Boolean → EXP	
id:	String → EXP	
add:	EXP × EXP → EXP	
var:	EXP → EXP	
deref:	EXP → EXP	(!x)
if:	EXP × EXP × EXP → EXP	
while:	EXP × EXP → EXP	
assign:	EXP × EXP → EXP	(x := y + z)
seq:	EXP × EXP → EXP	(S1 ; S2)
decl:	String × EXP × EXP → EXP	

Semântica de microML

A semântica de uma linguagem imperativa pode ser caracterizada por uma função I que dá uma denotação a todos os programas abertos dado um ambiente e uma memória.

$$I : P \times ENV \times MEM \rightarrow VAL \times MEM$$

P = Fragmentos de programa (abertos)
ENV = Ambientes (funções ID → VAL)
MEM = Memórias
VAL = Valores (Denotações)

O conjunto das denotações possíveis:

$$Val = Boolean \cup Integer \cup Ref$$

Esta função traduz a intuição que, em geral, um fragmento de programa P produz um valor e gera um efeito (na memória).

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( add(E1, E2) , env , m0) ≜ [ (v1 , m1) = eval( E1, env, m0);
                                  (v2 , m2) = eval( E2, env, m1);
                                  (v1 + v2 , m2) ]
eval( and(E1, E2) , env , m0) ≜ [ (v1 , m1) = eval( E1, env, m0);
                                  (v2 , m2) = eval( E2, env, m1);
                                  (v1 & v2 , m2) ]
```


Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( var(E) , env , m0) ≜ [ (v1 , m1) = eval( E, env, m0 );
                             (ref, m2) = m1.new(v1);
                             (ref, m2) ]
eval( deref(E) , env , m0) ≜ [ (ref , m1) = eval( E, env, m0);
                               (m1.get(ref) , m1) ]
eval( assign(E1, E2) , env , m0) ≜ [(v1 , m1) = eval( E1, env, m0);
                                    (v2 , m2) = eval( E2, env, m1);
                                    m3 = m2.set(v1, v2) ;
                                    (v2 , m3) ]
```


Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( seq(E1, E2) , env , m0) ≜ [(v1 , m1) = eval( E1, env, m0);
                                 (v2 , m2) = eval( E2, env, m1);
                                 (v2 , m2) ]
eval( if(E1, E2, E3) , env , m0) ≜
                                 [(v1 , m1) = eval( E1, env, m0);
                                  if (v1 = T) then (v2 , m2) = eval( E2, env, m1);
                                             else (v2 , m2) = eval( E3, env, m1);
                                  (v2 , m2) ]
```


Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( while(E1, E2) , env , m0) ≜
                                 [(v1 , m1) = eval( E1, env, m0 );
                                  if (v1 = T) then  [ (v2 , m2) = eval( E2, env, m1);
                                             (v,m1) =eval( while(E1, E2), m2) ]
                                             else (F , m1) ]
```

iteração interpretada em termos de recursão.

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( decl(s, EI, EB) , env , m0) ≜
                    [(v1 , m1) = eval( EI, env, m0 );
                     env = env.BeginScope();
                     env.Assoc(s, v1);
                     (v2 , m2) = eval(EB, env, m1);
                     env = env.EndScope();
                     (v2 , m2) ]
```


Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

```
decl a = newvar(2) in
decl b = newvar(!a) in
decl c = a in
(
  a := !b + 2;
  c := !c + 2
)
```

a e c são aliases (sinónimos), ou seja, referem a mesma célula de memória.

Compilador de microML

- Algoritmo comp(E) para traduzir uma expressão E qualquer de microML numa sequência de instruções CLR

$$comp : P \times ENV \rightarrow CodeSeq$$

P = Fragmentos de programas (abertos) microML

ENV = Ambiente que atribui aos identificadores posições na pilha de chamada (endereços)

CodeSeq = Sequências de instruções

Compilador de microML

Como estender o compilador de CALCI para microML...

- O domínio dos valores é diferente... tem booleanos e referências...
  - Um booleano pode ser implementado com um inteiro...
  - Uma referência é um inteiro que denota uma célula de memória, é um endereço. É um endereço diferente daqueles atribuídos aos identificadores (na stack).
- O tempo de vida das variáveis microML extravasa o âmbito dos identificadores, logo, as variáveis não podem ser guardadas na stack...
- Duas alternativas:
  - um vector com conteúdos acessível a todo o programa em que o índice é o endereço da célula.
  - usar um objecto para cada célula de memória e a referência (Java) desse objecto representa o endereço da célula de memória. O vector é a própria heap da máquina virtual.

Ideia geral: encapsular valores em objectos que podem ser guardados directamente na stack (constantes) e na memória (células na heap).

Boxing and Unboxing

- A máquina virtual CLR tem operações primitivas para converter valores primitivos em objectos das classes da biblioteca que os representam.

`bool`	`System.Boolean`
`char`	`System.Char`
`string`	`System.String`
`int32`	`System.Int32`

- Instruções da linguagem CIL para encapsular valores primitivos
  - box *type*: Retira o valor primitivo do topo da pilha e coloca uma referência para um objecto correspondente ao tipo *type* no topo da pilha.
  - unbox *type*: Retira a referência do objecto que está no topo da pilha e coloca um apontador para o valor primitivo no topo da pilha.
  - ldobj *type*: Retira um apontador para um valor do tipo *type* e deixa o valor primitivo no topo da pilha.

Boxing and Unboxing

- A máquina virtual CLR tem operações primitivas para converter valores primitivos em objectos das classes da biblioteca que os representam.

```
.locals init (object)
 ldc.i4 1
 box int32
 stloc.0
 ldloc.0
 unbox int32
 ldobj int32
```

- Instruções da linguagem CIL para encapsular valores primitivos
  - box *type*: Retira o valor primitivo do topo da pilha e coloca uma referência para um objecto correspondente ao tipo *type* no topo da pilha.
  - unbox *type*: Retira a referência do objecto que está no topo da pilha e coloca um apontador para o valor primitivo no topo da pilha.
  - ldobj *type*: Retira um apontador para um valor do tipo *type* e deixa o valor primitivo no topo da pilha.

Instruções de controlo

- Dentro de um método CIL, o controlo de um programa pode ser feito através de saltos (absolutos ou condicionais). Em CIL o destino dos saltos são marcados com etiquetas (labels).

```
L0:ldloc.0
   brfalse L1
   ...
   br L0
L1:nop
```

- Instruções da linguagem CIL:
  - brfalse *label*: Desempilha o topo da pilha de execução e continua a execução a partir da instrução indicada pela etiqueta *label*, se o valor desempilhado for zero.
  - brtrue *label*: Desempilha o topo da pilha de execução e continua a execução a partir da instrução indicada pela etiqueta *label*, se o valor desempilhado for diferente de zero.
  - br *label*: Salto incondicional, continua a execução a partir da instrução com a

Modelo de objectos

- A máquina virtual possui um modelo de objectos primitivo, permite a criação de objectos, a chamada de métodos de objectos e métodos de classes (estáticos).
- Instruções CIL
  - newobj instance *constructor-spec:* Retira da pilha os argumentos indicados no protótipo do constructos e deixa no topo da pilha uma referência para um novo objecto da classe indicada, criado no Heap.

```
newobj instance void Cell::.ctor(int32)
```

  - callvirt *method-spec:* Invoca o método indicado pelo protótipo, retira da pilha o objecto e os argumentos, o resultado (se houver) é deixado no topo da pilha.

```
ldloc.4
ldc.i4.s 2
callvirt instance int Cell::set(int32)
```

  - call *method-spec*: Invoca o método estático indicado pelo protótipo.

Compilador de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

$$comp : microML \times ENV \rightarrow CodeSeq$$

Dado um ambiente **env**

se E é da forma **num**(n):	comp(E,env) ≜ < **ldc.i4** n > @ < **box int32** >

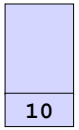
se E é da forma **add**(E',E''):	s1 = comp(E',env);
	s2 = comp(E'',env);
	comp(E,env) ≜
	s1@<**unbox int32**>@<**ldobj int32**>@
	s2 @<**unbox int32**>@<**ldobj int32**>@
	<**add**>@ < **box int32** >

Exemplo

```
ldc.i4 10
box int32        10
unbox int32
ldobj int32
ldc.i4 20
box int32        20
unbox int32
ldobj int32
add
box int32
```

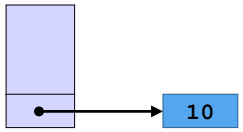

Exemplo

```
ldc.i4 10
box int32
unbox int32
ldobj int32
ldc.i4 20
box int32
unbox int32
ldobj int32
add
box int32
```



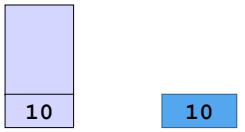
Exemplo

```
ldc.i4 10
box int32
unbox int32
ldobj int32
ldc.i4 20
box int32
unbox int32
ldobj int32
add
box int32
```



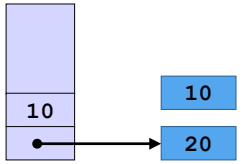
Exemplo

```
ldc.i4 10
box int32
unbox int32
ldobj int32
ldc.i4 20
box int32
unbox int32
ldobj int32
add
box int32
```



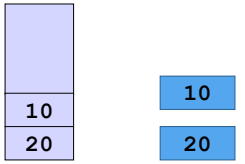
Exemplo

```
ldc.i4 10
box int32
unbox int32
ldobj int32
ldc.i4 20
box int32
unbox int32
ldobj int32
add
box int32
```



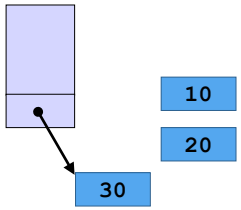
Exemplo

```
ldc.i4 10
box int32
unbox int32
ldobj int32
ldc.i4 20
box int32
unbox int32
ldobj int32
add
box int32
```



Exemplo

```
ldc.i4 10
box int32
unbox int32
ldobj int32
ldc.i4 20
box int32
unbox int32
ldobj int32
add
box int32
```



Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

$$comp : microML \times ENV \rightarrow CodeSeq$$

Dado um ambiente **env**

se E é da forma **bool(** v **)**: n = v?1:0;

comp(E,env) ≜ `<ldc.i4 n>@< box bool>`

se E é da forma **and**(E',E''): s1 = comp(E',env);

s2 = comp(E'',env);

comp(E,env) ≜

`s1@<unbox bool>@<ldobj bool>@`

`s2 @<unbox bool>@<ldobj bool>@`

`< and>@ < box bool>`

Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

$$comp : microML \times ENV \rightarrow CodeSeq$$

se E é da forma **var(** E' **)**:

s1 = comp(E', env);

comp(E, env) ≜ s1@ < `newobj instance void Cell::.ctor(object)`>

se E é da forma **deref(** E' **)**:

s1 = comp(E', env);

comp(E, env) ≜ s1 @ < `callvirt instance object Cell::get()`>

se E é da forma **assign(** E',E'' **)**:

s1 = comp(E1, env);

s2 = comp(E2, env);

comp(E, env) ≜ s1@s2@`<callvirt instance object Cell::set(object)>`

deixa no topo da pilha o valor passado!!

Implementação da "Memória"

Representação das células de memória por objectos implementados no ambiente de suporte à execução (***Runtime support system***), programado em C# (poderia ser programado directamente em CIL).

```
public class Cell {

        object value;

        public Cell(object v) { value = v; }

        public object get() { return value; }

        public object set(object v) { value = v; return v;}
}
```

Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

$$comp : microML \times ENV \rightarrow CodeSeq$$

se E é da forma **seq**(E', E''): s1 = comp(E', env);
s2 = comp(E'', env);
comp(E , env) ≜ s1@<pop> @ s2 ;

descarta o valor no topo da pilha!

se E é da forma **if**(E', E'', E'''):

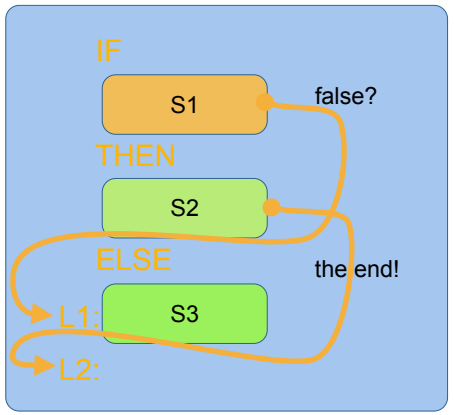
s1 = comp(E', env); s2 = comp(E'', env); s3 = comp(E''', env);

comp(E , env) ≜ s1 @ < **brfalse L1** > @
s2 @ < **br L2** > @
< **L1: nop** > @ s3 @
< **L2: nop** >

Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

$$comp : microML \times ENV \rightarrow CodeSeq$$




```
   [[ S1 - condition ]]
   brfalse L1:
   [[ S2 - then ]]
   br L2
L1:nop
   [[ S3 - else ]]
L2:nop
```


Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

$$comp : microML \times ENV \rightarrow CodeSeq$$

- Para compilar várias expressões encaixadas são necessárias etiqueta diferentes (um par por cada if)

```
if
  A
then if
       B
     then
       C
     else
       D
else
  E
```

```
   [[ A - condition ]]
   brfalse L1:
   [[ B - condition 2 ]]
   brfalse L3:
   [[ C - then 2 ]]
   br L4
L3:nop
   [[ D - else 2 ]]
L4:nop
   br L2
L1:nop
   [[ E - else ]]
L2:no
```


Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

$$comp : microML \times ENV \rightarrow CodeSeq$$

se E é da forma **while**(E', E''):

s1 = comp(E', env); s2 = comp(E'', env);

comp(E , env) ≜ < **L1: nop** > @ s1 @ < **brfalse L2** > @
s2 @ <**pop**>@ < **br L1** > @
< **L2:** **ldc.i4 0** >@ < **box int32** >

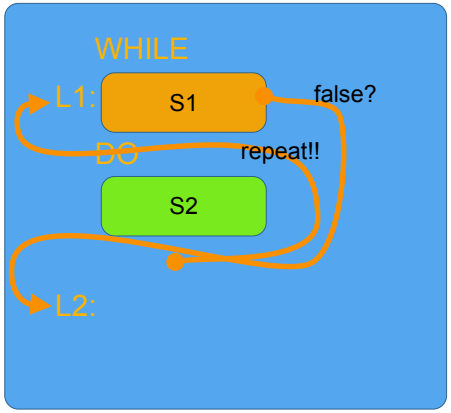
descarta o valor a cada iteração...!

No fim deixa um valor na pilha...

Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

$$comp : microML \times ENV \rightarrow CodeSeq$$




```
L1:nop
   [[ S1 - condition ]]
   brfalse L2:
   [[ S2 - body ]]
   pop
   br L1
L2:ldc.i4 0
   box int32
```


Compilador de CALCI

```
decl
  x = newvar(2)
in
  x := !x + 1
end
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 2
box int32
newobj instance void Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 0
ldelem.i4
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
ldc.i4 1
box int32
unbox int32
ldobj int32
add
box int32
callvirt instance object Cell::set(object)
```


Compilador de CALCI

```
decl
  x = newvar(2)
in
  x := !x + 1
end
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 2
box int32
newobj instance void Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 0
ldelem.i4
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
ldc.i4 1
box int32
unbox int32
ldobj int32
add
box int32
callvirt instance object Cell::set(object)
```

Compilador de CALCI

```
decl
  x = newvar(2)
in
  x := !x + 1
end
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 2
box int32
newobj instance void Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 0
ldelem.i4
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
ldc.i4 1
box int32
unbox int32
ldobj int32
add
box int32
callvirt instance object Cell::set(object)
```


Compilador de CALCI

```
decl
  x = newvar(2)
in
  x := !x + 1
end
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 2
box int32
newobj instance void Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 0
ldelem.i4
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
ldc.i4 1
box int32
unbox int32
ldobj int32
add
box int32
callvirt instance object Cell::set(object)
```


Compilador de CALCI

```
decl
  x = newvar(2)
in
  x := !x + 1
end
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 2
box int32
newobj instance void Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 0
ldelem.i4
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
ldc.i4 1
box int32
unbox int32
ldobj int32
add
box int32
callvirt instance object Cell::set(object)
```


Compilador de CALCI

```
decl
  x = newvar(2)
in
  x := !x + 1
end
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 2
box int32
newobj instance void Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 0
ldelem.i4
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
ldc.i4 1
box int32
unbox int32
ldobj int32
add
box int32
callvirt instance object Cell::set(object)
```

Compilador de CALCI

```
decl
  x = newvar(2)
in
  x := !x + 1
end
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 2
box int32
newobj instance void Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 0
ldelem.i4
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
ldc.i4 1
box int32
unbox int32
ldobj int32
add
box int32
callvirt instance object Cell::set(object)
```


Compilador de CALCI

```
decl
  x = newvar(2)
in
  x := !x + 1
end
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 2
box int32
newobj instance void Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 0
ldelem.i4
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
ldc.i4 1
box int32
unbox int32
ldobj int32
add
box int32
callvirt instance object Cell::set(object)
```


Compilador de CALCI

```
decl s = newvar(0) in
decl b = newvar(100) in
  while ( b > 0 ) do
    s := !s + !b;
    b := !b - 1;
  end;
!s
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 0
box int32
newobj ... Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 1
ldc.i4 100
box int32
newobj instance void Cell::.ctor
(object)
stelem.i4
...
```


Compilador de CALCI

```
decl s = newvar(0) in
decl b = newvar(100) in
  while ( b > 0 ) do
    s := !s + !b;
    b := !b - 1;
  end;
!s
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 0
box int32
newobj ... Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 1
ldc.i4 100
box int32
newobj instance void Cell::.ctor
(object)
stelem.i4
...
```

Compilador de CALCI

```
decl s = newvar(0) in
decl b = newvar(100) in
  while ( b > 0 ) do
    s := !s + !b;
    b := !b - 1;
  end;
!s
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 0
box int32
newobj ... Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 1
ldc.i4 100
box int32
newobj instance void Cell::.ctor
(object)
stelem.i4
...
```


Compilador de CALCI

```
decl s = newvar(0) in
decl b = newvar(100) in
  while ( b > 0 ) do
    s := !s + !b;
    b := !b - 1;
  end;
!s
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
ldloc table
ldc.i4 0
ldc.i4 0
box int32
newobj ... Cell::.ctor(object)
stelem.i4
ldloc table
ldc.i4 1
ldc.i4 100
box int32
newobj instance void .ctor(object)
stelem.i4
...
```


Compilador de CALCI

```
decl s = newvar(0) in
decl b = newvar(100) in
  while ( b > 0 ) do
    s := !s + !b;
    b := !b - 1;
  end;
!s
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
...
L0:nop
ldloc table
ldc.i4 1
ldelem.i4
unbox int32
ldobj int32
ldc.i4 0
box int32
unbox int32
ldobj int32
cgt
box bool
unbox bool
ldobj bool
brfalse L1
...
br L0
L1:nop
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
```


Compilador de CALCI

```
decl s = newvar(0) in
decl b = newvar(100) in
  while ( b > 0 ) do
    s := !s + !b;
    b := !b - 1;
  end;
!s
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
...
L0:nop
ldloc table
ldc.i4 1
ldelem.i4
unbox int32
ldobj int32
ldc.i4 0
box int32
unbox int32
ldobj int32
cgt
box bool
unbox bool
ldobj bool
brfalse L1
...
br L0
L1:nop
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
```

Compilador de CALCI

```
decl s = newvar(0) in
decl b = newvar(100) in
  while ( b > 0 ) do
    s := !s + !b;
    b := !b - 1;
  end;
!s
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
...
L0:nop
ldloc table
ldc.i4 1
ldelem.i4
unbox int32
ldobj int32
ldc.i4 0
box int32
unbox int32
ldobj int32
cgt
box bool
unbox bool
ldobj bool
brfalse L1
...
br L0
L1:nop
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
```


Compilador de CALCI

```
decl s = newvar(0) in
decl b = newvar(100) in
  while ( b > 0 ) do
    s := !s + !b;
    b := !b - 1;
  end;
!s
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
...
ldloc table
ldc.i4 0
ldelem.i4
ldloc table
ldc.i4 0
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
ldloc table
ldc.i4 1
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
add
box int32
callvirt instance object..set(object)
...
```


Compilador de CALCI

```
decl s = newvar(0) in
decl b = newvar(100) in
  while ( b > 0 ) do
    s := !s + !b;
    b := !b - 1;
  end;
!s
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
...
ldloc table
ldc.i4 1
ldelem.i4
ldloc table
ldc.i4 1
ldelem.i4
callvirt instance object Cell::get()
unbox int32
ldobj int32
ldc.i4 1
box int32
unbox int32
ldobj int32
sub
box int32
callvirt instance object..set(object)
...
```


Compilador de CALCI

```
decl s = newvar(0) in
decl b = newvar(100) in
  while ( b > 0 ) do
    s := !s + !b;
    b := !b - 1;
  end;
!s
```

```
.locals init (int32[] table)
ldc.i4 2
newarr int32
stloc table
...
L0:nop

...

unbox bool
ldobj bool
brfalse L1

...

br L0
L1:nop
...
```

Resumindo... ideias a reter...

- As linguagens imperativas têm como principal característica a produção de efeitos no estado da memória.
- Variáveis e identificadores são conceitos diferentes mas muitas vezes apresentados em associação.
  - A ligação entre identificadores e denotações é imutável
  - constantes - a denotação é o valor associado
  - variáveis C ou Java - a denotação é uma localização de memória
- O conteúdo da memória
  - é dividido em células com referências distintas (**var**)
  - é acessível um valor do tipo referência (**deref**)
  - é mutável por operações sobre referências (**assign**)
  - é reutilizável mediante análise do seu tempo de vida (**free** e garbage collection)

Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2010-2011

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Unidade 6: Sistemas de tipos

Os sistemas de tipos são ferramentas de análise estática que garantem boas propriedades de programas concretos. A análise estática é uma forma de interpretação "abstracta" de programas. A principal característica da análise estática é que termina sempre, mesmo para programas que não terminam.

A forma mais comum de análise estática é a verificação de tipos (type checking). Os sistemas de tipos garantem a ausência de certos tipos de erros durante a execução.

- Erros de execução
- Interpretação abstracta
- Sistemas de tipos
- Consistência de um sistema de tipos
- Detecção de erros de execução

Erros de execução...

- O que é que pode correr mal na execução de um programa?

```
decl
  a = newvar(0)
  b = newvar(2)
  c = newvar(a > b)
in
  if c then
    !a := !a + 1;
     c := 1 < !c
end
```


Erros de execução...

- O que é que pode correr mal na execução de um programa?

```
decl
  a = newvar(0)
  b = newvar(2)
  c = newvar(!a > !b)
in
  if c then
     a := !a + 1;
     c := 1 < !a
end
```

Erros de execução...

- O que é que pode correr mal na execução de um programa?

```
eval( add(E1, E2) , env , m0) ≜ [ (v1 , m1) = eval( E1, env, m0);
                                  (v2 , m2) = eval( E2, env, m1);
                                  (v1 + v2 , m2) ]
eval( deref(E) , env , m0) ≜ [ (ref , m1) = eval( E, env, m0);
                               (m1.get(ref) , m1) ]
eval( assign(E1, E2) , env , m0) ≜ [(v1 , m1) = eval( E1, env, m0);
                                    (v2 , m2) = eval( E2, env, m1);
                                    m3 = m2.set(v1, v2) ;
                                    (v1 , m3) ]
```

É impossível em tempo de compilação calcular todos os valores possíveis e assim evitar os erros de execução. Os domínios são infinitos...

Podemos interpretar os programas aglomerando conjuntos infinitos de valores num único representante, um "tipo".

Funções Semânticas

- Algoritmo **eval** para calcular o valor de uma expressão qualquer da linguagem microML:

$$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Algoritmo **comp** para gerar um programa na linguagem CIL equivalente a uma dada expressão da linguagem microML.

$$\text{comp} : \text{microML} \times \text{ENV} \rightarrow \text{CodeSeq}$$

Funções Semânticas (mais uma)

- Algoritmo **eval** para calcular o valor de uma expressão qualquer da linguagem microML:

$$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Algoritmo **comp** para gerar um programa na linguagem CIL equivalente a uma dada expressão da linguagem microML.

$$\text{comp} : \text{microML} \times \text{ENV} \rightarrow \text{CodeSeq}$$

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$\text{typechk} : \text{microML} \times \text{ENV} \rightarrow \text{TYPE}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\}, \text{none} \}$$

Tipos para microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$\text{typechk} : \text{microML} \times \text{ENV} \rightarrow \text{TYPE}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\}, \text{none} \}$$

- A função **typchk** pode ser vista como um interpretador que avalia um programa de acordo com uma semântica especial, mais abstracta, (em que os "valores" são "tipos").
- Nesta semântica, as operações da linguagem estão definidas com tipos como operandos e tipos como resultado.

Tipos para microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$typechk : microML \times ENV \rightarrow TYPE$$

$$ENV : ID \rightarrow TYPE$$

$$TYPE = \{ int, bool, ref\{TYPE\}, none \}$$

int: é o tipo dos valores inteiros.

bool: é o tipo dos valores booleanos.

ref{$\mathcal{T}$}:é o tipo das referências para células que só podem conter valores de tipo $\mathcal{T}$.

Exemplo: ref{ref{int}} é o tipo das referências para células que só podem conter referências para (células) com valores inteiros.

none: é o "tipo" dos programas para os quais as funções semânticas

Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$typechk : microML \times ENV \rightarrow TYPE$$

Se **E** for da forma **add**(E1, E2) e

se E1 for do tipo int num dado ambiente *env* e

se E2 for do tipo int num dado ambiente *env*

então o tipo de **E** é int

e se uma destas condições falhar?
então a execução da expressão não estaria definida...

senão, o tipo de **E** é none

Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$typechk : microML \times ENV \rightarrow TYPE$$

typchk(**add**(E1, E2) , *env*) ≜ [*t1* = **typchk** (E1, *env*)
t2 = **typchk** (E2, *env*)
if (*t1* == int) **and** (*t2* == int)
then int
else none]

Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$typchk : microML \times ENV \rightarrow TYPE$$

Todas as expressões num(n) denotam valores inteiros... o representante dos inteiros é int

typchk(**num**(*n*) , *env*) ≜ int ;
typchk(**id**(s) , *env*) ≜ *env*.Find(s) ;
typchk(**true** , *env*) ≜ bool ;
typchk(**false** , *env*) ≜ bool ;

Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$typechk : microML \times ENV \rightarrow TYPE$$

```
typchk( and(E1, E2) , env ) ≜ [ t1 =  typchk ( E1, env)
                                t2 =  typchk ( E2, env)
                                if  (t1 == bool ) and (t2 == bool )
                                    then bool
                                    else none ]
```


Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$typechk : microML \times ENV \rightarrow TYPE$$

```
typchk( assign(E1, E2) , env ) ≜
                        [ t1 = typchk( E1, env );
                          t2 = typchk( E2, env );
                          if (t1 == ref{t2})
                                then t2;
                                else none ; ]
```


Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$typechk : microML \times ENV \rightarrow TYPE$$

```
typchk( if(E1, E2, E3) , env ) ≜
                [ t1 = typchk( E1, env );
                  if ( t1 != bool ) then none
                  else [ t2 = typchk( E2, env );
                         t3 = typchk( E3, env );
                         if (t2 == none ) or (t3 == none ) or (t2 != t3)
                         then none
                         else t2 ] ]
```


Tipificação de microML

- Compare com o caso **if** do algoritmo **eval**.

  Os "ramos" E2 e E3 são ambos analisados.
  Impõe-se (como restrição) *t2* == *t3* . [Porquê?]

```
typchk( if(E1, E2, E3) , env ) ≜
                [ t1 = typchk( E1, env );
                  if ( t1 != bool ) then none
                  else [ t2 = typchk( E2, env );
                         t3 = typchk( E3, env );
                         if (t2 == none ) or (t3 == none ) or (t2 != t3)
                         then none
                         else t2 ] ]
```

Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$\text{typechk} : \text{microML} \times \text{ENV} \rightarrow \text{TYPE}$$

```
typchk( while(E1, E2) , env ) ≜
                [ t1 = typchk( E1, env );
                  if ( t1 != bool) then none
                  else [ t2 = typchk( E2, env );
                         if (t2 == none ) then none
                         else bool ] ]
```


Tipificação de microML

- Compare com o caso **while** do algoritmo **eval**.

  O "ramo" E2 é sempre analisado exactamente uma vez. Impõe-se como tipo o tipo bool. [Porquê?]

```
typchk( while(E1, E2) , env ) ≜
                [ t1 = typchk( E1, env );
                  if ( t1 != bool) then none
                  else [ t2 = typchk( E2, env );
                         if (t2 == none ) then none
                         else bool ] ]
```


Tipificação de microML

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem microML:

$$\text{typechk} : \text{microML} \times \text{ENV} \rightarrow \text{TYPE}$$

```
typchk( decl(s, E1, E2) , env ) ≜
                [ t1 = typchk( E1, env);
                  envlocal = env.BeginScope();
                  envlocal.Assoc(s, t1);
                  t2 = typchk(E2, envlocal);
                  env = envlocal.EndScope(); t2 ]
```


Correcção da Tipificação

- Podemos verificar que, para todo o programa microML P, e ambiente *env* que cubra todos os identificadores livres de P, a operação

  typchk(P, *env*)

  está bem definida e termina sempre [Porquê?]

- Podemos também demonstrar o seguinte teorema, que relaciona a tipificação com a avaliação de programas microML.[Como?]

  Teorema: Para todo o programa microML P e tipo $\mathcal{T}$,
  Se **typchk**(P,∅) = $\mathcal{T}$ e **eval**(P, ∅, ∅) = v então v ∈ $\mathcal{T}$.

Correcção da Tipificação

Teorema: Para todo o programa **micro**ML P e tipo τ,
Se **typchk**(P,∅) = τ e **eval**(P, ∅, ∅) = v então v ∈ τ.

Em particular, se **typchk**(P,∅) ≠ none então **eval**(P, ∅, ∅) ≠ error.
Ou seja:
ou P não termina,
ou P termina e não incorre em erros de execução.

"*Well-typed programs don´t go wrong*" (Robin Milner)

Correcção da Tipificação

Teorema: Para todo o programa **micro**ML P e tipo τ,
Se **typchk**(P,∅) = τ e **eval**(P, ∅, ∅) = v então v ∈ τ.

Esta propriedade é em geral conhecida como a "consistência do sistema de tipos". Garante que

"*Well-typed programs don´t go wrong*" (Robin Milner)

Existem muitos programas P tais que eval(P, ∅, ∅) = v e v ∈ int mas (infelizmente) typchk(P, ∅) = error ! [Exemplo?]

Robin Milner

ACM Turing Award (1991)

For three distinct and complete achievements:

1) LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2) ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3) CCS, a general theory of concurrency. In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

1. O programa está bem tipificado? se não, apresente uma versão correcta.
2. Qual o ambiente de tipificação da expressão `!y` ?
3. Qual o resultado/efeito do programa?

```
decl
      x = 10
      y = var(0)
in
  decl
      z = var(y)
      w = var(false)
  in
      while w do
        w := ((!z := !!z + y + 1) < x)
      end; !y
  end
end
```

Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2010-2011

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Unidade 7: Abstracção funcional

Nas linguagens de programação existem várias formas de tornar código mais abstracto e reutilizável em diferentes contextos. A abstracção tem por objectivo aumentar o nível de detalhe com que se olha para um problema e pode ser conseguida ao nível da funcionalidade, dos dados, da iteração de colecções, através da hierarquização da informação, etc.
Todas as linguagens de programação modernas possuem uma ou mais destas formas de abstracção, e normalmente todas incluem suporte primitivo para a abstracção funcional por parametrização.

Abstracção por parametrização
Equivalência alfa
A linguagem CALCF
Semântica com substituição da linguagem CALCF
Algoritmo interpretador da linguagem CALCF com ambiente
Estratégias de resolução de nomes
Algoritmo compilador da linguagem CALCF
Declarações recursivas
Passagem de parâmetros (por valor, por nome)

Abstracção por Parametrização

- As linguagens de programação têm um número finito de construções que podem ser usadas para representar um número infinito de computações.
- Podemos capturar grupos de computações semelhantes através da abstracção de alguns das suas componentes (as que variam).
- A abstracção é uma forma de enriquecer as linguagens de programação com novas operações, com um nível mais alto (manipulam entidades mais complexas como um todo) definidas a partir de outras construções da linguagem.

```
1*1 + 2*2
1*1 + 1*2
2*2 + 1*1
1*2 + 2*1
x*x + y*y
2*2 + 3*2
5*5 + 7*7
2*2 + 9*9
1*1 + 2*2
3*1 + 1*2
```


Abstracção por Parametrização

- É a possibilidade de definir entidades genéricas, introduzidas uma única vez e utilizáveis várias vezes num programa com diferentes instanciações de um conjunto de parâmetros definidos.
- É uma característica fundamental de todas as linguagens de programação modernas:
  - Funções / Procedimentos
  - Métodos
  - Tipos paramétricos
  - Classes paramétricas
  - etc...
- A parametrização e a abstracção são essenciais à modularidade dos programas, e em alguns casos, à expressividade computacional das linguagens.

Parametrização

- É o mecanismo geral para declarar os nomes que se querem encarar como genéricos, destacando-os sintacticamente através de notação conveniente:

  C: `int f(int x) { return x+1;}`

  ML: `(fun x -> x+1)`

  Lisp: `(lambda (x) (add x 1))`

  Cálculo Lambda (Church): $\lambda x.x$

- Tecnicamente, chama-se abstracção a uma construção sintáctica, explicitamente parametrizada num certo conjunto de nomes.

Abstracção

- Uma abstracção é uma construção sintáctica constituída por dois elementos fundamentais:
  - As declarações dos seus parâmetros
  - O corpo da abstracção (uma qualquer construção da linguagem)

  `(x → x+y)`

  Parâmetros declarados: `x`; Corpo: `x+y`

  `(y → (x → x+y))`

  Parâmetros declarados: `y`; Corpo: `(x → x+y)`

  `(y,x → x+y)`

  Parâmetros declarados: `x,y`; Corpo: `x+y`

- Uma abstracção representa uma função anónima dos seus parâmetros...

Observações

- As declarações dos parâmetros de uma abstracção são ocorrências ligantes dos nomes respectivos.
- O âmbito das declarações dos parâmetros é (exactamente) o corpo da abstracção.
- Os nomes livres duma abstracção são todos os nomes livres no seu corpo que não são parâmetros

$NomesLivres((x_1,\ldots,x_n \rightarrow E) = NomesLivres(E) \setminus \{x_1,\ldots,x_n\}$

exemplo:

`NomesLivres((x → x+y)) = NomesLivres(x+y)\{x} = {y}`

Observações

- As declarações dos parâmetros de uma abstracção são ocorrências ligantes dos nomes respectivos.
- O âmbito das declarações dos parâmetros é (exactamente) o corpo da abstracção.
- Os nomes livres duma abstracção são todos os nomes livres no seu corpo que não são parâmetros

```
NomesLivres(x) = {x}
NomesLivres(E1 op E2) = NomesLivres(E1) U NomesLivres(E2)
NomesLivres(decl x = E1 in E2) = NomesLivres(E1) U NomesLivres(E2)\{x}
```

exemplos:

```
NomesLivres(x+decl y = x in x+y end) = {x}
NomesLivres(decl x = 1 in decl y = z in x+y end end) = {z}
```

Equivalência-alfa ($=_\alpha$)

- Duas abstracções dizem-se alfa-equivalentes se uma pode ser obtida a partir da outra através da renomeação dos seus parâmetros, evitando conflitos com os seus nomes livres:

  `(x → x+y)` $=_\alpha$ `(z → z+y)`

  `(x → x+y)` $\neq_\alpha$ `(y → y+y)`

- Abstracções alfa-equivalentes são semanticamente equivalentes (são interpretadas como denotando a mesma função):

  `(x → x+1)` $=_\alpha$ `(z → z+1)`

- Por isso, um interpretador pode traduzir os nomes dos parâmetros em algo mais conveniente e conhecido apenas localmente...

O que denota uma abstracção?

- Uma abstracção é uma expressão sintáctica que denota uma certa função.
- Uma função é uma entidade semântica primitiva que suporta uma operação de aplicação, tal como um valor inteiro é uma entidade semântica primitiva que suporta a operação de adição, etc.
- Uma linguagem pode suportar funções mas não abstracções (exemplo: C, C++, Pascal)
- Várias linguagens suportam abstracções (ML, Smalltalk, Python, Javascript, Java (usando objectos anónimos))

Abstracções vs. Declarações

- Em C e Pascal, as expressões que representam funções aparecem sempre no contexto de uma declaração:
  - `int f(int x) { return x+1; } (`*`resto do programa`*`)`
- Em ML, o mecanismo de declaração está separado do mecanismo de abstracção:
  - `let x=2 in (resto do programa)`
  - `let f = (fun x->x+1) in (resto do programa)`
  - `(map (fun x-> x+1) [1;2;3]) = [2;3;4]`
- Em ML, as funções são "cidadãs de primeira classe" (first-class citizens) ou seja, são tratadas como qualquer outro valor da linguagem. O mesmo não se passa com as funções em Pascal ou C.

Abstracções (exemplos)

- Smalltalk
  - [:x | E] representa um bloco de programa, e é uma abstracção
  - 1 to: 10 do: [:i | s ← s+i] é a forma de fazer um ciclo...
- Abstracções em linguagens de objectos
  - Uma abstracção pode ser representada por um objecto que implementa um método "apply" (Function object)
  - Exemplo em Java...

```
Interface Function { int apply(int x); }
...
Function f = new Function{
       int apply(int x) {
                return x+1;
       }
} // f  = λx. x+1
...
int v = f.apply(2) // v = (f 2)
```

Abstracções (exemplos)

- As abstracções podem realizar-se não apenas sobre identificadores de valores, mas também sobre outras entidades, como por exemplo tipos.
- C++

```
template <class T> class Stack { int push(const &T); … }
```

- Java (5.0)

```
interface List<E> { void add(E x); Iterator<E> iterator(); }

interface Iterator<E> { E next(); boolean hasNext(); }

class LinkedList<E> implements List<E> { … }
```


Exemplo: A Linguagem CALCF

- A linguagem CALCF estende a linguagem CALCI com funções, representadas por abstracções:

```
fun Id -> Expression end
```

- As funções podem ser aplicadas ao seu argumento, usando a expressão de aplicação:

```
Expression1 ( Expression2 )
```

- Semântica pretendida:
  Se ***Expression1*** denota uma função f, e ***Expression2*** denota um valor qualquer v, então ***Expression1*(Expression2)*** denota o valor que resulta de aplicar f a v.

Exemplo: A Linguagem CALCF

- Um programa simples:

```
fun x -> x+2 end (4)
```

- Um outro exemplo:

```
decl f= fun x -> x+1 end in
  decl g = fun y -> f(y)+2 end in
    decl x = g(2)
      in x+x
```

- O mesmo exemplo numa sintaxe concreta tipo C:

```
function f(x){ return x+1;}
function g(y){ f(y)+2 }
{ ... x = g(2); return x+x; }
```


A Linguagem CALCF

- Tipo de dados CALCF com os constructores:

```
num:   Integer → CALCF
add:   CALCF × CALCF → CALCF
mul:   CALCF × CALCF → CALCF
div:   CALCF × CALCF → CALCF
sub:   CALCF × CALCF → CALCF
id:    String → CALCF
decl:  String × CALCF × CALCF → CALCF
fun:   String × CALCF → CALCF
call:  CALCF × CALCF → CALCF
```

Semântica de CALCF (1)

A função semântica I de CALCF:

$$I : CALCF \rightarrow RESULT$$

CALCF = conjunto dos programas fechados

RESULT = conjunto dos significados (denotações)

- Um significado pode ser um valor inteiro ou uma função (representada por uma abstracção):

$$RESULT = Integer \cup Abstraction \cup \{ error \}$$

Resultados de CALCF

- Os significados de programas da linguagem CALCF podem ser apresentados como um tipo indutivo
- Tipo de dados RESULT com os constructores num e abstraction

```
num:                    Integer → RESULT
abstraction:  String × CALCF → RESULT
error:                  void → RESULT
```


Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

$$eval : CALCF \rightarrow RESULT$$

```
eval( num(n) )                ≜ n
eval( add(E1,E2) )        ≜ eval(E1) + eval(E2)
   …
eval( decl(s, E1, E2))    ≜ eval(subst(s, E2, E1))
```


Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

$$eval : CALCF \rightarrow RESULT$$

```
eval( num(n) )                ≜ n
eval( add(E1,E2) )        ≜ eval(E1) + eval(E2)
   …
eval( decl(s, E1, E2))    ≜ eval(subst(s, E2, eval(E1)))
```

calcular o valor primeiro
substitui depois...

Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

eval : CALCF → RESULT

```
eval( num(n) )                ≜ n
eval( add(E1,E2) )            ≜ eval(E1) + eval(E2)
      …
eval( decl(s, E1, E2))        ≜ eval(subst(s, E2, eval(E1)))
eval( fun(s, E) )             ≜ abstraction(s, E)
eval( call(E1, E2) )          ≜ [arg = eval(E2); fun = eval(E1);
                          if fun is abstraction(param, body)
                          then eval( subst(param, body, arg) )
                          else error ]
```


Exemplos

```
subst(x, x, 2*y ) = 2*y

subst(x, 2*4+(x+2), 2*y ) = 2*4+(2*y+2)

subst(x, decl x = 1 in x+y end, 2*y ) = decl x = 1 in x+y end

subst(x, decl z = 1 in x+z end, 2*y ) = decl z = 1 in 2*y+z end

subst(x, decl y = 1 in x+y end, 2*y ) = decl y = 1 in 2*y+y end
```

A ligação da ocorrência do identificador y que foi inserido na expressão à sua declaração foi "capturada"!!

Definição da Função Subst

```
subst(s, num(n), F )              ≜ num(n);
subst(s, id(s), F )         ≜ F;
subst(s, add(E1,E2),F )     ≜ add( subst(s, E1, F), subst
(s,E2,F));
…

subst(s, decl(s, E1, E2), F) ≜ [ /* caso s = s' */
                           G = subst(s, E1, F);
                           decl(s, G, E2);   ]

subst(s, decl(s', E1, E2), F) ≜ [ /* caso s ≠ s' */
                            G = subst(s, E1, F);
                            newid = fresh_identifier();
                            E2' = subst(s', E2, newid);
                            decl(newid, G, subst(s, E2', F)); ]
```


Definição da Função Subst

```
subst(s, num(n), F )              ≜ num(n);
subst(s, id(s), F )         ≜ F;
subst(s, add(E1,E2),F )     ≜ add( subst(s, E1, F), subst
(s,E2,F));
…

subst(s, decl(s, E1, E2), F) ≜ [ /* caso s = s' */
                           G = subst(s, E1, F);
                           decl(s, G, E2);   ]

subst(s, decl(s', E1, E2), F) ≜ [ /* caso s ≠ s' */
                            G = subst(s, E1, F);
                            newid = fresh_identifier();
                            E2' = subst(s', E2, newid);
                            decl(newid, G, subst(s, E2', F)); ]
```

A renomeação do identificador local evita a captura ilegal de ocorrências livres do identificador s' em F

Definição da Função Subst

```
subst(s, call(E1, E2) )    ≜      call( subst(s, E1,F), subst(s,
E2,F))

subst(s, fun(s', E), F) ≜ /* caso s = s' */
                          fun (s, E)

subst(s, fun(s', E), F) ≜ [/* caso s ≠ s' */
                          newid = fresh_identifier();
                          E' = subst(s', E, newid);
                          fun(newid, subst(s, E', F))]
```


Definição da Função Subst

```
subst(s, call(E1, E2) )    ≜      call( subst(s, E1,F), subst(s,
E2,F))

subst(s, fun(s', E), F) ≜ /* caso s = s' */
                          fun (s, E)

subst(s, fun(s', E), F) ≜ [/* caso s ≠ s' */
                          newid = fresh_identifier();
                          E' = subst(s', E, newid);
                          fun(newid, subst(s, E', F))]
```

A renomeação do identificador local evita a captura ilegal de ocorrências livres do identificador s' em F

Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

```
(5)
```


Quiz

- Quais os passos da avaliação do programa P segundo a semântica com substituição?
- P:

```
decl y = 3 in
  decl x = 2*y in
    decl f = (fun y -> y+x) in
      decl g = (fun x -> (fun h -> x+h(x))
        in (g(2))(f)
```

```
eval( num(n) )              ≜ n
eval( add(E1,E2) )          ≜ eval(E1) + eval(E2)
      …
eval( decl(s, E1, E2))      ≜ eval(subst(s, E2, eval(E1)))
eval( fun(s, E) )           ≜ abstraction(s, E)
eval( call(E1, E2) )        ≜ [arg = eval(E2); fun = eval(E1);
                          if fun is abstraction(param, body)
                          then eval( subst(param, body, arg) )
                          else error ]
```

Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2010-2011

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Unidade 7: Abstracção funcional

Nas linguagens de programação existem várias formas de tornar código mais abstracto e reutilizável em diferentes contextos. A abstracção tem por objectivo aumentar o nível de detalhe com que se olha para um problema e pode ser conseguida ao nível da funcionalidade, dos dados, da iteração de colecções, através da hierarquização da informação, etc.
Todas as linguagens de programação modernas possuem uma ou mais destas formas de abstracção, e normalmente todas incluem suporte primitivo para a abstracção funcional por parametrização.

Abstracção por parametrização
Equivalência alfa
A linguagem CALCF
Semântica com substituição da linguagem CALCF
Algoritmo interpretador da linguagem CALCF com ambiente
Estratégias de resolução de nomes
Algoritmo compilador da linguagem CALCF
Declarações recursivas
Passagem de parâmetros (por valor, por nome)

Abstracção por Parametrização

- As linguagens de programação têm um número finito de construções que podem ser usadas para representar um número infinito de computações.
- Podemos capturar grupos de computações semelhantes através da abstracção de alguns das suas componentes (as que variam).
- A abstracção é uma forma de enriquecer as linguagens de programação com novas operações, com um nível mais alto (manipulam entidades mais complexas como um todo) definidas a partir de outras construções da linguagem.

```
1*1 + 2*2
1*1 + 1*2
2*2 + 1*1
1*2 + 2*1
x*x + y*y
2*2 + 3*2
5*5 + 7*7
2*2 + 9*9
1*1 + 2*2
3*1 + 1*2
```


Abstracção por Parametrização

- É a possibilidade de definir entidades genéricas, introduzidas uma única vez e utilizáveis várias vezes num programa com diferentes instanciações de um conjunto de parâmetros definidos.
- É uma característica fundamental de todas as linguagens de programação modernas:
  - Funções / Procedimentos
  - Métodos
  - Tipos paramétricos
  - Classes paramétricas
  - etc...
- A parametrização e a abstracção são essenciais à modularidade dos programas, e em alguns casos, à expressividade computacional das linguagens.

Parametrização

- É o mecanismo geral para declarar os nomes que se querem encarar como genéricos, destacando-os sintacticamente através de notação conveniente:

  C: `int f(int x) { return x+1;}`

  ML: `(fun x -> x+1)`

  Lisp: `(lambda (x) (add x 1))`

  Cálculo Lambda (Church): $\lambda x.x$

- Tecnicamente, chama-se abstracção a uma construção sintáctica, explicitamente parametrizada num certo conjunto de nomes.

Abstracção

- Uma abstracção é uma construção sintáctica constituída por dois elementos fundamentais:
  - As declarações dos seus parâmetros
  - O corpo da abstracção (uma qualquer construção da linguagem)

  `(x → x+y)`

  Parâmetros declarados: `x`; Corpo: `x+y`

  `(y → (x → x+y))`

  Parâmetros declarados: `y`; Corpo: `(x → x+y)`

  `(y,x → x+y)`

  Parâmetros declarados: `x,y`; Corpo: `x+y`

- Uma abstracção representa uma função anónima dos seus parâmetros...

Observações

- As declarações dos parâmetros de uma abstracção são ocorrências ligantes dos nomes respectivos.
- O âmbito das declarações dos parâmetros é (exactamente) o corpo da abstracção.
- Os nomes livres duma abstracção são todos os nomes livres no seu corpo que não são parâmetros

$NomesLivres((x_1,\ldots,x_n \rightarrow E) = NomesLivres(E) \setminus \{x_1,\ldots,x_n\}$

exemplo:

`NomesLivres((x → x+y)) = NomesLivres(x+y)\{x} = {y}`

Observações

- As declarações dos parâmetros de uma abstracção são ocorrências ligantes dos nomes respectivos.
- O âmbito das declarações dos parâmetros é (exactamente) o corpo da abstracção.
- Os nomes livres duma abstracção são todos os nomes livres no seu corpo que não são parâmetros

```
NomesLivres(x) = {x}
NomesLivres(E1 op E2) = NomesLivres(E1) U NomesLivres(E2)
NomesLivres(decl x = E1 in E2) = NomesLivres(E1) U NomesLivres(E2)\{x}
```

exemplos:

```
NomesLivres(x+decl y = x in x+y end) = {x}
NomesLivres(decl x = 1 in decl y = z in x+y end end) = {z}
```

Equivalência-alfa ($=_\alpha$)

- Duas abstracções dizem-se alfa-equivalentes se uma pode ser obtida a partir da outra através da renomeação dos seus parâmetros, evitando conflitos com os seus nomes livres:

  `(x → x+y)` $=_\alpha$ `(z → z+y)`

  `(x → x+y)` $\neq_\alpha$ `(y → y+y)`

- Abstracções alfa-equivalentes são semanticamente equivalentes (são interpretadas como denotando a mesma função):

  `(x → x+1)` $=_\alpha$ `(z → z+1)`

- Por isso, um interpretador pode traduzir os nomes dos parâmetros em algo mais conveniente e conhecido apenas localmente...

O que denota uma abstracção?

- Uma abstracção é uma expressão sintáctica que denota uma certa função.
- Uma função é uma entidade semântica primitiva que suporta uma operação de aplicação, tal como um valor inteiro é uma entidade semântica primitiva que suporta a operação de adição, etc.
- Uma linguagem pode suportar funções mas não abstracções (exemplo: C, C++, Pascal)
- Várias linguagens suportam abstracções (ML, Smalltalk, Python, Javascript, Java (usando objectos anónimos))

Abstracções vs. Declarações

- Em C e Pascal, as expressões que representam funções aparecem sempre no contexto de uma declaração:
  - `int f(int x) { return x+1; } (`*`resto do programa`*`)`
- Em ML, o mecanismo de declaração está separado do mecanismo de abstracção:
  - `let x=2 in (resto do programa)`
  - `let f = (fun x->x+1) in (resto do programa)`
  - `(map (fun x-> x+1) [1;2;3]) = [2;3;4]`
- Em ML, as funções são "cidadãs de primeira classe" (first-class citizens) ou seja, são tratadas como qualquer outro valor da linguagem. O mesmo não se passa com as funções em Pascal ou C.

Abstracções (exemplos)

- Smalltalk
  - [:x | E] representa um bloco de programa, e é uma abstracção
  - 1 to: 10 do: [:i | s ← s+i] é a forma de fazer um ciclo...
- Abstracções em linguagens de objectos
  - Uma abstracção pode ser representada por um objecto que implementa um método "apply" (Function object)
  - Exemplo em Java...

```
Interface Function { int apply(int x); }
...
Function f = new Function{
       int apply(int x) {
                return x+1;
       }
} // f  = λx. x+1
...
int v = f.apply(2) // v = (f 2)
```

Abstracções (exemplos)

- As abstracções podem realizar-se não apenas sobre identificadores de valores, mas também sobre outras entidades, como por exemplo tipos.
- C++

```
template <class T> class Stack { int push(const &T); … }
```

- Java (5.0)

```
interface List<E> { void add(E x); Iterator<E> iterator(); }

interface Iterator<E> { E next(); boolean hasNext(); }

class LinkedList<E> implements List<E> { … }
```


Exemplo: A Linguagem CALCF

- A linguagem CALCF estende a linguagem CALCI com funções, representadas por abstracções:

```
fun Id -> Expression end
```

- As funções podem ser aplicadas ao seu argumento, usando a expressão de aplicação:

```
Expression1 ( Expression2 )
```

- Semântica pretendida:
  Se ***Expression1*** denota uma função f, e ***Expression2*** denota um valor qualquer v, então ***Expression1*(Expression2)*** denota o valor que resulta de aplicar f a v.

Exemplo: A Linguagem CALCF

- Um programa simples:

```
fun x -> x+2 end (4)
```

- Um outro exemplo:

```
decl f= fun x -> x+1 end in
  decl g = fun y -> f(y)+2 end in
    decl x = g(2)
      in x+x
```

- O mesmo exemplo numa sintaxe concreta tipo C:

```
function f(x){ return x+1;}
function g(y){ f(y)+2 }
{ ... x = g(2); return x+x; }
```


A Linguagem CALCF

- Tipo de dados CALCF com os constructores:

```
num:   Integer → CALCF
add:   CALCF × CALCF → CALCF
mul:   CALCF × CALCF → CALCF
div:   CALCF × CALCF → CALCF
sub:   CALCF × CALCF → CALCF
id:    String → CALCF
decl:  String × CALCF × CALCF → CALCF
fun:   String × CALCF → CALCF
call:  CALCF × CALCF → CALCF
```

Semântica de CALCF (1)

A função semântica I de CALCF:

$$I : CALCF \rightarrow RESULT$$

CALCF = conjunto dos programas fechados

RESULT = conjunto dos significados (denotações)

- Um significado pode ser um valor inteiro ou uma função (representada por uma abstracção):

$$RESULT = Integer \cup Abstraction \cup \{ error \}$$

Resultados de CALCF

- Os significados de programas da linguagem CALCF podem ser apresentados como um tipo indutivo
- Tipo de dados RESULT com os constructores num e abstraction

```
num:                    Integer → RESULT
abstraction:  String × CALCF → RESULT
error:                  void → RESULT
```


Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

$$eval : CALCF \rightarrow RESULT$$

```
eval( num(n) )                ≜ n
eval( add(E1,E2) )        ≜ eval(E1) + eval(E2)
   …
eval( decl(s, E1, E2))    ≜ eval(subst(s, E2, E1))
```


Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

$$eval : CALCF \rightarrow RESULT$$

```
eval( num(n) )                ≜ n
eval( add(E1,E2) )        ≜ eval(E1) + eval(E2)
   …
eval( decl(s, E1, E2))    ≜ eval(subst(s, E2, eval(E1)))
```

calcular o valor primeiro
substitui depois...

Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

eval : CALCF → RESULT

```
eval( num(n) )                ≜ n
eval( add(E1,E2) )            ≜ eval(E1) + eval(E2)
      …
eval( decl(s, E1, E2))        ≜ eval(subst(s, E2, eval(E1)))
eval( fun(s, E) )             ≜ abstraction(s, E)
eval( call(E1, E2) )          ≜ [arg = eval(E2); fun = eval(E1);
                          if fun is abstraction(param, body)
                          then eval( subst(param, body, arg) )
                          else error ]
```


Exemplos

```
subst(x, x, 2*y ) = 2*y

subst(x, 2*4+(x+2), 2*y ) = 2*4+(2*y+2)

subst(x, decl x = 1 in x+y end, 2*y ) = decl x = 1 in x+y end

subst(x, decl z = 1 in x+z end, 2*y ) = decl z = 1 in 2*y+z end

subst(x, decl y = 1 in x+y end, 2*y ) = decl y = 1 in 2*y+y end
```

A ligação da ocorrência do identificador y que foi inserido na expressão à sua declaração foi "capturada"!!

Definição da Função Subst

```
subst(s, num(n), F )              ≜ num(n);
subst(s, id(s), F )         ≜ F;
subst(s, add(E1,E2),F )     ≜ add( subst(s, E1, F), subst
(s,E2,F));
…

subst(s, decl(s, E1, E2), F) ≜ [ /* caso s = s' */
                           G = subst(s, E1, F);
                           decl(s, G, E2);   ]

subst(s, decl(s', E1, E2), F) ≜ [ /* caso s ≠ s' */
                            G = subst(s, E1, F);
                            newid = fresh_identifier();
                            E2' = subst(s', E2, newid);
                            decl(newid, G, subst(s, E2', F)); ]
```


Definição da Função Subst

```
subst(s, num(n), F )              ≜ num(n);
subst(s, id(s), F )         ≜ F;
subst(s, add(E1,E2),F )     ≜ add( subst(s, E1, F), subst
(s,E2,F));
…

subst(s, decl(s, E1, E2), F) ≜ [ /* caso s = s' */
                           G = subst(s, E1, F);
                           decl(s, G, E2);   ]

subst(s, decl(s', E1, E2), F) ≜ [ /* caso s ≠ s' */
                            G = subst(s, E1, F);
                            newid = fresh_identifier();
                            E2' = subst(s', E2, newid);
                            decl(newid, G, subst(s, E2', F)); ]
```

A renomeação do identificador local evita a captura ilegal de ocorrências livres do identificador s' em F

Definição da Função Subst

```
subst(s, call(E1, E2) )    ≜      call( subst(s, E1,F), subst(s,
E2,F))

subst(s, fun(s', E), F) ≜ /* caso s = s' */
                          fun (s, E)

subst(s, fun(s', E), F) ≜ [/* caso s ≠ s' */
                          newid = fresh_identifier();
                          E' = subst(s', E, newid);
                          fun(newid, subst(s, E', F))]
```


Definição da Função Subst

```
subst(s, call(E1, E2) )    ≜      call( subst(s, E1,F), subst(s,
E2,F))

subst(s, fun(s', E), F) ≜ /* caso s = s' */
                          fun (s, E)

subst(s, fun(s', E), F) ≜ [/* caso s ≠ s' */
                          newid = fresh_identifier();
                          E' = subst(s', E, newid);
                          fun(newid, subst(s, E', F))]
```

A renomeação do identificador local evita a captura ilegal de ocorrências livres do identificador s' em F

Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

```
(5)
```


Quiz

- Quais os passos da avaliação do programa P segundo a semântica com substituição?
- P:

```
decl y = 3 in
  decl x = 2*y in
    decl f = (fun y -> y+x) in
      decl g = (fun x -> (fun h -> x+h(x))
        in (g(2))(f)
```

```
eval( num(n) )              ≜ n
eval( add(E1,E2) )          ≜ eval(E1) + eval(E2)
      …
eval( decl(s, E1, E2))      ≜ eval(subst(s, E2, eval(E1)))
eval( fun(s, E) )           ≜ abstraction(s, E)
eval( call(E1, E2) )        ≜ [arg = eval(E2); fun = eval(E1);
                          if fun is abstraction(param, body)
                          then eval( subst(param, body, arg) )
                          else error ]
```

Semântica de CALCF (2)

A função semântica I de CALCF:

$$I : CALCF \times ENV \rightarrow RESULT$$

CALCF = conjunto dos programas abertos

ENV = conjunto dos ambientes válidos

RESULT = conjunto dos significados (denotações)

- Um significado pode ser um valor inteiro ou uma função(representada por uma abstracção):

$$RESULT = Integer \cup Abstraction \cup \{ error \}$$

Interpretador de CALCF (2)

- Algoritmo eval(E, env) para calcular a denotação de uma expressão E de CALCF:

$$eval : CALCF \times ENV \rightarrow RESULT$$

```
eval( num(n) , env)   ≜ n
eval( id(s) , env)          ≜ env.Find(s)
…
eval( fun(s, B), env )      ≜ abstraction(s, B)
eval( call(E1, E2) , env )  ≜ fun = eval(E1, env )
                        if fun is abstraction(s, B) then
                        [ env'=env.BeginScope();
                          env'.Assoc(s, eval(E2, env )) ;
                          val = eval(B, env' );
                          env'.EndScope();
                          val ]
                        else error
```


Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```


Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g, [g=abs(y,f(y)+2), f=abs(x,x+1)]) = abs(y,f(y)+2)

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) =

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(f, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) = abs(x,x+1)

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(x+1, [x=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) = 3

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) = 5

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(x+x, [x=5, g=abs(y,f(y)+2), f=abs(x,x+1)]) = 10

Interpretador de CALCF (2)

- Outro exemplo:

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

E = [g=abs(x,x+f(x)); f=abs(y,y+x); x=1]
eval(g(2),E) =
eval(x+f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+eval(f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+eval(y+x, [y=2;x=2;g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+2+2 = 6

- Seguindo a sua intuição, qual é o valor deste programa?

Interpretador de CALCF (2)

- Outro exemplo:

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

E = [g=abs(x,x+f(x)); f=abs(y,y+x); x=1]
eval(g(2),E) =
eval(x+f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+eval(f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+eval(y+x, [y=2;x=2;g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+2+2 = 6

- O valor do programa deveria ser 5 ! O que falhou???

Resolução dinâmica de nomes

- A semântica simplificada (2) que definimos para a linguagem CALCF adopta a "regra dinâmica" de resolução de nomes (dynamic scoping).
- Segundo esta regra, os valores dos nomes **não locais** são interpretados no contexto da chamada da função, em vez do contexto da definição.
- Historicamente, algumas linguagens de programação (ex: Lisp, JavaScript 1.0) adoptaram a resolução dinâmica de nomes, por ser de implementação mais simples.
- Actualmente, é considerado um mecanismo indesejável e até incorrecto (por não respeitar o princípio da substituição), apesar de às vezes "dar jeito" interpretar nomes não locais no contexto da chamada (por exemplo: "hostname").

Princípio da substitutividade

- O valor de qualquer expressão permanece inalterado sempre que nela se substitui uma subexpressão por outra expressão com o mesmo significado / valor.
- A semântica da linguagem CALCF viola este princípio, pois os dois programa seguintes têm valores diferentes:

```
decl x=1 in
  decl f = (fun y→y+x) in
    decl g = (fun x→x+f(x))
      in g(2)
```

```
decl x=1 in
  decl f = (fun y→y+1) in
    decl g = (fun x→x+f(x))
      in g(2)
```


Resolução estática de nomes

- Todas as linguagens de programação modernas adoptam a regra da resolução estática de identificadores (static scoping).
- Segundo esta regra, os valores dos identificadores livres que ocorram no corpo de abstracções são interpretados no contexto em que as abstracções ocorrem (na definição das funções), e não no contexto da chamada.
- Para implementar esta semântica de forma eficiente é necessário usar um domínio de resultados mais rico, em que as funções são representadas por entidades chamadas "fechos".

Semântica de CALCF (3)

- A (nova) função semântica I de CALCF:

$$I : CALCF \times ENV \rightarrow RESULT$$

CALCF = conjunto dos programas abertos

ENV = conjunto dos ambientes

RESULT = conjunto dos significados (denotações)

Os resultados podem ser valores inteiros, fechos (uma abstracção + um ambiente), ou um erro.

$$RESULT = Integer \cup Closure \cup \{ error \}$$

Resultados de CALCF

- Os significados de programas da linguagem CALCF podem ser apresentados como um tipo indutivo

Tipo de dados RESULT com os constructores num e closure

```
num:      Integer → RESULT
closure:  String × CALCF × ENV → RESULT
error:    void → RESULT
```

Um fecho representa uma função através de um triplo contendo o parâmetro, o corpo, e o ambiente que regista os valores dos nomes livres no corpo

Assim, ao contrário de uma abstracção, um fecho é efectivamente um valor "fechado" (não depende de nenhum nome externo).

Ambiente "mutável"

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável:

```
Environ BeginScope()
```

  - Cria um novo nível **vazio**, onde serão colocadas as ligações de um novo âmbito local.
  - Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível.

```
Environ EndScope()
```

  - Devolve o ambiente no estado anterior à última operação BeginScope().
  - Mas **não destrói** o último nível pois podem existir no contexto de execução fechos que o referem!

Interpretador de CALCF (3)

- Algoritmo eval(E, env) para calcular o valor de uma expressão E de CALCF:

eval : CALCF × ENV → RESULT

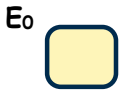
```
eval( fun(s, B), env )      ≜ closure(s, env, B)
eval( call(E1, E2) , env )  ≜ fun = eval(E1, env )
                            if fun is closure(s, envc, B) then
                            [ envloc = envc.BeginScope();
                              envloc.Assoc(s, eval(E2, env )) ;
                              val = eval(B, envloc );
                              envc = envloc.EndScope();
                              val ]
                            else error
```


Avaliação de Funções

- Exemplo: avaliar o seguinte programa, na semântica usando ambientes e fechos.

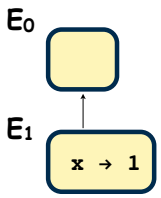
```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```


Exemplo



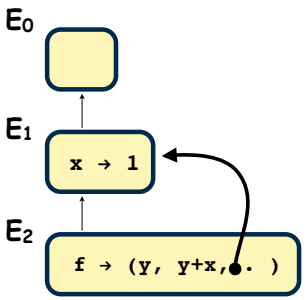

```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```

Exemplo



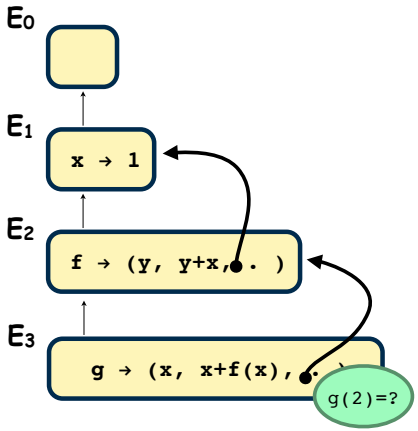

```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```


Exemplo



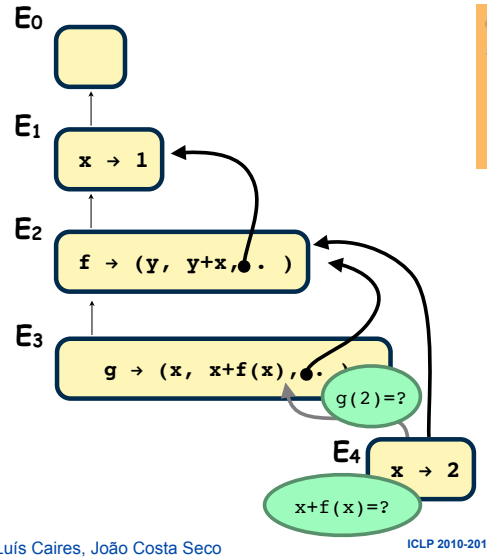

```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```


Exemplo



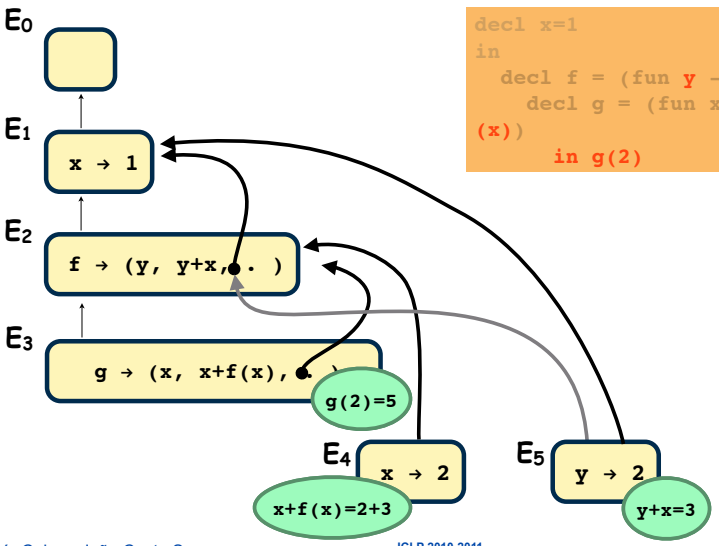

```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```


Exemplo




```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```

Exemplo



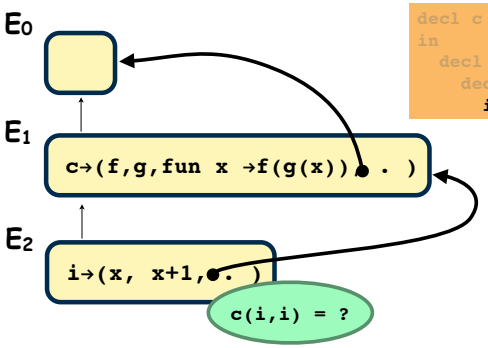

```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```


Avaliação de Funções

- Exemplo: avaliar o seguinte programa, na semântica usando ambientes e fechos.

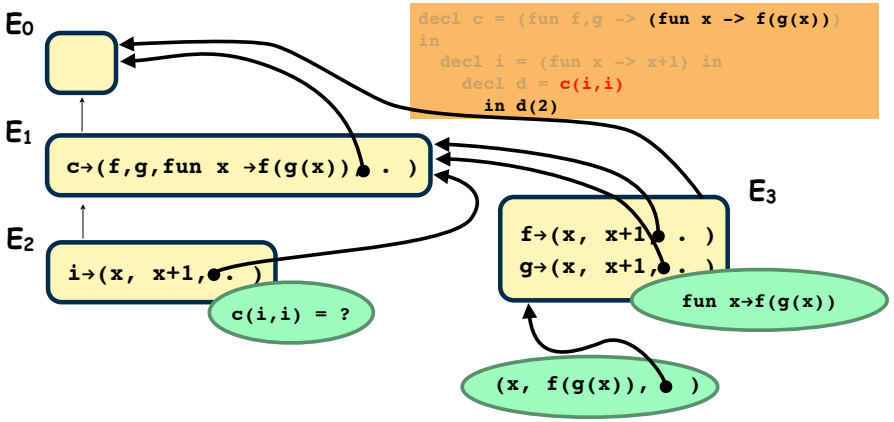
```
decl comp = (fun f,g -> (fun x -> f(g(x)))) in
  decl inc = (fun x -> x+1) in
    decl dup = comp(inc,inc)
      in dup(2)
```


Avaliação de Funções



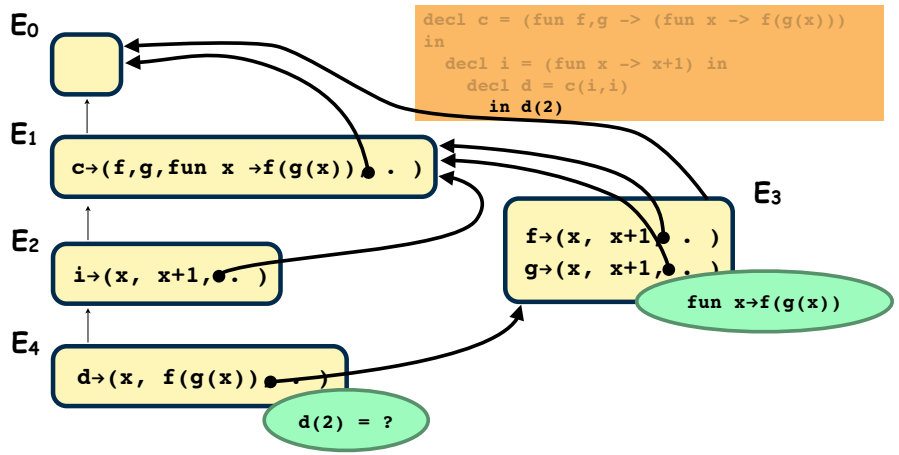

```
decl c = (fun f,g -> (fun x -> f(g(x)))
in
  decl i = (fun x -> x+1) in
    decl d = c(i,i)
      in d(2)
```


Avaliação de Funções

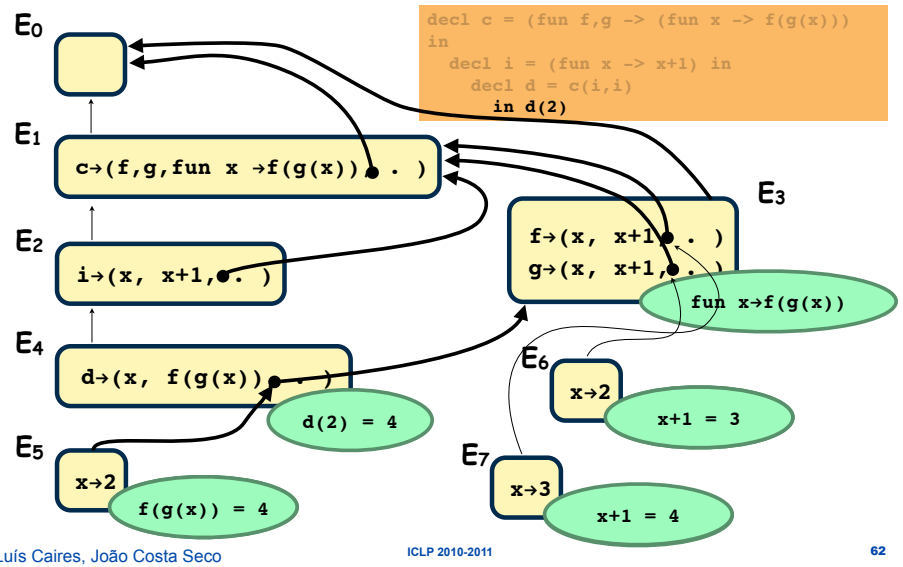



```
decl c = (fun f,g -> (fun x -> f(g(x)))
in
  decl i = (fun x -> x+1) in
    decl d = c(i,i)
      in d(2)
```

Avaliação de Funções



Avaliação de Funções



Quiz

- Qual o valor (se existir) das seguintes expressões:

```
decl f = (fun x -> x+1) in
  decl g = (fun y -> y(2)) in g(f) end end

decl f = (fun x -> x(x)) in f(f) end
```

- Qual o valor da expressão seguinte quando avaliada pela regra dinâmica e pela regra estática de resolução de nomes:

```
decl x=2 in
decl g = (fun y -> y-x) in
decl x = 4 in g(x) end end end
```

- Considera que as duas expressões seguintes têm sempre o mesmo valor? Porquê?

```
decl id = E1 in E2 end
(fun id -> E2)( E1)
```

Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2010-2011

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Unidade 7: Abstracção funcional

Nas linguagens de programação existem várias formas de tornar código mais abstracto e reutilizável em diferentes contextos. A abstracção tem por objectivo aumentar o nível de detalhe com que se olha para um problema e pode ser conseguida ao nível da funcionalidade, dos dados, da iteração de colecções, através da hierarquização da informação, etc.
Todas as linguagens de programação modernas possuem uma ou mais destas formas de abstracção, e normalmente todas incluem suporte primitivo para a abstracção funcional por parametrização.

Abstracção por parametrização
Equivalência alfa
A linguagem CALCF
Semântica com substituição da linguagem CALCF
Algoritmo interpretador da linguagem CALCF com ambiente
Estratégias de resolução de nomes
Algoritmo compilador da linguagem CALCF
Declarações recursivas
Passagem de parâmetros (por valor, por nome)

Abstracção por Parametrização

- As linguagens de programação têm um número finito de construções que podem ser usadas para representar um número infinito de computações.
- Podemos capturar grupos de computações semelhantes através da abstracção de alguns das suas componentes (as que variam).
- A abstracção é uma forma de enriquecer as linguagens de programação com novas operações, com um nível mais alto (manipulam entidades mais complexas como um todo) definidas a partir de outras construções da linguagem.

```
1*1 + 2*2
1*1 + 1*2
2*2 + 1*1
1*2 + 2*1
x*x + y*y
2*2 + 3*2
5*5 + 7*7
2*2 + 9*9
1*1 + 2*2
3*1 + 1*2
```


Abstracção por Parametrização

- É a possibilidade de definir entidades genéricas, introduzidas uma única vez e utilizáveis várias vezes num programa com diferentes instanciações de um conjunto de parâmetros definidos.
- É uma característica fundamental de todas as linguagens de programação modernas:
  - Funções / Procedimentos
  - Métodos
  - Tipos paramétricos
  - Classes paramétricas
  - etc...
- A parametrização e a abstracção são essenciais à modularidade dos programas, e em alguns casos, à expressividade computacional das linguagens.

Parametrização

- É o mecanismo geral para declarar os nomes que se querem encarar como genéricos, destacando-os sintacticamente através de notação conveniente:

  C: `int f(int x) { return x+1;}`

  ML: `(fun x -> x+1)`

  Lisp: `(lambda (x) (add x 1))`

  Cálculo Lambda (Church): $\lambda x.x$

- Tecnicamente, chama-se abstracção a uma construção sintáctica, explicitamente parametrizada num certo conjunto de nomes.

Abstracção

- Uma abstracção é uma construção sintáctica constituída por dois elementos fundamentais:
  - As declarações dos seus parâmetros
  - O corpo da abstracção (uma qualquer construção da linguagem)

  `(x → x+y)`

  Parâmetros declarados: `x`; Corpo: `x+y`

  `(y → (x → x+y))`

  Parâmetros declarados: `y`; Corpo: `(x → x+y)`

  `(y,x → x+y)`

  Parâmetros declarados: `x,y`; Corpo: `x+y`

- Uma abstracção representa uma função anónima dos seus parâmetros...

Observações

- As declarações dos parâmetros de uma abstracção são ocorrências ligantes dos nomes respectivos.
- O âmbito das declarações dos parâmetros é (exactamente) o corpo da abstracção.
- Os nomes livres duma abstracção são todos os nomes livres no seu corpo que não são parâmetros

$NomesLivres((x_1,\ldots,x_n \rightarrow E) = NomesLivres(E) \setminus \{x_1,\ldots,x_n\}$

exemplo:

`NomesLivres((x → x+y)) = NomesLivres(x+y)\{x} = {y}`

Observações

- As declarações dos parâmetros de uma abstracção são ocorrências ligantes dos nomes respectivos.
- O âmbito das declarações dos parâmetros é (exactamente) o corpo da abstracção.
- Os nomes livres duma abstracção são todos os nomes livres no seu corpo que não são parâmetros

```
NomesLivres(x) = {x}
NomesLivres(E1 op E2) = NomesLivres(E1) U NomesLivres(E2)
NomesLivres(decl x = E1 in E2) = NomesLivres(E1) U NomesLivres(E2)\{x}
```

exemplos:

```
NomesLivres(x+decl y = x in x+y end) = {x}
NomesLivres(decl x = 1 in decl y = z in x+y end end) = {z}
```

Equivalência-alfa ($=_\alpha$)

- Duas abstracções dizem-se alfa-equivalentes se uma pode ser obtida a partir da outra através da renomeação dos seus parâmetros, evitando conflitos com os seus nomes livres:

  `(x → x+y)` $=_\alpha$ `(z → z+y)`

  `(x → x+y)` $\neq_\alpha$ `(y → y+y)`

- Abstracções alfa-equivalentes são semanticamente equivalentes (são interpretadas como denotando a mesma função):

  `(x → x+1)` $=_\alpha$ `(z → z+1)`

- Por isso, um interpretador pode traduzir os nomes dos parâmetros em algo mais conveniente e conhecido apenas localmente...

O que denota uma abstracção?

- Uma abstracção é uma expressão sintáctica que denota uma certa função.
- Uma função é uma entidade semântica primitiva que suporta uma operação de aplicação, tal como um valor inteiro é uma entidade semântica primitiva que suporta a operação de adição, etc.
- Uma linguagem pode suportar funções mas não abstracções (exemplo: C, C++, Pascal)
- Várias linguagens suportam abstracções (ML, Smalltalk, Python, Javascript, Java (usando objectos anónimos))

Abstracções vs. Declarações

- Em C e Pascal, as expressões que representam funções aparecem sempre no contexto de uma declaração:
  - `int f(int x) { return x+1; } (`*`resto do programa`*`)`
- Em ML, o mecanismo de declaração está separado do mecanismo de abstracção:
  - `let x=2 in (resto do programa)`
  - `let f = (fun x->x+1) in (resto do programa)`
  - `(map (fun x-> x+1) [1;2;3]) = [2;3;4]`
- Em ML, as funções são "cidadãs de primeira classe" (first-class citizens) ou seja, são tratadas como qualquer outro valor da linguagem. O mesmo não se passa com as funções em Pascal ou C.

Abstracções (exemplos)

- Smalltalk
  - [:x | E] representa um bloco de programa, e é uma abstracção
  - 1 to: 10 do: [:i | s ← s+i] é a forma de fazer um ciclo...
- Abstracções em linguagens de objectos
  - Uma abstracção pode ser representada por um objecto que implementa um método "apply" (Function object)
  - Exemplo em Java...

```
Interface Function { int apply(int x); }
...
Function f = new Function{
       int apply(int x) {
                return x+1;
       }
} // f  = λx. x+1
...
int v = f.apply(2) // v = (f 2)
```

Abstracções (exemplos)

- As abstracções podem realizar-se não apenas sobre identificadores de valores, mas também sobre outras entidades, como por exemplo tipos.
- C++

```
template <class T> class Stack { int push(const &T); … }
```

- Java (5.0)

```
interface List<E> { void add(E x); Iterator<E> iterator(); }

interface Iterator<E> { E next(); boolean hasNext(); }

class LinkedList<E> implements List<E> { … }
```


Exemplo: A Linguagem CALCF

- A linguagem CALCF estende a linguagem CALCI com funções, representadas por abstracções:

```
fun Id -> Expression end
```

- As funções podem ser aplicadas ao seu argumento, usando a expressão de aplicação:

```
Expression1 ( Expression2 )
```

- Semântica pretendida:
  Se ***Expression1*** denota uma função f, e ***Expression2*** denota um valor qualquer v, então ***Expression1*(Expression2)*** denota o valor que resulta de aplicar f a v.

Exemplo: A Linguagem CALCF

- Um programa simples:

```
fun x -> x+2 end (4)
```

- Um outro exemplo:

```
decl f= fun x -> x+1 end in
  decl g = fun y -> f(y)+2 end in
    decl x = g(2)
      in x+x
```

- O mesmo exemplo numa sintaxe concreta tipo C:

```
function f(x){ return x+1;}
function g(y){ f(y)+2 }
{ ... x = g(2); return x+x; }
```


A Linguagem CALCF

- Tipo de dados CALCF com os constructores:

```
num:   Integer → CALCF
add:   CALCF × CALCF → CALCF
mul:   CALCF × CALCF → CALCF
div:   CALCF × CALCF → CALCF
sub:   CALCF × CALCF → CALCF
id:    String → CALCF
decl:  String × CALCF × CALCF → CALCF
fun:   String × CALCF → CALCF
call:  CALCF × CALCF → CALCF
```

Semântica de CALCF (1)

A função semântica I de CALCF:

$$I : CALCF \rightarrow RESULT$$

CALCF = conjunto dos programas fechados

RESULT = conjunto dos significados (denotações)

- Um significado pode ser um valor inteiro ou uma função (representada por uma abstracção):

$$RESULT = Integer \cup Abstraction \cup \{ error \}$$

Resultados de CALCF

- Os significados de programas da linguagem CALCF podem ser apresentados como um tipo indutivo
- Tipo de dados RESULT com os constructores num e abstraction

```
num:                    Integer → RESULT
abstraction:  String × CALCF → RESULT
error:                  void → RESULT
```


Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

$$eval : CALCF \rightarrow RESULT$$

```
eval( num(n) )                ≜ n
eval( add(E1,E2) )        ≜ eval(E1) + eval(E2)
   …
eval( decl(s, E1, E2))    ≜ eval(subst(s, E2, E1))
```


Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

$$eval : CALCF \rightarrow RESULT$$

```
eval( num(n) )                ≜ n
eval( add(E1,E2) )        ≜ eval(E1) + eval(E2)
   …
eval( decl(s, E1, E2))    ≜ eval(subst(s, E2, eval(E1)))
```

calcular o valor primeiro
substitui depois...

Interpretador de CALCF (1)

- Algoritmo "de referência" eval(E) para calcular o valor de uma expressão fechada E da linguagem CALCF:

eval : CALCF → RESULT

```
eval( num(n) )                ≜ n
eval( add(E1,E2) )            ≜ eval(E1) + eval(E2)
      …
eval( decl(s, E1, E2))        ≜ eval(subst(s, E2, eval(E1)))
eval( fun(s, E) )             ≜ abstraction(s, E)
eval( call(E1, E2) )          ≜ [arg = eval(E2); fun = eval(E1);
                          if fun is abstraction(param, body)
                          then eval( subst(param, body, arg) )
                          else error ]
```


Exemplos

```
subst(x, x, 2*y ) = 2*y

subst(x, 2*4+(x+2), 2*y ) = 2*4+(2*y+2)

subst(x, decl x = 1 in x+y end, 2*y ) = decl x = 1 in x+y end

subst(x, decl z = 1 in x+z end, 2*y ) = decl z = 1 in 2*y+z end

subst(x, decl y = 1 in x+y end, 2*y ) = decl y = 1 in 2*y+y end
```

A ligação da ocorrência do identificador y que foi inserido na expressão à sua declaração foi "capturada"!!

Definição da Função Subst

```
subst(s, num(n), F )              ≜ num(n);
subst(s, id(s), F )         ≜ F;
subst(s, add(E1,E2),F )     ≜ add( subst(s, E1, F), subst
(s,E2,F));
…

subst(s, decl(s, E1, E2), F) ≜ [ /* caso s = s' */
                           G = subst(s, E1, F);
                           decl(s, G, E2);   ]

subst(s, decl(s', E1, E2), F) ≜ [ /* caso s ≠ s' */
                            G = subst(s, E1, F);
                            newid = fresh_identifier();
                            E2' = subst(s', E2, newid);
                            decl(newid, G, subst(s, E2', F)); ]
```


Definição da Função Subst

```
subst(s, num(n), F )              ≜ num(n);
subst(s, id(s), F )         ≜ F;
subst(s, add(E1,E2),F )     ≜ add( subst(s, E1, F), subst
(s,E2,F));
…

subst(s, decl(s, E1, E2), F) ≜ [ /* caso s = s' */
                           G = subst(s, E1, F);
                           decl(s, G, E2);   ]

subst(s, decl(s', E1, E2), F) ≜ [ /* caso s ≠ s' */
                            G = subst(s, E1, F);
                            newid = fresh_identifier();
                            E2' = subst(s', E2, newid);
                            decl(newid, G, subst(s, E2', F)); ]
```

A renomeação do identificador local evita a captura ilegal de ocorrências livres do identificador s' em F

Definição da Função Subst

```
subst(s, call(E1, E2) )    ≜      call( subst(s, E1,F), subst(s,
E2,F))

subst(s, fun(s', E), F) ≜ /* caso s = s' */
                          fun (s, E)

subst(s, fun(s', E), F) ≜ [/* caso s ≠ s' */
                          newid = fresh_identifier();
                          E' = subst(s', E, newid);
                          fun(newid, subst(s, E', F))]
```


Definição da Função Subst

```
subst(s, call(E1, E2) )    ≜      call( subst(s, E1,F), subst(s,
E2,F))

subst(s, fun(s', E), F) ≜ /* caso s = s' */
                          fun (s, E)

subst(s, fun(s', E), F) ≜ [/* caso s ≠ s' */
                          newid = fresh_identifier();
                          E' = subst(s', E, newid);
                          fun(newid, subst(s, E', F))]
```

A renomeação do identificador local evita a captura ilegal de ocorrências livres do identificador s' em F

Interpretador de CALCF (1)

- Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

```
(5)
```


Quiz

- Quais os passos da avaliação do programa P segundo a semântica com substituição?
- P:

```
decl y = 3 in
  decl x = 2*y in
    decl f = (fun y -> y+x) in
      decl g = (fun x -> (fun h -> x+h(x))
        in (g(2))(f)
```

```
eval( num(n) )              ≜ n
eval( add(E1,E2) )          ≜ eval(E1) + eval(E2)
      …
eval( decl(s, E1, E2))      ≜ eval(subst(s, E2, eval(E1)))
eval( fun(s, E) )           ≜ abstraction(s, E)
eval( call(E1, E2) )        ≜ [arg = eval(E2); fun = eval(E1);
                          if fun is abstraction(param, body)
                          then eval( subst(param, body, arg) )
                          else error ]
```

Semântica de CALCF (2)

A função semântica I de CALCF:

$$I : CALCF \times ENV \rightarrow RESULT$$

CALCF = conjunto dos programas abertos

ENV = conjunto dos ambientes válidos

RESULT = conjunto dos significados (denotações)

- Um significado pode ser um valor inteiro ou uma função(representada por uma abstracção):

$$RESULT = Integer \cup Abstraction \cup \{ error \}$$

Interpretador de CALCF (2)

- Algoritmo eval(E, env) para calcular a denotação de uma expressão E de CALCF:

$$eval : CALCF \times ENV \rightarrow RESULT$$

```
eval( num(n) , env)   ≜ n
eval( id(s) , env)          ≜ env.Find(s)
…
eval( fun(s, B), env )      ≜ abstraction(s, B)
eval( call(E1, E2) , env )  ≜ fun = eval(E1, env )
                        if fun is abstraction(s, B) then
                        [ env'=env.BeginScope();
                          env'.Assoc(s, eval(E2, env )) ;
                          val = eval(B, env' );
                          env'.EndScope();
                          val ]
                        else error
```


Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```


Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g, [g=abs(y,f(y)+2), f=abs(x,x+1)]) = abs(y,f(y)+2)

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) =

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(f, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) = abs(x,x+1)

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(x+1, [x=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) = 3

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(g(2), [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(f(y)+2, [y=2, g=abs(y,f(y)+2), f=abs(x,x+1)]) = 5

Interpretador de CALCF (2)

- Avaliação do programa:

```
decl f=(fun x -> x+1) in
  decl g = (fun y -> f(y)+2) in
    decl x = g(2)
      in x+x
```

eval(P1,0) =
eval(P2, [f=abs(x,x+1)]) =
eval(P3, [g=abs(y,f(y)+2), f=abs(x,x+1)]) =
eval(x+x, [x=5, g=abs(y,f(y)+2), f=abs(x,x+1)]) = 10

Interpretador de CALCF (2)

- Outro exemplo:

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

E = [g=abs(x,x+f(x)); f=abs(y,y+x); x=1]
eval(g(2),E) =
eval(x+f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+eval(f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+eval(y+x, [y=2;x=2;g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+2+2 = 6

- Seguindo a sua intuição, qual é o valor deste programa?

Interpretador de CALCF (2)

- Outro exemplo:

```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```

E = [g=abs(x,x+f(x)); f=abs(y,y+x); x=1]
eval(g(2),E) =
eval(x+f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+eval(f(x), [x=2; g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+eval(y+x, [y=2;x=2;g=abs(x,x+f(x)),f=abs(y, y+x)]) =
2+2+2 = 6

- O valor do programa deveria ser 5 ! O que falhou???

Resolução dinâmica de nomes

- A semântica simplificada (2) que definimos para a linguagem CALCF adopta a "regra dinâmica" de resolução de nomes (dynamic scoping).
- Segundo esta regra, os valores dos nomes **não locais** são interpretados no contexto da chamada da função, em vez do contexto da definição.
- Historicamente, algumas linguagens de programação (ex: Lisp, JavaScript 1.0) adoptaram a resolução dinâmica de nomes, por ser de implementação mais simples.
- Actualmente, é considerado um mecanismo indesejável e até incorrecto (por não respeitar o princípio da substituição), apesar de às vezes "dar jeito" interpretar nomes não locais no contexto da chamada (por exemplo: "hostname").

Princípio da substitutividade

- O valor de qualquer expressão permanece inalterado sempre que nela se substitui uma subexpressão por outra expressão com o mesmo significado / valor.
- A semântica da linguagem CALCF viola este princípio, pois os dois programa seguintes têm valores diferentes:

```
decl x=1 in
  decl f = (fun y→y+x) in
    decl g = (fun x→x+f(x))
      in g(2)
```

```
decl x=1 in
  decl f = (fun y→y+1) in
    decl g = (fun x→x+f(x))
      in g(2)
```


Resolução estática de nomes

- Todas as linguagens de programação modernas adoptam a regra da resolução estática de identificadores (static scoping).
- Segundo esta regra, os valores dos identificadores livres que ocorram no corpo de abstracções são interpretados no contexto em que as abstracções ocorrem (na definição das funções), e não no contexto da chamada.
- Para implementar esta semântica de forma eficiente é necessário usar um domínio de resultados mais rico, em que as funções são representadas por entidades chamadas "fechos".

Semântica de CALCF (3)

- A (nova) função semântica I de CALCF:

$$I : CALCF \times ENV \rightarrow RESULT$$

CALCF = conjunto dos programas abertos

ENV = conjunto dos ambientes

RESULT = conjunto dos significados (denotações)

Os resultados podem ser valores inteiros, fechos (uma abstracção + um ambiente), ou um erro.

$$RESULT = Integer \cup Closure \cup \{ error \}$$

Resultados de CALCF

- Os significados de programas da linguagem CALCF podem ser apresentados como um tipo indutivo

Tipo de dados RESULT com os constructores num e closure

```
num:      Integer → RESULT
closure:  String × CALCF × ENV → RESULT
error:    void → RESULT
```

Um fecho representa uma função através de um triplo contendo o parâmetro, o corpo, e o ambiente que regista os valores dos nomes livres no corpo

Assim, ao contrário de uma abstracção, um fecho é efectivamente um valor "fechado" (não depende de nenhum nome externo).

Ambiente "mutável"

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável:

```
Environ BeginScope()
```

  - Cria um novo nível **vazio**, onde serão colocadas as ligações de um novo âmbito local.
  - Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível.

```
Environ EndScope()
```

  - Devolve o ambiente no estado anterior à última operação BeginScope().
  - Mas **não destrói** o último nível pois podem existir no contexto de execução fechos que o referem!

Interpretador de CALCF (3)

- Algoritmo eval(E, env) para calcular o valor de uma expressão E de CALCF:

eval : CALCF × ENV → RESULT

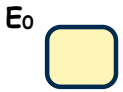
```
eval( fun(s, B), env )      ≜ closure(s, env, B)
eval( call(E1, E2) , env )  ≜ fun = eval(E1, env )
                            if fun is closure(s, envc, B) then
                            [ envloc = envc.BeginScope();
                              envloc.Assoc(s, eval(E2, env )) ;
                              val = eval(B, envloc );
                              envc = envloc.EndScope();
                              val ]
                            else error
```


Avaliação de Funções

- Exemplo: avaliar o seguinte programa, na semântica usando ambientes e fechos.

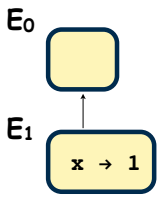
```
decl x=1 in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f(x))
      in g(2)
```


Exemplo



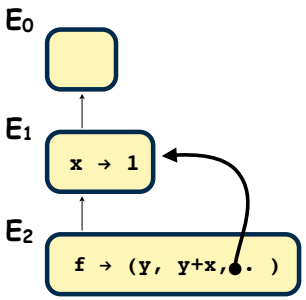

```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```

Exemplo



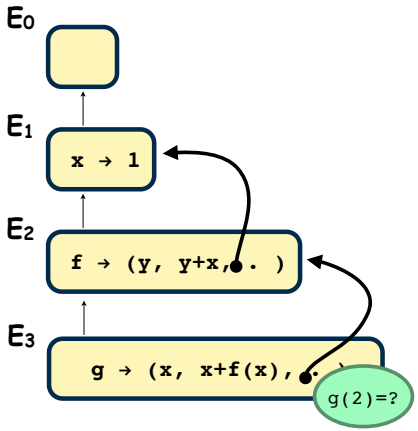

```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```


Exemplo



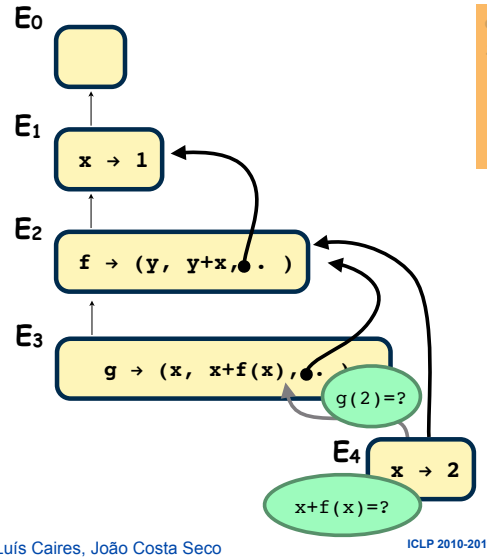

```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```


Exemplo



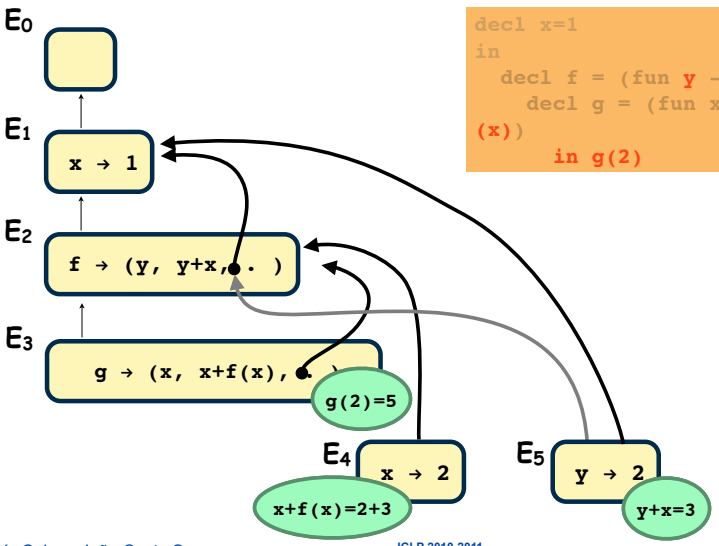

```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```


Exemplo




```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```

Exemplo



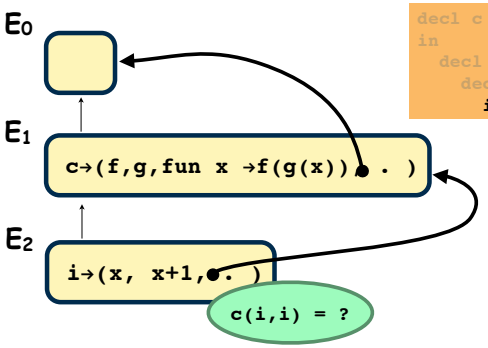

```
decl x=1
in
  decl f = (fun y -> y+x) in
    decl g = (fun x -> x+f
(x))
      in g(2)
```


Avaliação de Funções

- Exemplo: avaliar o seguinte programa, na semântica usando ambientes e fechos.

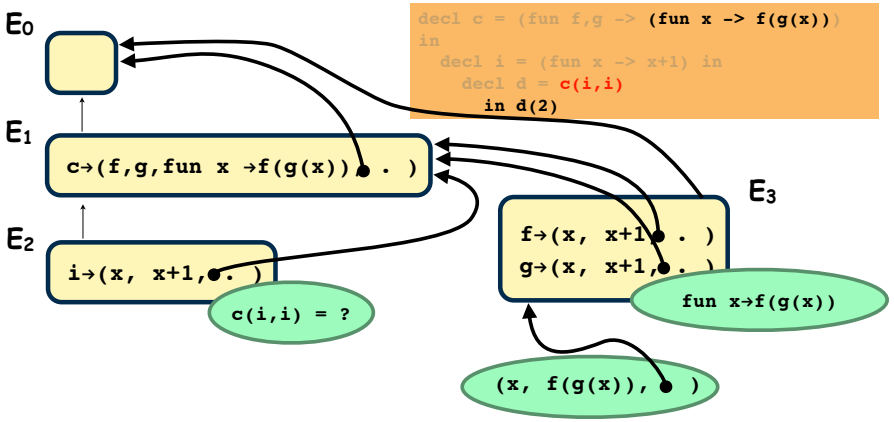
```
decl comp = (fun f,g -> (fun x -> f(g(x)))) in
  decl inc = (fun x -> x+1) in
    decl dup = comp(inc,inc)
      in dup(2)
```


Avaliação de Funções



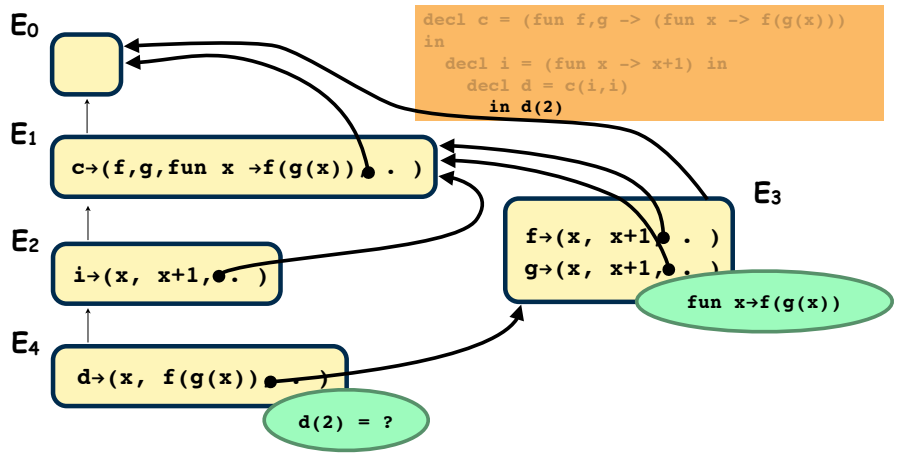

```
decl c = (fun f,g -> (fun x -> f(g(x)))
in
  decl i = (fun x -> x+1) in
    decl d = c(i,i)
      in d(2)
```


Avaliação de Funções

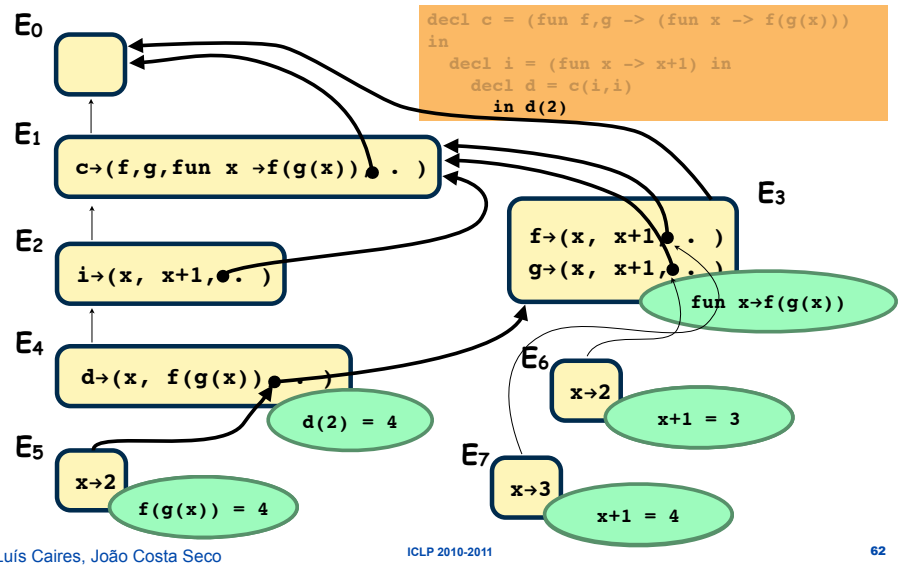



```
decl c = (fun f,g -> (fun x -> f(g(x)))
in
  decl i = (fun x -> x+1) in
    decl d = c(i,i)
      in d(2)
```

Avaliação de Funções



Avaliação de Funções



Quiz

- Qual o valor (se existir) das seguintes expressões:

```
decl f = (fun x -> x+1) in
  decl g = (fun y -> y(2)) in g(f) end end

decl f = (fun x -> x(x)) in f(f) end
```

- Qual o valor da expressão seguinte quando avaliada pela regra dinâmica e pela regra estática de resolução de nomes:

```
decl x=2 in
decl g = (fun y -> y-x) in
decl x = 4 in g(x) end end end
```

- Considera que as duas expressões seguintes têm sempre o mesmo valor? Porquê?

```
decl id = E1 in E2 end
(fun id -> E2)( E1)
```


Quiz (solução)

- Considera que as duas expressões seguintes têm sempre o mesmo valor? Porquê?

```
decl id = E1 in E2 end

(fun id -> E2)( E1 )
```

- Temos (aplicando as regras de avaliação):

```
eval(decl id = E1 in E2 end, env) =
  eval(E2, env.Assoc(id, eval(E1, env )))
```

- Por outro lado:

```
eval((fun id -> E2), env ) = closure(id, E2, env )

eval((fun id -> E2) (E1), env ) =
  eval(E2, env.Assoc(id, eval(E1, env )))
```

Resumo e Leituras

- Abstracção por parametrização é um mecanismo aplicado a um subprograma que o generaliza, abstraindo o valor de certas subexpressões em parâmetros bem identificados) e permite a sua instanciação e reutilização aplicada a diferentes contextos.
- Uma abstracção é uma construção sintáctica composta por um conjunto de parâmetros e um subprograma.
- Cada instanciação de uma abstracção é activada em associação com uma interpretação dos seus parâmetros e nomes livres.
- Leituras:
  - Liskov, Guttag "Program Development in Java"
  - Friedmand "Essentials of programming languages" 3rd edition, Cap 3.
  - Mitchell, "Concepts in programming languages", Cap 7
  - Appel, Cap. 15, "Modern Compiler Implementation"

Compilação de CALCF (1)

- A compilação de CALCF pode ser caracterizada por uma função:

  comp: P × ENV → CodeSeq

  P = Fragmento de programa (aberto)

  ENV = Ambiente (função String → int)

  O ambiente atribui a cada identificador uma posição na pilha de chamada (um número inteiro).

  CodeSeq = Sequências de instruções

Ingredientes...

Chamada indirecta de funções

- A máquina CLR dispõe de instruções para a chamada de funções indirectamente a partir de valores (referências ou apontadores).

```
ldstr "Hello"
ldftn void f(object)
calli void (object)
```

- Instruções da linguagem CIL para referir e chamar funções por apontadores:
  - ldftn assinatura

    Coloca no topo da pilha um apontador para o método com a assinatura dada (completa com tipo de retorno, nome e tipos de parâmetros). ... -> ftn

    o tipo CIL dos apontadores é `native int`, em C# é `System.IntPtr`
  - calli assinatura

    Chamada indirecta de métodos. Retira do topo da pilha os argumentos e o apontador para um método a chamar. Verifica se a assinatura dada (sem nome) é compatível com o método referido pelo apontador. Deixa o resultado (se houver) no topo da pilha.

    arg1, arg2, ftn -> retVal

Chamada indirecta de funções

- A estrutura de uma assembly CIL é composta por um conjunto de declarações: métodos e classes.

```
.assembly 'A' {}
.module A.dll
.class public auto ansi A extends [mscorlib]
System.Object
{
  .field private int32 x
  .method ...
}
.method ... static ... object m0(object) {...}
.method ... static ... object m1(object) {
  .locals (int32)
  .entrypoint
  ...
  ret
}
```


Chamada indirecta de funções

- A chamada indirecta de métodos estáticos passa numa primeira fase por produzir um apontador para o método (sabendo qual a assinatura completa) e numa segunda fase chamar o método tendo o apontador e sabendo a sua assinatura (sem nome).

```
.method static public void m0(object) {
  .locals init (native int)
  ldftn void m1(object)
  stloc.0
  ldstr "Hello"
  ldloc.0
  calli void(object)
  ret
}
.method static public void m1(object) {
  ldarg.0
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```


Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
```


Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
.method ... static ... object f0(...) {...

.method ... static ... object f1(...) {...

.method ... static ... object f2(...) {...

.method ... static ... object f3(...) {...
```

Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
.method ... static ... object f0(...) {...
[[ y + x ]]

.method ... static ... object f1(...) {...

.method ... static ... object f2(...) {...

.method ... static ... object f3(...) {...
```


Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
.method ... static ... object f0(...) {...

.method ... static ... object f1(...) {...
[[ f(x)+1 ]]

.method ... static ... object f2(...) {...

.method ... static ... object f3(...) {...
```


Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
.method ... static ... object f0(...) {...

.method ... static ... object f1(...) {...

.method ... static ... object f2(...) {...
[[ (fun x -> x)(g(y)) ]]

.method ... static ... object f3(...) {...
```


Compilação de CALCF

Um programa CALCF contém abstracções anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral**: Associar a cada abstracção CALCF um método estático em CIL que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
 ...
}
.method ... static ... object f0(...) {...

.method ... static ... object f1(...) {...

.method ... static ... object f2(...) {...

.method ... static ... object f3(...) {...
[[ x ]]
```

Compilação de CALCF

- A compilação de CALCF pode ser caracterizada por uma função:

  comp: P × ENV → CodeSeq x CodeSeq list

  P = Fragmento de programa (aberto)

  ENV = Ambiente (função String → int)

O ambiente atribui a cada identificador uma posição na pilha de chamada (um número inteiro).

CodeSeq = Sequências de instruções

Compilação de CALCF

No processo de compilação das linguagens anteriores, os valores (constantes) associados aos identificadores são guardados em vectores locais ao método CIL que implementa um programa. Ao espalhar a compilação de um programa por vários métodos CIL perde-se o acesso aos identificadores que não estão no ambiente mais próximo. A linguagem CIL não suporta o aninhamento de funções da linguagem CALCF, i.e. as variáveis locais CIL são apenas acessíveis dentro do método.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
end
```

```
.method ... static  ... void Main () {
  .locals (object[] table)
  ...
}
.method ... static ... object f0(object y) {
  ldloc table // x ????
  ...
}
.method ... static ... object f1(object x) {
  ldloc table // g ????
  ...
}
...
```


Resolução estática de nomes...

... em Linguagens tipo Algol

- As linguagens tipo Algol suportam aninhamento (nesting) de declarações e o acesso a variáveis não locais é feito através de ligações entre registos na pilha de execução (static link).

```
Program P;
var x:integer;

Function f0(y:integer):integer;
  var z:integer;

    Function f1(w:integer):integer;
    begin
     f1 := w+y+x+z
    end

    Function f2(w:integer):integer;
    begin
      f2 := f3(f1)
    end

  begin
    f0 := f2(f1(y))
  end
```

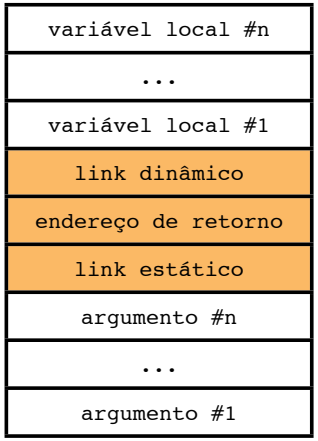
```
Function f3(function f
(y:integer):integer):integer;
  begin
    f3 := f(x)+x
  end

Function f4(x:integer):integer;
  begin
    f4 := x+1
  end


begin
  x := 1;
  f4(f0(1))
end.
```

... em Linguagens tipo Algol

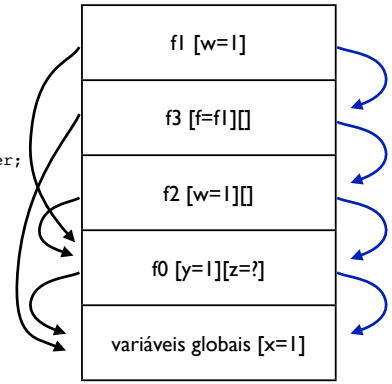
- Acesso a identificadores compilado para acesso a endereços de memória na stack.
- Cada chamada de uma função ou procedimento gera um bloco na pilha com informação relavante:
  - Endereço de retorno
  - Link dinâmico (bloco da função que o chamou)
  - Link estático (bloco envolvente no programa)
  - Memória para parâmetros/argumentos
  - Memória para variáveis locais
- Os blocos são desempilhados quando a função retorna.


```
Program P;
var x:integer;

Function f0(y:integer):Integer;
  var z:integer;

  Function f1(w:integer):Integer;
    begin
     f1 := w+y+x+z
    end

  Function f2(w:integer):Integer;
    begin
      f2 := f3(f1)
    end

  begin
    f0 := f2(y)
  end

Function f3(function f(y:integer):integer):Integer;
  begin
    f3 := f(x)+x
  end

Function f4(x:integer):Integer;
   ...

begin
  x := 1;
  f0(1)
end.
```

Stack frame

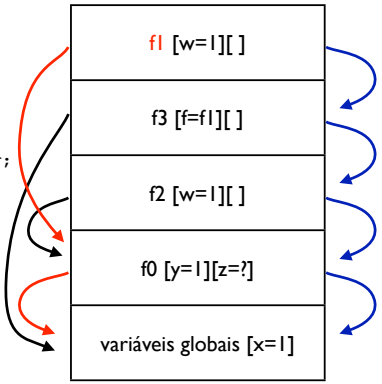



```
Program P;
var x:integer;

Function f0(y:integer):Integer;
  var z:integer;

  Function f1(w:integer):Integer;
    begin
     f1 := w+y+x+z
    end

  Function f2(w:integer):Integer;
    begin
      f2 := f3(f1)
    end

  begin
    f0 := f2(y)
  end

Function f3(function f(y:integer):integer):Integer;
  begin
    f3 := f(x)+x
  end

Function f4(x:integer):Integer;
   ...

begin
  x := 1;
  f0(1)
end.
```



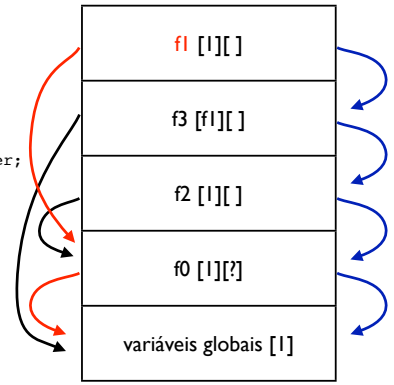

```
Program P;
var x:integer;

Function f0(y:integer):Integer;
  var z:integer;

  Function f1(w:integer):Integer;
    begin
     f1 := w+y+x+z
    end

  Function f2(w:integer):Integer;
    begin
      f2 := f3(f1)
    end

  begin
    f0 := f2(y)
  end

Function f3(function f(y:integer):integer):Integer;
  begin
    f3 := f(x)+x
  end

Function f4(x:integer):Integer;
   ...

begin
  x := 1;
  f0(1)
end.
```

id	(saltos,offset)
w	(0,p#1)
y	(1,p#1)
x	(2,v#1)
z	(1,v#1)



```
Program P;
var x:integer;

Function f0(y:integer):Integer;
  var z:integer;

  Function f1(w:integer):Integer;
    begin
     f1 := w+y+x+z
    end

  Function f2(w:integer):Integer;
    begin
      f2 := f3(f1)
    end

  begin
    f0 := f2(y)
  end

Function f3(function f(y:integer):integer):Integer;
  begin
    f3 := f(x)+x
  end

Function f4(x:integer):Integer;
   ...

begin
  x := 1;
  f0(1)
end.
```

id	(saltos,offset)
w	(0,p#1)
y	(1,p#1)
x	(2,v#1)
z	(1,v#1)



... em Linguagens tipo C

- Acesso a identificadores compilado para acesso a endereços de memória na stack.
- Cada chamada de uma função ou procedimento gera um bloco na pilha com informação relavante:
  - Endereço de retorno
  - Link dinâmico (bloco da função que o chamou)
  - Memória para parâmetros/argumentos
  - Memória para variáveis locais
- Não é necessário nenhum link estático. [Porquê?]

... em Linguagens tipo ML

- Acesso a identificadores compilado para acesso a endereços de memória na stack.
- Cada chamada de uma função ou procedimento gera um bloco na pilha com informação relavante:
  - Memória para parâmetros/argumentos
  - Memória para variáveis locais
  - [Que mais?]
- Não é suficiente o link estático numa pilha simples. [Por

Funções de ordem superior... High-order functions

Porque as funções são valores de runtime (closures) e os stack frames não podem ser desempilhados.

Compilação de CALCF

- A compilação de CALCF pode ser caracterizada por uma função:

  comp: P × ENV → CodeSeq x CodeSeq list

  P = Fragmento de programa (aberto)

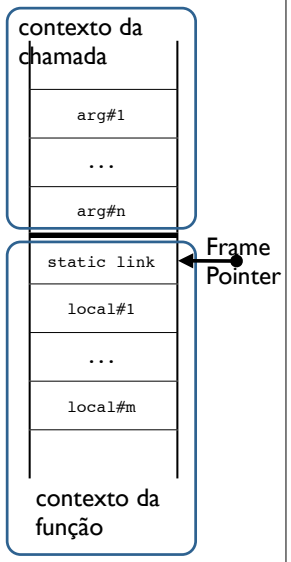
  ENV = Ambiente (função String → int)

O ambiente atribui a cada identificador uma posição na pilha de chamada (um número inteiro).

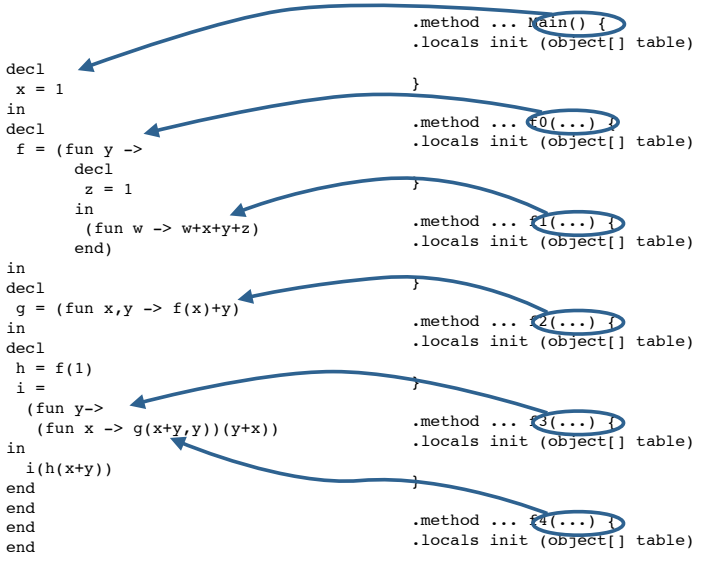
CodeSeq = Sequências de instruções

Compilação de CALCF (Ideias Gerais)

- Cada abstracção é mapeada numa função CIL com um registo de activação próprio onde são guardados os valores correspondentes aos argumentos e às declarações locais.
- O registo de activação contém um link estático para o registo de activação da última ocorrência na pilha da abstracção envolvente estaticamente no programa. Posição 0 do registo de activação.
- Os argumentos são passados através do registo de activação, que é passado pelo mecanismo normal da linguagem CIL. Os valores são colocados no registo de activação **antes** da chamada. Numa pilha fazem parte do registo de activação anterior e têm deslocamentos negativos em relação ao FramePointer.



Compilação de CALCF (mapeamento de



Compilação de CALCF

Os valores associados aos identificadores são guardados no âmbito local do método CIL correspondente. Ao separar o código em vários métodos não aninhados perde-se o acesso aos identificadores. As variáveis locais são apenas acessíveis dentro do método.

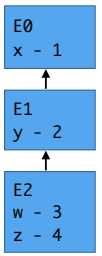
```
decl
  x = 1
  decl
    f = (fun y -> y+x)
  in
    decl
      g = (fun x -> f(x)+1)
      i = (fun y->(fun x -> x)(f(y)))
    in
      i(f(1))
    end
  end
end
```

```
.method ... static  ... void Main () {
  .locals (object[] table)
  ...
}
.method ... static ... object f0(object y) {
  ldloc table // x ????
  ...
}
.method ... static ... object f1(object x) {
  ldloc table // f ????
  ...
}
...
```


Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

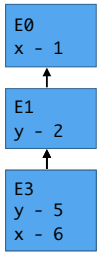
```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end

 .locals (object[] table)
      ...
 ldloc table
 ldc.i4 1
 ldelem.ref // x
```



Endereçamento em CALCI

Os identificadores são associados a endereços numa zona contígua de memória. Os endereços são atribuídos sequencialmente.

```
decl x = 1 in
  decl y = x + 2 in
    decl w = y z = 2*x+y in w+z+x end
    +
    decl y = x x = 3 in y+x end
  end
end

  .locals (object[] table)
  ...
```



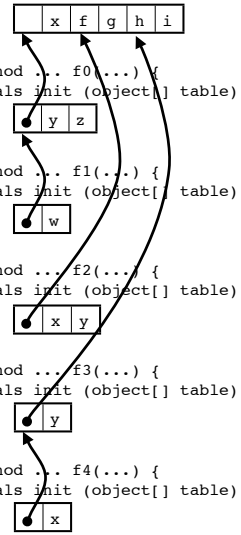
Compilação de CALCF (registos activação)

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```

A primeira posição (0) fica reservada para o static link

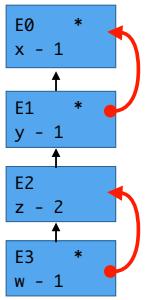
Os endereços de identificadores começam em 1

```
.method ... Main() {
.locals init (object[] table)
}

.method ... f0(...) {
.locals init (object[] table)
}

.method ... f1(...) {
.locals init (object[] table)
}

.method ... f2(...) {
.locals init (object[] table)
}

.method ... f3(...) {
.locals init (object[] table)
}

.method ... f4(...) {
.locals init (object[] table)
}
```



Compilação de CALCF (endereçamento)

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```



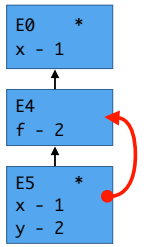
- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
w - (0,1)
x - (2,1)
y - (1,1)
z - (1,2)
```

* começa um novo registo de activação

Compilação de CALCF (endereçamento)

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```



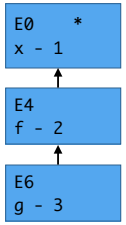
- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
f - (1,2)
x - (0,1)
y - (0,2)
```

* começa um novo registo de activação

Compilação de CALCF (endereçamento)

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```



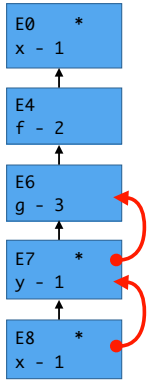
- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
f - (0,2)
```

```
* começa um novo registo de activação
```


Compilação de CALCF (endereçamento)

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```



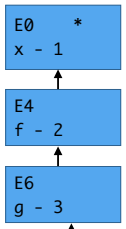
- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
g - (2,3)
x - (0,1)
y - (1,1)
```

```
* começa um novo registo de activação
```


Compilação de CALCF (endereçamento)

```
decl
 x = 1
in
decl
 f = (fun y ->
       decl
        z = 1
       in
        (fun w -> w+x+y+z)
       end)
in
decl
 g = (fun x,y -> f(x)+y)
in
decl
 h = f(1)
 i =
  (fun y->
   (fun x -> g(x+y,y))(y+x))
in
  i(h(x+1))
end
end
end
end
```



- O ambiente guarda o deslocamento dentro de um registo de activação.
- Quando se começa um novo registo de activação os endereços começam novamente em 1
- O endereço de um identificador depende do local do código onde está a ser consultado.
- Um endereço é um par (saltos, deslocamento)

```
i - (0,5)
h - (0,4)
x - (0,1)
```

```
* começa um novo registo de activação
```


Ambiente "mutável"

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável:

```
Environ BeginScope()
```

  - Cria um novo nível **vazio**, onde serão colocadas as ligações de um novo âmbito local.
  - Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível.

```
Environ EndScope()
```

  - Devolve o ambiente no estado anterior à última operação BeginScope().
  - Mas **não destrói** o último nível pois podem existir no contexto de execução fechos que o referem!

Ambiente "mutável"

- Para suportar o endereçamento usando vários registos de activação
- Os ambientes devem distinguir dois tipos de situações

`Environ BeginScope(boolean startsAR)`

  - cria um novo nível local vazio, onde serão colocadas as novas ligações.
  - Regista se o novo nível corresponde ao início de um novo registo de activação.
  - Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível.

`int getNumDecls()`

  - denota o número de declarações dentro do registo de activação corrente.

`Pair<Integer,Integer> find(String id)`

  - procura o identificador id no nível corrente e depois nos anteriores, o primeiro elemento do par denota o número de saltos entre registos de activação necessários, o segundo elemento denota o endereço guardado no ambiente.

Ambiente "mutável"

Atenção: Os endereços dentro de um registo de activação não podem ser reutilizados (da mesma maneira que os registos de activação não podem ser desempilhados). Sugestão de implementação da classe ambiente:

```
class Env {
   ...
   Env up;
   boolean startFrame;
   int count;

   void assoc(String id, int addr) {
          ...
          incDecls();
   }

   private void incDecls() {
          if(startFrame) count++;
          else up.incDecls();
   }

   int getNumDecls() {
          if(startFrame) return count;
          else return up.getNumDecls();
   }
}
```


Compilação de CALCF (Closures)

- As closures são valores de runtime que representam funções. Contêm uma referência para o código a executar e o acesso ao registo de activação da última instância da função que a envolve sintacticamente.
- Uma função pode produzir closures e por isso o seu registo de activação não pode ser destruído quando acaba a execução. Os registos de activação só são destruídos por "garbage collection".
- Nesse caso os registos de activação devem ser criados na heap e formam uma estrutura de dados chamada pilha esparguete (Spaghetti Stack).
- Implementação de closures em objectos no sistema de suporte à execução.
- Implementação de registos de activação em objectos no sistema de suporte à execução.

Compilação de CALCF (Closures)

- Uma closure é composta por uma referência para o código de uma abstração e uma referência ao seu static link (um registo de activação).

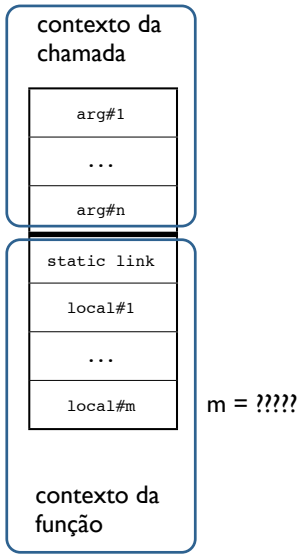
```
class Closure {
   StackFrame stackFrame;
   IntPtr ftn;

   Closure(StackFrame stackFrame, IntPtr ftn) {
       this.stackFrame = stackFrame;
       this.ftn = ftn;
   }
   StackFrame getSF() { return stackFrame; }
   IntPtr getFtn() { return ftn; }
}
```

- O registo de activação da closure é o registo de activação onde foi avaliada a definição da função ou procedimento. Neste ambiente de compilação é conhecido o nome da função CIL que lhe corresponde.

```
.locals init (object stackFrame)
...
ldloc stackFrame
ldftn object f0(object)
newobj instance void Closure::.ctor(class [StackFrame], native int)
```

Compilação de CALCF (passagem parâmetros)

- O registo de activação deve que ser criado e preenchido com os valores dos argumentos **antes** da chamada.
- No contexto da chamada sabemos calcular os argumentos e sabemos quantos são, mas...
- não sabemos quantas variáveis locais vão ser necessárias para a execução da função... (recorde-se que as funções são anónimas)
- logo, é necessária uma estrutura de dados mais dinâmica que um vector com um número fixo de posições.
- Suporte para a criação da memória para os argumentos e a memória para as variáveis locais em alturas diferentes (antes e depois da chamada da função).

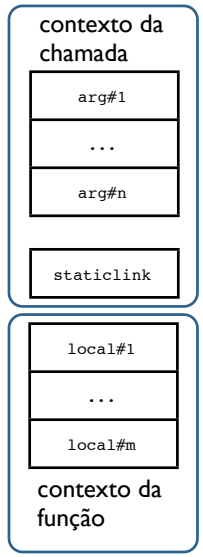


Compilação de CALCF (passagem parâmetros)

- Suporte para a criação da memória para os argumentos e a memória para as variáveis locais em alturas diferentes (antes e depois da chamada da função).

  Sugestão de implementação:

```
class StackFrame {
   object staticlink; // indice 0

   object[] params;   // 1 a nParams
   int nParams;

   object[] locals;     // ... > nParams+1

   StackFrame(object SL) {...}

   void initArgs(int nParams) {...}
   void initLocals(init nLocals) {...}

   object get(int i) {...}
   void set(int i, object o) {...}
}
```



Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

comp: P × ENV → CodeSeq x CodeSeq list

Dado um ambiente **env**

se *E* é da forma **num**(n):

```
s = < ldc.i4 n > @ < box int32 >

comp(E,env) ≜ (s, [ ] )
```

se *E* é da forma **add**(E',E''):

```
(s1, f1) = comp(E',env);
(s2, f2) = comp(E'',env);
s ≜ s1 @ < unbox int32 > @ < ldobj int32 > @
    s2 @ < unbox int32 > @ < ldobj int32 > @
    <add> @ < box int32 >

comp(E,env) ≜ (s, f1 @ f2 )
```


Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

comp: P × ENV → CodeSeq x CodeSeq list

Dado um ambiente **env**

se E é da forma **id**(s):

```
(jumps,offset) = env.find(s);
s = <ldloc stackframe>
@<ldc.i4 0>
@<callvirt instance object StackFrame::get(int32)>
... // n jumps
@<ldc.i4 0>
@<callvirt instance object StackFrame::get(int32)>
@<ldc.i4 offset>
@<callvirt instance object StackFrame::get(int32)>

comp(E, env) ≜ ( s, [ ] )
```

Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

comp: P × ENV → CodeSeq x CodeSeq list

Dado um ambiente **env**

se E é da forma **fun**(id,E): n = fresh_fun_identifier()

```
env = env.BeginScope(true) // true => starts a new Stackframe...
env.Assoc(id, env.getNumDecls());
( sE, f ) = comp(E,env);
env = env.EndScope();
 s =    <ldftn object fn(object)> @ <ldloc stackframe> @
        <newobj instance void Closure::.ctor(object,object)>
```

comp(E,*env*) ≜ `( s ,  [("fn",sE)] @ f )`

Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

comp: P × ENV → CodeSeq x CodeSeq list

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2  ) = comp(E1, env);
s = s1 @ ...
```

1. Construir um StackFrame com o static link da closure
2. Colocar os argumentos no topo da pilha e depois no StackFrame
3. Retirar a referência para a função da Closure que se encontra na pilha
4. Chamar a função passando o StackFrame já preenchido

comp(E,env) ≜ `(s, f1@f2 )`

Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

comp: P × ENV → CodeSeq x CodeSeq list

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2  ) = comp(E1, env);
s = s1 @
```

1. Construir um StackFrame com o static link da closure
   ```
   <dup> @
   <callvirt instance Closure::getSF()> @
   <newobj instance void StackFrame::.ctor(object)> @ ...
   ```
2. Colocar os argumentos no topo da pilha e depois no StackFrame
3. Retirar a referência para a função da Closure que se encontra na pilha
4. Chamar a função passando o StackFrame já preenchido

comp(E,env) ≜ `(s, f1@f2 )`

Compilador de CALCF

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem CALCF:

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2  ) = comp(E1, env);
s = s1 @
```

1. Construir um StackFrame com o static link da closure
   ```
   <dup> @
   <callvirt instance Closure::getSF()> @
   <newobj instance void StackFrame::.ctor(object)> @
   ```
2. Colocar os argumentos no topo da pilha e depois no StackFrame
   ```
   <dup> @ <ldc.i4 1> @ <callvirt void initArgs(int32)> @
   <dup> @ <ldc.i4 1> @ s2 @
   <callvirt void StackFrame::set(int32,object)> @ ...
   ```
3. Retirar a referência para a função da Closure que se encontra na pilha
4. Chamar a função passando o StackFrame já preenchido

comp(E,env) ≜ `(s, f1@f2 )`

Compilador de CALCF

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2  ) = comp(E1, env);
s = s1 @
```

1. Construir um StackFrame com o static link da closure
```
<dup> @
<callvirt instance Closure::getSF()> @
<newobj instance void StackFrame::.ctor(object)> @
```
2. Colocar os argumentos no topo da pilha e depois no StackFrame
```
<dup> @ <ldc.i4 1> @ <callvirt void initArgs(int32)> @
<dup> @ <ldc.i4 1> @ s2 @
<callvirt void StackFrame::set(int32,object)> @
```
3. Retirar a referência para a função da Closure que se encontra na pilha

3.1 trocar os 2 valores que estão no topo da pilha para que a closure fique acessível

3.2 obter o apontador para a função a partir da Closure.
```
<callvirt instance object Closure::getFtn()> @ ...
```
4. Chamar a função passando o StackFrame já preenchido

comp(E,env) ≜ `(s, f1@f2 )`

Compilador de CALCF

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2  ) = comp(E1, env);
s = s1 @
```

1. Construir um StackFrame com o static link da closure
```
<dup> @
<callvirt instance Closure::getSF()> @
<newobj instance void StackFrame::.ctor(object)> @
```
2. Colocar os argumentos no topo da pilha e depois no StackFrame
```
<dup> @ <ldc.i4 1> @ <callvirt void initArgs(int32)> @
<dup> @ <ldc.i4 1> @ s2 @
<callvirt void StackFrame::set(int32,object)> @
```
3. Retirar a referência para a função da Closure que se encontra na pilha

3.1 trocar os 2 valores que estão no topo da pilha para que a closure fique acessível
```
<stloc tmp> @
<stloc tmp2> @
<ldloc tmp> @
<ldloc tmp2> @
```
3.2 obter o apontador para a função a partir da Closure.
```
<callvirt instance object Closure::getFtn()> @ ...
```
4. Chamar a função passando o StackFrame já preenchido

comp(E,env) ≜ `(s, f1@f2 )`

Dado um ambiente **env**

se E é da forma **call**(E1,E2):

```
( s1, f1 ) = comp(E1, env);
( s2, f2  ) = comp(E1, env);
s = s1 @
```

1. Construir um StackFrame com o static link da closure
```
<dup> @
<callvirt instance Closure::getSF()> @
<newobj instance void StackFrame::.ctor(object)> @
```
2. Colocar os argumentos no topo da pilha e depois no StackFrame
```
<dup> @ <ldc.i4 1> @ <callvirt void initArgs(int32)> @
<dup> @ <ldc.i4 1> @ s2 @
<callvirt void StackFrame::set(int32,object)> @
```
3. Retirar a referência para a função da Closure que se encontra na pilha

3.1 trocar os 2 valores que estão no topo da pilha para que a closure fique acessível
```
<stloc tmp> @
<stloc tmp2> @
<ldloc tmp> @
<ldloc tmp2> @
```
3.2 obter o apontador para a função a partir da Closure.
```
<callvirt instance object Closure::getFtn()> @
```
4. Chamar a função passando o StackFrame já preenchido
```
<calli object(object)>
```

comp(E,env) ≜ `(s, f1@f2 )`

Compilação de CALCF (exemplo)

preâmbulo e estrutura da assembly

`(fun x -> decl y = 2 in x+y)(2)`

```
.assembly Calc {}
.assembly extern Runtime {}
.module Calc.exe
.method public static  hidebysig default void Main ()  cil managed
{
  .entrypoint
  .locals init (object stackframe, object tmp, object tmp2)
  ldnull
  newobj instance void [Runtime]StackFrame::.ctor(class [Runtime]StackFrame)
  stloc stackframe
  ... // closure, chamada & unbox
  call void class [mscorlib]System.Console::Write(int32)
  ret
}
.method public static  hidebysig default object f1(object) cil managed
{
  .locals init (object stackframe, object tmp, object tmp2)
  ldarg 0
  stloc stackframe
   ldloc stackframe
   ldc.i4 1
   callvirt instance void [Runtime]StackFrame::initLocals(int32)
  ... // corpo da função
  ret
}
```

Compilação de CALCF (exemplo)

definição da abstração

```
(fun x -> decl y = 2 in x+y)(2)
```

```
...
.method public static  hidebysig default void Main ()  cil managed
{
  .entrypoint
  .locals init (object stackframe, object tmp, object tmp2)
  ... // preambulo
  ldloc stackframe
  ldftn object f1(object)
  newobj instance void [Runtime]Closure::.ctor(class [Runtime]StackFrame,object)
  ... // chamada, unbox & print
}
.method public static  hidebysig default object f1(object) cil managed
{
  .locals init (object stackframe, object tmp, object tmp2)
  ... // preambulo, initLocals
  ldloc stackframe // decl y = ...
  ldc.i4 2
  ldc.i4 2
  box int32
  callvirt instance void [Runtime]StackFrame::set(int32,object)

  ldloc stackframe // x
  ldc.i4 1
  callvirt instance object [Runtime]StackFrame::get(int32)
  ... // unbox, y, unbox, add & box
  ret
}
```


Compilação de CALCF (exemplo)

chamada da função...

```
(fun x -> decl y = 2 in x+y)(2)
```

```
...
.method public static  hidebysig default void Main ()  cil managed
{
  .entrypoint
  .locals init (object stackframe, object tmp, object tmp2)
  ... // Closure no topo da pilha...
  dup
  callvirt instance class [Runtime]StackFrame [Runtime]Closure::getSF()
  newobj instance void [Runtime]StackFrame::.ctor(class [Runtime]StackFrame)
  dup
  ldc.i4 1
  callvirt instance void [Runtime]StackFrame::initArgs(int32)
  dup
  ldc.i4 1
  ldc.i4 2
  box int32
  callvirt instance void [Runtime]StackFrame::set(int32,object)
  stloc tmp
  stloc tmp2
  ldloc tmp
  ldloc tmp2
  callvirt instance object [Runtime]Closure::getFtn()
  calli object(object)
  ... // unbox & print
```


Definições Recursivas

- Na construção de declaração local de identificadores

  `decl id = E1 in E2 end`

  o nome `id` não é ligado às ocorrências do mesmo nome em E_1

- Por exemplo, na expressão

```
decl x=1 in
  decl x=x+1 in
    x+1
  end
end
```

  a ocorrência de x na inicialização do segundo `decl` está ligada à primeira declaração `x = 1`.

- A segunda definição de x não é “recursiva” !!

Definições Recursivas

- Uma definição recursiva é uma declaração onde o identificador declarado pode ocorrer dentro do corpo da definição:

```
declrec Sum = (fun x → if x=0 then 1
                       else x + Sum(x-1))
in
  Sum(10)
end
```

  Mais rigorosamente: numa declaração recursiva a ocorrência ligante de um identificador também liga as ocorrências livres do mesmo identificador na expressão de inicialização.

- Problema: como introduzir e como definir a semântica de definições recursivas (de funções) ?

A Linguagem RECF

- A linguagem RECF é a extensão da linguagem CALCF com expressões condicionais e definições recursivas.

num:	integer → RECF
var:	string → RECF
add:	RECF × RECF → RECF
...	
if:	RECF × RECF × RECF → RECF
declrec:	string × RECF × RECF → RECF
fun:	string × RECF → RECF
call:	RECF × RECF → RECF

– Nota: a construção decl não existe na linguagem RECF, mas pode ser codificada da forma apresentada atrás:

(decl x = E1 in E2) = (fun x -> E2)(E1)

A Linguagem RECF

- Exemplo de programa em RECF:

```
declrec
  fact = fun n → if n then n*fact(n-1) else 1 end
in
  fact(2)
end
```

- Abreviaturas:

$$\texttt{decl id = } E_1 \texttt{ in } E_2 \triangleq \texttt{call((fun id} \rightarrow E_2\texttt{),}E_1\texttt{)}$$

$$E_1(E_2) \triangleq \texttt{call}(E_1, E_2)$$

$$\texttt{if } E_1 \texttt{ then } E_1 \texttt{ else } E_3 \triangleq \texttt{if}(E_1, E_2, E_3)$$

A Linguagem RECF

- Exemplo de programa em RECF:

```
declrec
  fact = fun n → if n then n*fact(n-1) else 1 end
in
  fact(2)
end
```

- Qual o ambiente no qual a abstracção deve ser avaliada/definida? Note que:
  - fact é uma ocorrência livre na abstracção
  - O identificador fact deverá estar definido no ambiente activo no momento em que a abstracção for avaliada
  - Por outro lado, o valor a associar a fact nesse mesmo ambiente deverá ser a própria função.

A Linguagem RECF

- Exemplo de programa em RECF:

```
declrec
  fact = fun n → if n then n*fact(n-1) else 1 end
in
  fact(2)
end
```

- Qual o ambiente no qual a abstracção deve ser avaliada/definida?
  - As definições recursivas introduzem uma "circularidade" na construção do ambiente:
  - O ambiente E resultante da declaração de fact deverá ter uma ligação `fact → closure(n, if ... end , E)` que associe ao nome `fact` um fecho cuja componente ambiente é o próprio `E`
  - Em geral, um ambiente E deverá pode conter ligações entre identificadores e fechos com referências para (partes de) o próprio ambiente E

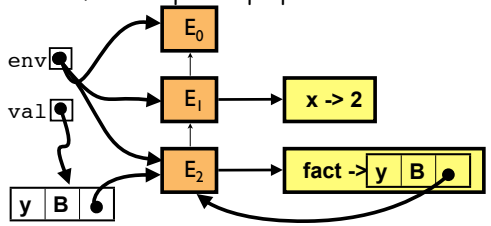
Ambiente Mutável (revisitado)

- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.
- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = BeginScope();
env.Assoc("x", 2);
env = env.BeginScope();
env.Assoc("fact", null);
val = closure("y", B,
env);
env.Update("fact", val);
```



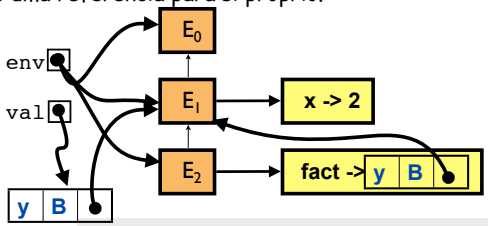
Ambiente Mutável (revisitado)

- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val.
- Se o valor `val` contiver uma referência para o ambiente `env`, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();
env = env.BeginScope();
env.Assoc("x", 2);
val = closure(n, B, env);
env.BeginScope();
env.Assoc("fact", val);
```



Aqui, durante a avaliação do corpo B, o ambiente não refere a definição de fact (apenas de x) …

Semântica de RECF

- A função semântica I de RECF:

  I : RECF × ENV → RESULT

  RECF = conjunto dos programas abertos
  ENV = conjunto dos ambientes válidos
  RESULT = conjunto dos significados

- Os resultados podem ser valores inteiros, fechos (uma abstracção + um ambiente), ou um erro.

  RESULT = integer ∪ closure ∪ { error }

Semântica de RECF

- Algoritmo eval para calcular a denotação de qualquer expressão E de RECF num ambiente env:

eval : RECF × ENV → RESULT

Dado um ambiente **env**

Se E é da forma **if**(E1, E2 , E3): c = **eval**(E1, *env*);
if c != 0 **then eval**(E, *env*) ≜ **eval**(E2, env)
else eval(E, *env*) ≜ **eval**(E3, env)

Se E é da forma **declrec**(id, E1, E2):
[*envloc* = *env*.BeginScope();
envloc.Assoc(id, **null**);
val = **eval**(E1, *envloc*);
envloc.Update(id, val);
v = **eval**(E2, *envloc*);
envloc.EndScope() **]**

eval(E, *env*) ≜ v

Auto-avaliação...

- Avalie o programa P escrito na linguagem **RECF** usando a semântica apresentada:

```
declrec
  fact = fun n → if n then n*fact(n-1) else 1 end
in
  fact(3)
end
```

E1 = [fact → clos(n, (if …) , E_1)]

eval(P, ∅) =

eval(fact(3), E1) =

Se E é da forma **declrec**(id, E1, E2):
[*envloc* = *env*.BeginScope();
envloc.Assoc(id, **null**);
val = **eval**(E1, *envloc*);
envloc.Update(id, val);
v = **eval**(E2, *envloc*);

Passagem de parâmetros

Passagem de parâmetros

- Recorde a semântica da chamada de função adoptada nas linguagens CALCF e RECF:

Se E é da forma **call**($\mathbf{E_1}$, $\mathbf{E_2}$) :
if closure(*id*, *envc*, **B**) = **eval**($\mathbf{E_1}$, *env*)
then [*envloc* = *envc*.BeginScope();
envloc.Assoc(*id*, **eval**($\mathbf{E_2}$, *env*));
eval(E,*env*) ≜ **eval**(**B**, *envloc*) **]**
else eval(E,*env*) ≜ error

- A expressão E2 que denota o argumento da função é avaliada antes da função denotada pela expressão E1 ser efectivamente aplicada.
- Qual o valor intuitivo/efectivo do programa seguinte?

```
declrec f = fun x → f(x) end in
     decl g = fun y → 1 end in g(f(1)) end end
```

Passagem de parâmetros por valor

- Qual o valor da expressão seguinte ?

```
declrec f = fun x → f(x) end in
   decl g = fun y → 1 end in g(f(1)) end end
```

- Valor de acordo com a semântica "declarativa":
  - `g` é a função constante que devolve sempre o valor `1` independentemente do valor do argumento.
  - Então, o valor `g(f(1))` deverá ser `1`.
- Valor determinado pela semântica "operacional":
  - A semântica apresentada não define valor para esta expressão, pois a avaliação de `f(1)` não termina!
  - A regra de passagem de argumentos utilizada é a "passagem por valor" (*call-by-value*)

Passagem de parâmetros por "nome"

- Qual o valor da expressão (`1` ou nenhum)?

```
declrec f = fun x → f(x) end in
   decl g = fun y → 1 end in g(f(1)) end end
```

- Existe um modo de avaliação que permite sempre encontrar o valor "declarativo" das expressões.
- Consiste em suspender a avaliação dos argumentos das funções até ao momento em que estes se tornam necessários.
- Corresponde em passar a expressão como argumento, em vez do seu resultado.
- A esta regra de avaliação de argumentos chama-se "passagem por nome" (*call-by-name*) (CBN)
- A estratégia de avaliação por omissão do Algol 60.

Resultados de RECF (CBN)

- Algoritmo eval para calcular a denotação de qualquer expressão E de RECF num ambiente env usando a passagem de parâmetros CBN:

$$eval : RECF \times ENV \rightarrow RESULT$$

- Construtores do tipo de dados RESULT

num:	**integer → RESULT**
closure:	**string × RECF × ENV → RESULT**
susp:	**RECF × ENV → RESULT**

- Uma suspensão representa uma expressão por avaliar, juntamente com o ambiente respectivo.
- É necessário guardar na suspensão, junto com cada expressão, o ambiente correcto, pois em geral a expressão contém identificadores livres.

Semântica de RECF (CBN)

- Algoritmo eval para calcular a denotação de qualquer expressão E de RECF num ambiente env usando a passagem de parâmetros CBN:

$$eval : RECF \times ENV \rightarrow RESULT$$

Se E é da forma **id**(s): val = *env*.Find(s)
  if val = **susp**(B, *envsusp*)
  then eval(E, *env*) ≜ **eval**(B, *envsusp*)
  else eval(E, *env*) ≜ val

Se E é da forma **fun**(id, B): **eval**(E, *env*) ≜ **closure**(ident, B, *env*)

Se E é da forma **call**(E1, E2): **if eval**(E1, env) = **closure**(id, B , *envloc*)
  then [*envloc* = *envloc*.BeginScope();
    envloc.Assoc(id, **susp**(E2, env));
    eval(E, *env*) ≜ **eval**(B, *envloc*) **]**
  else eval(E, *env*) ≜ error

Semântica de RECF (CBN)

- Qual o valor (CBN) da expressão P ?

```
declrec f = fun x → f(x) end in
   decl g = fun y → 1 end in g(f(1)) end end
```

A_1 =[f → closure(x, f(x), A_1)]

```
eval(P, ∅) =
eval(decl g = (fun y → 1) in g(f(1)), A1) =
eval(g(f(1)), [g → closure(y,1,A1), f → closure(x, f(x), A1)] ) =
eval(1, [y → susp(f(1),A2), g → closure(y,1,A1), f → ... ] ) = 1
```

A_2

Avaliação RECF (CBN)

- Qual o valor (CBN) da expressão P ?

```
decl y = 2 in decl f = (fun z → z+y) in f(2+y)
```

A_1 A_2

```
eval(P, ∅) =
eval(decl f = (fun z → z+y) in f(2+y), [y → 2] ) =
eval(f(2+y), [f → closure(z, z+y, A1), y → 2] ) =
eval(z+y, [z → susp(2+y, A2), f → closure(z, z+y, A1), y → 2] ) =
eval(z, [z → susp(2+y, A2),f → closure(z, z+y, A1), y → 2] )+2 =
eval(2+y, A2 )+2 = 6
```

- Note que a expressão argumento 2+y é avaliada apenas quando o valor do parâmetro z se torna necessário!

Passagem de parâmetros por "necessidade"

- Qual o valor da expressão?

```
decl x = var(0) in
  declrec f = fun x → x := !x+1 end in
    decl g = fun y → y+y+!x end in g(f(x)) end end
```

- Se se suspender a avaliação dos argumentos das funções até ao momento em que estes se tornam necessários passando a expressão como argumento, em vez do seu resultado, a utilização repetida dos parâmetros pode levar a resultados não esperados.
- Para avaliar só uma vez cada suspensão é preciso "guardar" o seu resultado após a primeira avaliação.
- A esta regra de avaliação de argumentos chama-se "passagem por necessidade" (***call-by-need***). Utilizada em linguagens com avaliação Lazy (e.g. Haskell)

Semântica de RECF (Lazy)

- Algoritmo eval para calcular a denotação de qualquer expressão E de RECF num ambiente env usando a passagem de parâmetros Lazy:

  eval : RECF × ENV → RESULT

- É necessário representar a suspensão por um valor mutável:

```
Se E é da forma var(id): val = env.Find(id)
                 if val = susp (B, envsusp)
                 then [ if val.isUnevaluated()
                      then val.set(eval(B, envsusp))
                      eval(E, env) ≜ val.getValue() ]
                 else eval(E, env) ≜ val
```

Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2010-2011

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Unidade 8: Tipos funcionais

- Os sistemas de tipos definem um tipo de análise estática sobre programas que evita um conjunto de erros de execução. Nesta unidade estudamos a tipificação de linguagens de programação com abstracção num cenário simplificado, o do cálculo lambda.

- Fragmento funcional de CALCF
- Cálculo Lambda
- Semântica por substituição
- Operador Y (não terminação e recursão)
- Booleanos e Naturais em Cálculo Lambda
- Tipificação do cálculo Lambda
- Sistema de tipos (Sistemas de regras)
- Correcção dos sistemas de tipos

A Linguagem CALCF (recap)

- Tipo de dados CALCF com os constructores:

num:	Integer → CALCF
add:	CALCF × CALCF → CALCF
mul:	CALCF × CALCF → CALCF
div:	CALCF × CALCF → CALCF
sub:	CALCF × CALCF → CALCF
id:	String → CALCF
decl:	String × CALCF × CALCF → CALCF

fragmento funcional da linguagem

O Cálculo Lambda (λ-Calculus)

- É uma linguagem funcional minimal, permitindo a definição de nomes locais e funções como entidades de "primeira classe".
- Inventada por Alonzo Church em 1936, é a base dos mecanismos de abstracção usados em todas as linguagens de programação
- Sintaxe do cálculo lambda:

$$\mathbf{E} ::= x \mid \lambda x.\mathbf{E} \mid \mathbf{E1\ E2}$$

- Sintaxe abstacta do cálculo lambda:

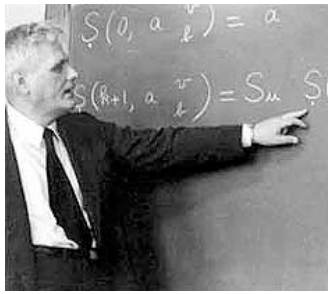
var:	string → LAMBDA	(x)
abs:	string × LAMBDA → LAMBDA	$(\lambda x.\mathbf{E})$
app:	LAMBDA × LAMBDA → LAMBDA	**(E1 E2)**

- As expressões da linguagem constroem-se **só** com variáveis (**var**), aplicação (**app**), e abstracção (**app**)

Alonzo Church (1903-1995)

Tese de Church:

"As funcões intuitivamente computáveis são exactamente as funções programáveis no cálculo lambda."



Church foi o orientador de tese de doutoramento de Alan Turing.

Juntos, demonstraram que o cálculo lambda tem o mesmo poder computacional que a máquina de Turing.

O Cálculo Lambda

- O cálculo lambda é computacionalmente completo: **qualquer algoritmo** pode ser programado no cálculo lambda (usando codificações mais ou menos complicadas)
- A ideia geral consiste em **traduzir** os mecanismos existentes nas linguagens de programação (tipos de dados, construções de controlo, definições recursivas) em expressões do cálculo lambda.
- Exemplo já conhecido (definições locais):

$$\text{decl } x = \mathbf{E}_1 \text{ in } \mathbf{E}_2 \text{ end} \triangleq (\lambda x.\mathbf{E}_2)\ \mathbf{E}_1$$

Semântica do Cálculo λ

- Algoritmo eval(E, env) para calcular o valor de uma expressão E de LAMBDA:

  eval : LAMBDA → LAMBDA

- Os resultados da avaliação de expressões LAMBDA são expressões lambda com certas características (formas "normais").
- Segue a regra de avaliação "Redução Topo-Esquerda"

eval(var(*ident*)) ≜ var(*ident*)

eval(abs(*ident*, B)) ≜ abs(*ident*, B)

eval(app(E_1, E_2)) ≜ v = **eval**(E_1);

if (v = **abs**(*ident*, B) **then eval(subst**(*ident*, B, E_2))

else app(v, E_2))

Definição da Função Subst

- A substituição é feita de maneira a evitar a captura de ocorrências

subst(s, **var**(*s*), E) ≜ E; /* caso x = s */

subst(s, **var**(x), E) ≜ **var**(x) /* caso x ≠ s */

subst(s, **abs**(s, M), E) ≜ **abs**(s, M) ; /* caso x = s */

subst(s, **abs**(x, M), E) ≜ **abs**(f, **subst**(s, **subst**(x, M, f), E));

subst(s, **app**(M, N), E) ≜ **app**(**subst**(s, M,E), **subst**(s, N,E));

A substituição por um identificador "fresco" garante que não são quebradas ligações do identificador x em M.

- Exemplos:

$\text{subst}(x, x + x, 2) = 2 + 2$

$\text{subst}(x, \lambda y.(x+y), z+3) = \lambda y.((z+3)+y)$

$\text{subst}(x, \lambda z.(x+z), z+3) = \lambda w.((z+3)+w)$

Redução Beta

- A semântica operacional do cálculo lambda pode ser definida usando uma única regra de avaliação, chamada redução beta, que procede à aplicação mais à esquerda no termo (*redex* - reducible expression).

$$\textbf{eval}(\ (\lambda x.\text{E1})\ \text{E2}) = \textbf{eval}(\ \textbf{subst}(x, \text{E1}, \text{E2})\ )$$

- Exemplos:

$\textbf{eval}(\ (\lambda x.\ x)\ y\ ) = \textbf{eval}(\ \textbf{subst}(x, x, y)\ ) = \textbf{eval}(\ y\ ) = y$

$\textbf{eval}(\ (\lambda x.\ x + x)\ 2) = \textbf{eval}(\ \textbf{subst}(x, x + x, 2)\ ) = \textbf{eval}(\ 2 + 2\ ) = 4$

$\textbf{eval}(\ \underline{(\lambda x.\lambda z.(x\ z))\ w}\ 2) =$

$\textbf{eval}(\textbf{subst}(x, \lambda z.(x\ z), w)\ 2) =$

$\textbf{eval}(\underline{(\lambda z.(w\ z))\ 2}) =$

$\textbf{eval}(\textbf{subst}(z, (w\ z), 2)) =$

$\textbf{eval}(w\ 2) = w\ 2$

Redução Beta

- A semântica operacional do cálculo lambda pode ser definida usando uma única regra de avaliação, chamada redução beta, que procede à aplicação mais à esquerda no termo (*redex* - reducible expression).

$$\textbf{eval}(\ (\lambda x.\text{E1})\ \text{E2}) = \textbf{eval}(\ \text{subst}(x, \text{E1}, \text{E2})\ )$$

- Exemplos:

$\textbf{eval}(\ \underline{(\lambda x.\lambda z.(x\ z))\ (\lambda y.(z\ y))}\ 2\ ) =$

$\textbf{eval}(\ \textbf{subst}(x, \lambda z.(x\ z), \lambda y.(z\ y))\ 2\ ) =$

$\textbf{eval}(\ (\lambda w.(\lambda y.(z\ y)\ w))\ 2) =$

$\textbf{eval}(\ \textbf{subst}(\ w, \lambda y.(z\ y)\ w, 2)\ ) =$

$\textbf{eval}(\ (\lambda y.(z\ y))\ 2\ ) =$

$\text{eval}(\ \textbf{subst}(\ y, z\ y, 2)\ ) = z\ 2$

Redução Beta

- A semântica operacional do cálculo lambda pode ser definida usando uma única regra de avaliação, chamada redução beta, que procede à aplicação mais à esquerda no termo (*redex* - reducible expression).

$$\textbf{eval}(\ (\lambda x.\text{E1})\ \text{E2}) = \textbf{eval}(\ \text{subst}(x, \text{E1}, \text{E2})\ )$$

- Exemplos:

$\textbf{eval}(\ \underline{(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ ((\lambda x.\ x)\ z)))}) =$

$\textbf{eval}(\ \textbf{subst}(x, (\lambda x.\ x)\ (\lambda z.\ ((\lambda x.\ x)\ z))\ ,\ x)\ ) =$

$\textbf{eval}(\ \underline{(\lambda x.\ x)\ (\lambda z.\ ((\lambda x.\ x)\ z))}) =$

$\textbf{eval}(\ \textbf{subst}(x, (\lambda z.\ ((\lambda x.\ x)\ z)), x)) =$

$\textbf{eval}(\ \lambda z.\ ((\lambda x.\ x)\ z)\ ) = \lambda z.\ ((\lambda x.\ x)\ z)$

Redução Beta

- A semântica operacional do cálculo lambda pode ser definida usando uma única regra de avaliação, chamada redução beta, que procede à aplicação mais à esquerda no termo (*redex* - reducible expression).

$$\textbf{eval}(\ (\lambda x.\text{E1})\ \text{E2}) = \textbf{eval}(\ \text{subst}(x, \text{E1}, \text{E2})\ )$$

- Exemplos:

$Id = \lambda x.\ x$

$\textbf{eval}(\ Id\ (Id\ (\lambda z.\ (Id\ z)))) =$

$\textbf{eval}(\ Id\ (\lambda z.\ (Id\ z))) =$

$\textbf{eval}(\ \lambda z.\ (Id\ z)\ ) = \lambda z.\ (Id\ z)$

Potencial de não-terminação

- A possibilidade de escrever programas que não terminam resulta da possibilidade de exprimir a auto-aplicação:
  uma função aplicada a ela própria.

Combinador $\Omega \triangleq (\lambda x.(x\ x))$

$$\begin{aligned}
&\textbf{eval}(\Omega\ \Omega) = \textbf{eval}(\ (\lambda x.(x\ x))\ \Omega\ ) = \\
&\quad \textbf{eval}(\textbf{subst}(x, (x\ x), \Omega\ )) = \\
&\quad \textbf{eval}(\Omega\ \Omega) = \textbf{eval}(\ (\lambda x.(x\ x))\ \Omega\ ) = \\
&\quad \textbf{eval}(\textbf{subst}(x, (x\ x), \Omega\ )) = \\
&\quad \textbf{eval}(\Omega\ \Omega) = \ \ldots
\end{aligned}$$

$$\textbf{eval}(\Omega\ \Omega) = \textbf{eval}(\Omega\ \Omega) = \textbf{eval}(\Omega\ \Omega) = \ldots$$

- Permite exprimir construções com potencial de computação Turing (ciclos while, recursividade, …)

Representação de Booleanos

- O tipo de dados Boolean consiste em dois valores diferentes true e false, que podem ser usados para seleccionar entre duas alternativas, usando a função

if C **then** T **else** E

que deve satisfazer as igualdades equações:

$$\begin{aligned}
&\textbf{eval}(\ \textbf{if true then}\ \mathrm{T}\ \textbf{else}\ \mathrm{E}\ ) &&= \textbf{eval}(\mathrm{T}) \\
&\textbf{eval}(\ \textbf{if false then}\ \mathrm{T}\ \textbf{else}\ \mathrm{E}\ ) &&= \textbf{eval}(\mathrm{E})
\end{aligned}$$

Podemos definir:

$$\begin{aligned}
\textbf{true} &\triangleq (\lambda x.\ \lambda y.\ x) \\
\textbf{false} &\triangleq (\lambda x.\ \lambda y.\ y) \\
\textbf{if} &\triangleq (\lambda x.\ \lambda y.\ \lambda z.\ (x\ y\ z))
\end{aligned}$$

Representação de Naturais

- O tipo de dados Nat consiste numa constante zero e numa função succ, que podem ser usados para construir qualquer valor natural.
- Para programar qualquer função aritmética, precisamos também de um predicado isZero? tal que

$$\begin{aligned}
&\textbf{eval}(\textbf{isZero?}(\textbf{zero})) &&= \textbf{true} \\
&\textbf{eval}(\textbf{isZero?}(\textbf{succ}(\mathrm{M}))) &&= \textbf{false}
\end{aligned}$$

Podemos definir (desafio ☺)

$$\begin{aligned}
\textbf{zero} &\triangleq \lambda s.\ \lambda z.\ z \\
\textbf{um} &\triangleq \lambda s.\ \lambda z.\ s\ (\ z\ ) \\
\textbf{succ} &\triangleq \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z) \\
\textbf{isZero?} &\triangleq \lambda n.\ n\ (\ \lambda x.\ \textbf{false}\ )\ \textbf{true} \\
\textbf{Add} &\triangleq \lambda n.\ \lambda m.\ \lambda s.\ \lambda z.\ n\ s\ (m\ s\ z)
\end{aligned}$$

Definições Recursivas

- A recursividade pode ser expressa internamente por meio de mais um combinador (**Rec**) que satisfaz a seguinte igualdade semântica:

eval (Rec E) = eval(E(Rec E))

Usando este operador, a declaração recursiva

declrec f = E in B end

pode ser expressa de forma não recursiva por

decl f = Rec (λf.E**) in B end**

- Define-se Rec em LAMBDA (Church-Kleene):

$$\mathrm{Rec} \triangleq \lambda f.\ ((\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$$

declrec fact = if n then n*fact(n-1) else 1 in fact 3

- *λf. ((λx. f (x x)) (λx. f (x x))) (λf. if (n) n*f(n-1) else 1)*
- *((λx. (λf. if (n) n* f(n-1) else 1)(x x)) (λx. (λf. if (n) n*f(n-1) else 1) (x x)))*
- *(λf. if (n) n* f(n-1) else 1)((λx. (λf. if (n) n* f(n-1) else 1) (x x))) ...)*
- *...*
- *(if (n) n*(if (n-1) ((n-1)*if (n-2) ((n-2)*if(n-3) ... else 1 ...)*

Semântica do Cálculo λ (CBV)

- Algoritmo eval(E, env) para calcular o valor de uma expressão E de LAMBDA usando a estratégia *call-by-value*:

$$eval : LAMBDA \times ENV \rightarrow RESULT$$

eval(**var**(id), *env*) : *env*.Find(id)

eval(**abs**(id, B), *env*) : **closure**(id, *env* , B)

eval(**app**(E1, E2) , *env*) : **if eval**(E1, env) = **closure**(id, *envclos*, B)

then [*envloc* = *envclos*.BeginScope();
envloc.Assoc(id, **eval**(E2, *env*)) ;
eval(B, *envloc*)]

else error

Tipos para o Cálculo λ+int

- Algoritmo typchk para calcular o tipo de uma expressão qualquer da linguagem LAMBDA:

$$typchk : LAMBDA \times ENV \rightarrow TYPE$$

$$TYPE \triangleq \{ int, Fun[TYPE,TYPE], none \}$$

- int é o tipo dos valores inteiros.
- Fun[T1, T2] é o tipo das funções que, dado um argumento de tipo T1, são garantidas devolver um resultado de tipo T2.
- none é o "tipo" das expressões indefinidas.

Podemos abreviar Fun[T1, T2] por (T1→ T2). Por exemplo:

(int → int) é o tipo da função sucessor succ.

((int → int) → int) é o tipo da função (λf.succ(f (2))).

Cálculo λ+int Tipificado

- Para obter um algoritmo de tipificação simples, é necessário alterar a sintaxe do cálculo lambda, etiquetando os parâmetros das funções com tipos.

var:	string → LAMBDA
abs:	string × TYPE × LAMBDA → LAMBDA
app:	LAMBDA × LAMBDA → LAMBDA

- Exemplos:

$$\lambda x{:}\text{int} \,.\, x$$

$$\lambda y{:}\text{int} \,.\, \lambda f{:}\text{int} \rightarrow \text{int} \,.\, f(y)$$

Tipificação do Cálculo λ+int

- Algoritmo typchk para calcular o tipo de uma expressão qualquer da linguagem LAMBDA:

typechk : LAMBDA × ENV → TYPE

typchk(num(*n*) , *env*) ≜ **int** ;

typchk(id(s) , *env*) ≜ *env*.Find(s) ;

poderíamos ainda introduzir outros literais e tipos básicos:

typchk(**true** , *env*) ≜ **bool** ;

typchk(**false** , *env*) ≜ **bool** ;

Tipificação do Cálculo λ+int

- Algoritmo typchk para calcular o tipo de uma expressão qualquer da linguagem LAMBDA:

typechk : LAMBDA × ENV → TYPE

typchk(**abs**(*id*, T, B) , *env*) ≜ *envloc* = *envc*.BeginScope();
envloc.Assoc(*id*, T) ;
T1 = **typchk**(B, *envloc*);
(T → T1)

typchk(**app**(M, N) , *env*) ≜ *Tm* = **typchk**(M, *env*);
Tn = **typchk**(N, *env*);
if *Tm* = (Tn → A) **then** A **else** none

Regras de Tipificação

- O algoritmo de tipificação pode ser expresso por um conjunto de regras de inferência usadas para derivar asserções de tipificação com a forma:

Env ⊢ Expr : Type (para expressões)

Env ⊢ Stmt **ok** (para comandos)

- *Env* representa o ambiente e tem a forma de uma lista de declarações: Env = $\mathbf{id}_0$: Type_0, …, $\mathbf{id}_n$: Type_n
- *Expr / Stmt* é uma expressão da **sintaxe abstracta**
- *Type* é o **tipo** atribuído à expressão Expr.
- Os comandos não denotam valores e por isso não têm tipo. A asserção apenas indica um comando está bem formado (tipificado)

Regras de Tipificação

- Exemplos de asserções de tipificação válidas:

$\varnothing \vdash 1 : \text{int}$

$s: \text{int} \vdash 1 + s : \text{int}$

$x: \text{int}, y : \text{int} \vdash x + y > 1 : \text{bool}$

$x: \text{Ref[int]} \vdash \mathbf{while}\ (!x<0)\ \mathbf{do}\ x:=!x+1\ \mathbf{end}\ \ \mathbf{ok}$

$y: \text{int} \vdash (\mathbf{fun}\ x: \text{int} \rightarrow y + x\ )\ \mathbf{:}\ \text{int} \rightarrow \text{int}$

- Exemplos de asserções de tipificação inválidas:

$s: \text{bool} \vdash 1 + s : \text{int}$

$x: \text{int}, y : \text{int} \vdash x + y > 1 : \text{int}$

$x: \text{Ref[bool]} \vdash \mathbf{while}\ (!x<0)\ \mathbf{do}\ x:=!x+1\ \mathbf{end}\ \ \mathbf{ok}$

$y: \text{int} \vdash (\mathbf{fun}\ x: \text{bool} \rightarrow y + x\ )\ \mathbf{:}\ \text{bool} \rightarrow \text{int}$

Regras de Tipificação

- Uma regra de tipificação é uma regra da forma

$$\frac{P1 \cdots Pn}{C}$$

- Num sistema de regras pode-se derivar ***C*** (***C*** é válida) se for possível derivar as premissas ***P1*** ··· ***Pn*** (***P1*** ··· ***Pn*** são válidas).

- No caso dos sistemas de tipos as premissas e a conclusão são asserções de tipificação.

$$\frac{Env_0 \vdash \text{Expr}_0\text{:Type}_0 \quad \cdots \quad Env_n \vdash \text{Expr}_n\text{:Type}_n}{Env \vdash \text{Expr :Type}}$$

- Zero ou mais premissas, uma só conclusão. Um axioma é uma regra sem premissas.

Sistema de Tipos para LAMBDA

- Regras para o Cálculo Lambda tipificado

R1 (axioma) $Env,\ x{:}T,\ Env' \vdash x : T$

R2 (abstracção)
$$\frac{Env, x{:}A \vdash M : B}{Env \vdash (\lambda x{:}A.M) : (A \to B)}$$

R3 (aplicação)
$$\frac{Env \vdash M : (A \to B) \qquad Env \vdash N : A}{Env \vdash (M\ N) : B}$$

Sistema de Tipos para LAMBDA

- Derivação da asserção z: int, f: int → int ⊢(λx:int.(f x)) z: int

R3
$$\frac{}{z{:}int,\ f{:}int \to int \vdash (\lambda x{:}int.(f\ x))\ z : int}$$

Sistema de Tipos para LAMBDA

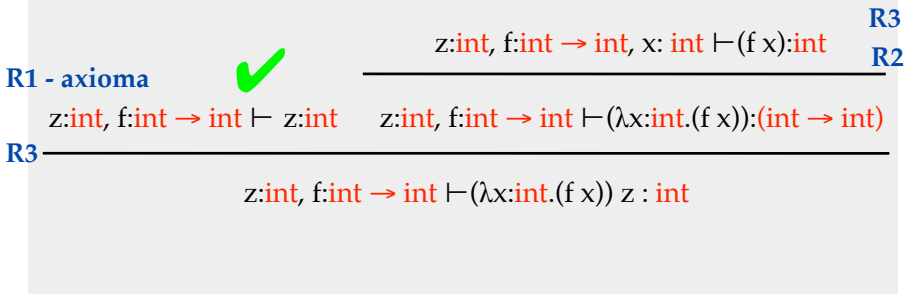
- Derivação da asserção z: int, f: int → int ⊢(λx:int.(f x)) z: int

R1 - axioma ✔ **R2**

R3
$$\frac{z{:}int,\ f{:}int \to int \vdash z{:}int \qquad z{:}int,\ f{:}int \to int \vdash (\lambda x{:}int.(f\ x)){:}(int \to int)}{z{:}int,\ f{:}int \to int \vdash (\lambda x{:}int.(f\ x))\ z : int}$$

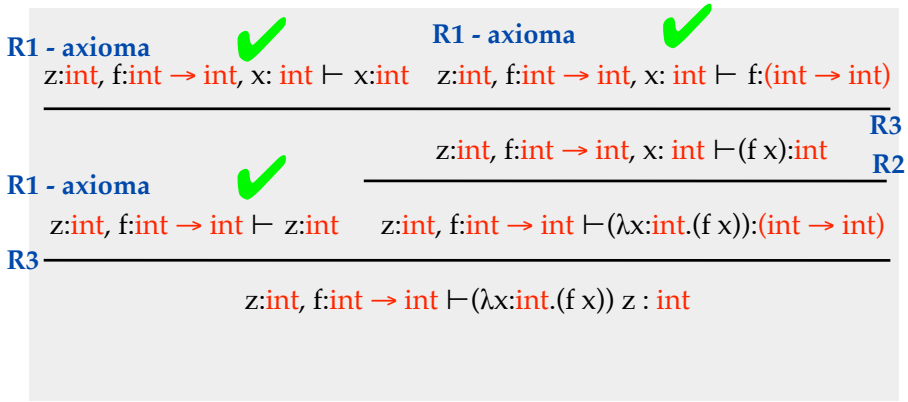
Sistema de Tipos para LAMBDA

- Derivação da asserção z: int, f: int → int ⊢(λx:int.(f x)) z: int



Sistema de Tipos para LAMBDA

- Derivação da asserção z: int, f: int → int ⊢(λx:int.(f x)) z: int



Correcção da Tipificação

Teorema: Para todo o programa P da linguagem LAMBDA, se

$$\varnothing \vdash P : \mathcal{T}$$

é derivável, então **eval**(P, ∅) = v ∈ $\mathcal{T}$ \ { none }

- Ou seja, se P está bem tipificado, então a sua execução termina sempre, e sem erros.

  Well-typed programs don´t go wrong (Robin Milner)

- Como todos os seus programas terminam, o cálculo lambda tipificado não é Turing-completo.
- Impossível tipificar a auto-aplicação: não é possível em particular representar o combinador Rec.