Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 7 - Compilação de linguagens imperativas, modos de passagem de parâmetros

terminamos a fase de **análise**

$$\boxed{\text{fonte}} \rightarrow \cdots \rightarrow \boxed{\text{sintaxe abstracta}}$$

vamos agora considerar a fase de **síntese**

$$\boxed{\text{sintaxe abstracta}} \rightarrow \cdots \rightarrow \boxed{\text{código máquina}}$$

estratégia de avaliação e de passagem de parâmetro

numa **declaração** de função

```
function f(x1, ..., xn) =
  ...
```

as variáveis `x1,...,xn` (introduzidas localmente na função `f`) são designadas de **parâmetros formais** de `f`

e na **chamada** desta função

```
f(e1, ..., en)
```

as expressões `e1,...,en` são designadas **parâmetros efectivos** de `f`

numa linguagem que contempla modificações *in-place*, uma atribuição

```
e1 := e2
```

modifica o local de memória designada pela expressão `e1`

a expressão `e1` é limitada na sua expressão nalgumas construções (as que representam locais memória que podem ser alterados)
isto porque atribuições como

```
42 := 17
true := false
```

não fazem em geral sentido

fala-se de **valor esquerdo/left value** para designar as expressões legais/autorizadas à esquerda de uma atribuição

a estratégia de avaliação de uma linguagem de programação estabelece em que ordem os cálculos são realizados

podemos definí-la a custa de uma definição semântica formal da linguagem - como vimos na aula 2

o compilador deve respeitar a estratégia de avaliação estabelecida

em particular, a estratégia de avaliação **pode** especificar

- em que momento os parâmetros efectivos de uma chamada são avaliados
- a ordem da avaliação dos operandos e dos parâmetros efectivos

certos aspectos da avaliação podem no entanto ficar **sem especificação**

isto deixa flexibildade e liberdade ao (designer do) compilador, em especial para efectar optimizações (por exemplo, deixando-o escolher a ordem dos cálculos como lhe será mais convenientes)

distinguimos em particular

- a **avaliação estrita** : os operandos / parâmetros efectivos são avaliados **antes** das operação / chamada

  exemplos : C, C++, Java, OCaml, Python, mini-ML das aulas anteriores

- a **avaliação preguiçosa** : os operandos / parâmetros efectivos só são avaliados quando são necessários

  exemplos : Haskell, Clojure
  mas também conectivas lógicas && e || da maioria das linguagens

uma linguagem imperativa adota sistematicamente uma avaliação estrita para garantir uma sequência previsível dos seus efeitos laterais que corresponde ao texto do código

por exemplo, o programa OCaml

```
let r = ref 0
let id x = r := !r + x; x
let f x y = !r
let () = print_int (f (id 40) (id 2))
```

mostra 42 porque os dois argumentos de `f` foram avaliados

a não terminação tem também efeito

assim, o programa

```
let rec loop () = loop ()
let f x y = x + 1
let v = f 41 (loop ())
```

não termina apesar do argumento `y` não ser utilizado.

uma linguagem puramente aplicativa, em contrapartida, pode adoptar ua estratégia de avaliação da sua escolha, porque uma expressão terá sempre o mesmo valor independentemente desta (fala-se de **transparência referencial**).

em particular, pode fazer a escolha de uma avaliação preguiçosa

o programa Haskell

```
loop () = loop ()
f x y = x
main = putChar (f 'a' (loop ()))
```

termina (após ter mostrado a)

a semântica detalha igualmente o **modo de passagem** dos parâmetros aquando de uma chamada

distinguimos em particular

- a **chamada por valor** (*call by value)*
- a **chamada por referência** (*call by reference*)
- a **chamada por nome** (*call by name*)
- a **chamada por necessidade** (*call by need*)

(fala-se às vezes também de **passagem** por valor, referência, etc.)

novas variáveis representando os parâmetros formais recebem os **valores** dos parâmetros efectivos

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)    // mostra 41
```

os parâmetros formais designam os **mesmos** *valores esquerdos* que os parâmetros efectivos

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)    // mostra 42
```

os parâmetros formais são **substituídos** pelo parâmetros efectivos, textualmente, e assim só são avaliados se necessário

```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0)) // mostra 25
  // 1+2 é avaliada duas vezes
  // 2+2 é avaliada duas vezes
  // 1/0 nunca é avaliada
```

os parâmetros efectivos só são avaliados se são necessários na avaliação do (sub-)programa em causa,
mas **no máximo uma vez**

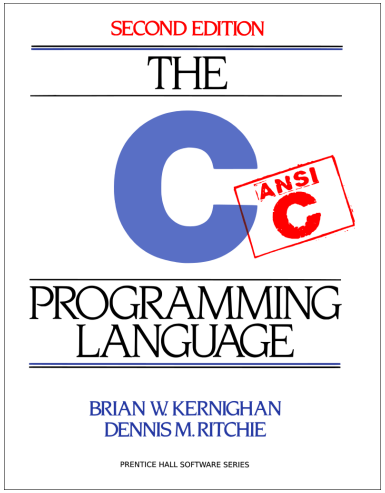
```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0)) // mostra 25
  // 1+2 é avaliada uma vez
  // 2+2 é avaliada uma vez
  // 1/0 nunca é avaliada
```

We need to talk about C

a linguagem C é uma linguagem imperativa de relativamente baixo nível, em parte porque a noção de apontador, em particular de aritmética de apontador, é explícita

podemos também considerá-lo como uma linguagem assembly de alto nível.

um livro sempre actual :
The C Programming Language
de Brian Kernighan e Dennis Ritchie



a linguagem está provida de uma estratégia de avaliação estrita com chamada **por valor**

a ordem de avaliação não está especificada

- encontramos os tipos de base tais como `char`, `int`, `float`, etc. (mas não há booleanos)

- um tipo τ`*` dos apontadores para valores de tipo τ

  se p é um apontador de tipo τ`*`, então `*`p designa um valor apontado por p, de tipo τ

  se e é um valor de tipo τ, então `&`e é um apontador para a localização memória correspondente, de tipo τ`*`

- registos, designados de *estructuras*, tais que

```
struct L { int head; struct L *next; };
```

  se e tem o tipo `struct L`, designamos por `e.head` o acesso ao campo `head`

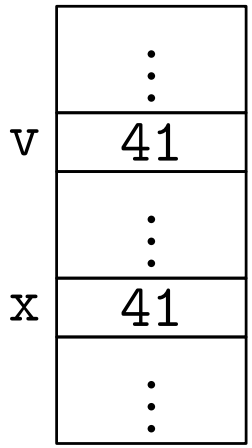
valores esquerdos de C

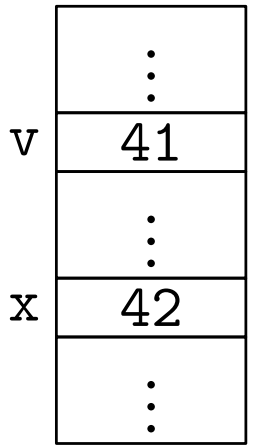
em C, um valor esquerdo é da forma

- `x`, uma variável
- `*e`, a dereferenciação de um apontador
- `e.x`, o acesso a um campo de estrutura,
  se `e` é igualmente em si um valor esquerdo

- `t[e]`, que não é outra coisa senão `*(t+e)`
- `e->x`, que não é outra coisa senão `(*e).x`

```
void f(int x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v continua com 41
}
```

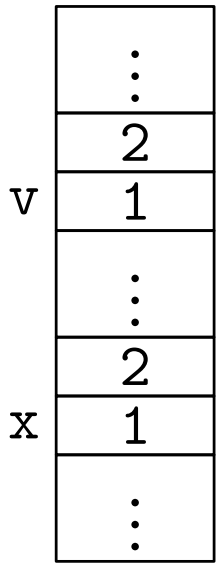


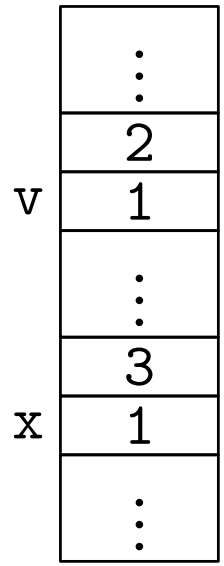


a chamada por valor implica que as estruturas são **copiadas** quando são passadas em parâmetro ou retornadas

as estruturas são igualmente copiadas aquando das atribuições de estruturas, *i.e.* nas atribuições da forma `x = y`, onde `x` e `y` tem ambas o tipo `struct S`

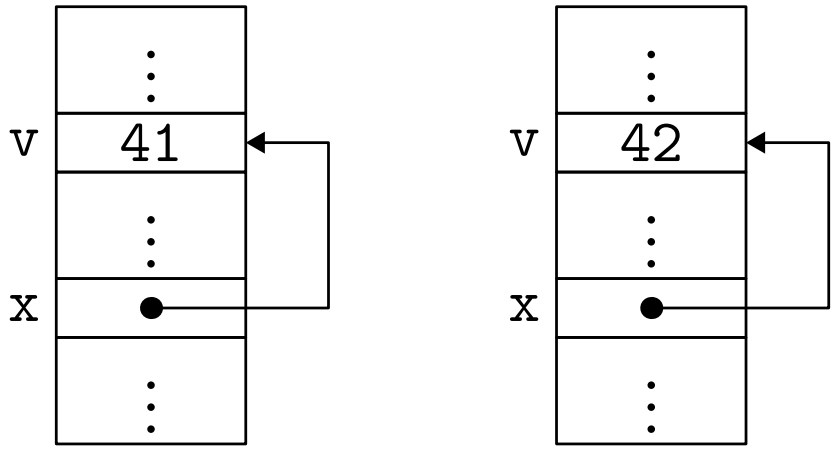
```c
struct S { int a; int b; };

void f(struct S x) {
  x.b = x.b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b ainda vale  2
}
```





podemos **simular** uma passagem por referência pela passagem explícita de um apontador

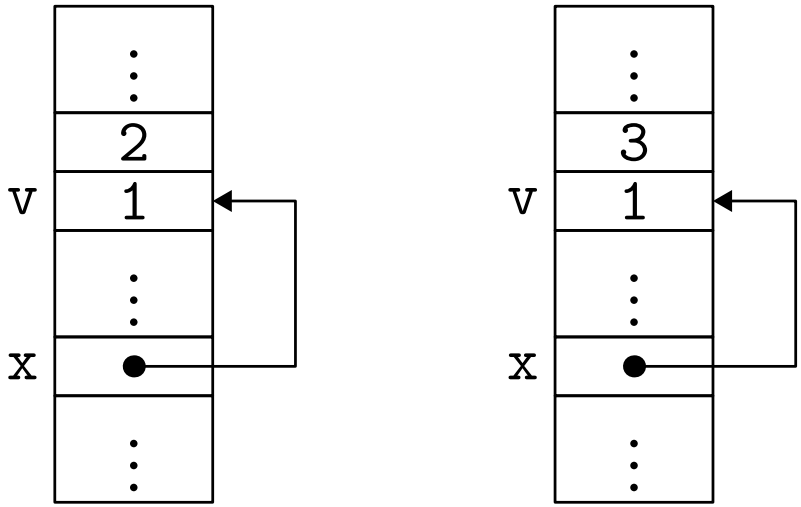
```
void incr(int *x) {
  *x = *x + 1;
}

int main() {
  int v = 41;
  incr(&v);
  // v vale agora 42
}
```



mas isso não é nada mais do que uma passagem de apontador **por valor**

para evitar o custo das cópias, a passagem das estruturas faz-se mais frequentemente pelos seus apontadores

```
struct S { int a; int b; };

void f(struct S *x) {
  x->b = x->b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(&v);
  // v.b vale agora 3
}
```



a manipulação explícita dos apontadores pode ser perigosa

consideremos o programa

```
int* p() {
  int x;
  ...
  return &x;
}
```

este devolve um apontador que corresponde a um endereço na pilha que acaba oportunamente de desaparecer (este aponta, mais precisamente, para a tabela de activação de `p`), e que será muito provavelmente reutilizado por uma outra tabela de activação

fala-se de referência fantasma (*dangling reference*)

podemos declarar um vector da seguinte forma :

```
int t[10];
```

a notação `t[i]` é açúcar sintáctico para `*(t+i)` onde

- `t` designa um apontador para o início de uma zona contendo 10 inteiros
- `+` designa uma operação de *aritmética de apontador* (que consiste em acrescentar a `t` a quantidade `4i` no caso de um vector de inteiros de 32 bits)

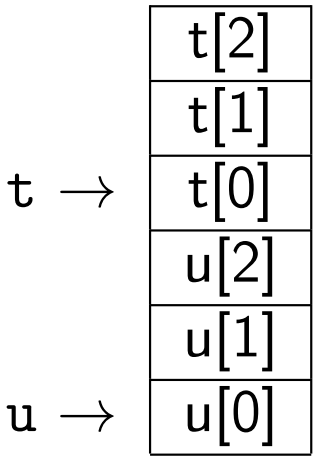
o primeiro elemento de um vector é então `t[0]` isto é `*t`

quando passamos um vector em parâmetro, só passamos o apontador (por valor, sempre)

não podemos atribuir vectores, somente apontadores

assim, não podemos escrever

```
void p() {
  int t[3];
  int u[3];
  t = u;
}
```



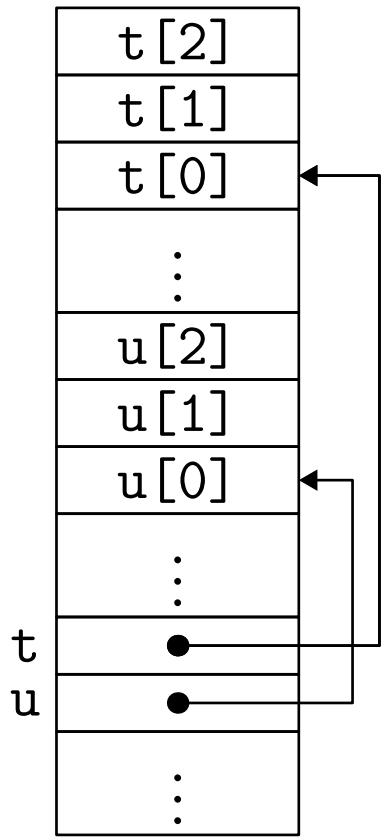
porque `t` e `u` são vectores (alocados na pilha) e a atribuição de vectores não é autorizada

por outro lado, podemos escrever

```
void q(int t[3], int u[3]) {
  t = u;
}
```

porque é precisamente a mesma coisa que

```
void q(int *t, int *u) {
  t = u;
}
```



e a atribuição de apontadores esta autorizada

```
/*    O que acontece aqui ?  */
void p() {
  int t[3];
  int *u = t;
}
```

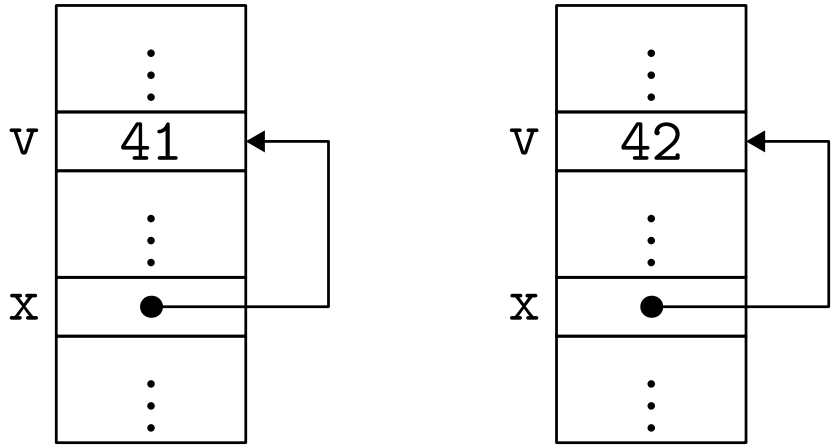
We need to talk about C++

em C++, encontramos (entre outros) os tipos e as construções de C, com uma estratégia de avaliação estrita

o modo de passagem é o modo **por valor** por omissão

mas encontramos também a passagem **por referência** indicada pelo símbolo & ao nível do argumento formal

```
void f(int &x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v vale agora 42
}
```

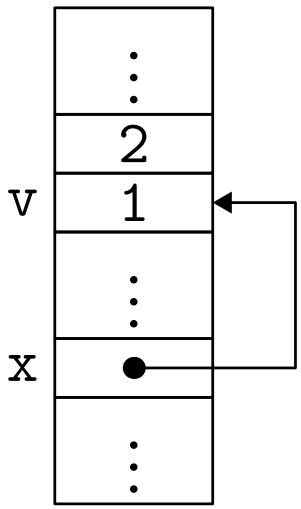


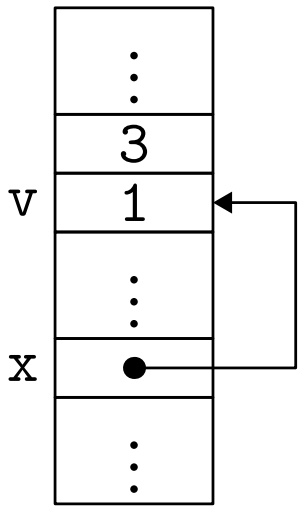
em particular, é o compilador que

- tomou nota do endereço de `v` no momento da chamada
- dereferenciou o endereço de `x` dentro da função `f`

podemos passar por uma estrutura por referência

```
struct S { int a; int b; };

void f(struct S &x) {
  x.b = x.b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b tem agora por valor 3
}
```





podemos passar um apontador por referência

por exemplo para juntar um elemento numa árvore

```
struct Node { int elt; Node *left, *right; };

void add(Node* &t, int x) {
  if       (t == NULL ) t = create(NULL, x, NULL);
  else if (x < t->elt) add(t->left,  x);
  else if (x > t->elt) add(t->right, x);
}
```

Let's talk about OCaml

OCaml está munido de uma estratégia de avaliação estrita com chamada **por valor**

a ordem de avaliação não está especificada

um valor é

- ou de um tipo primitivo (booleano, caracter, inteiro máquina, etc.)
- ou um apontador para um bloco de memória (vector, estrutura/registo, constructor não constante, etc.) alocados na *heap* em geral

São valores esquerdos, elementos dos vectores

```
a.(2) <- true
```

e os campos mutáveis dos registos

```
x.idade <- 42
```

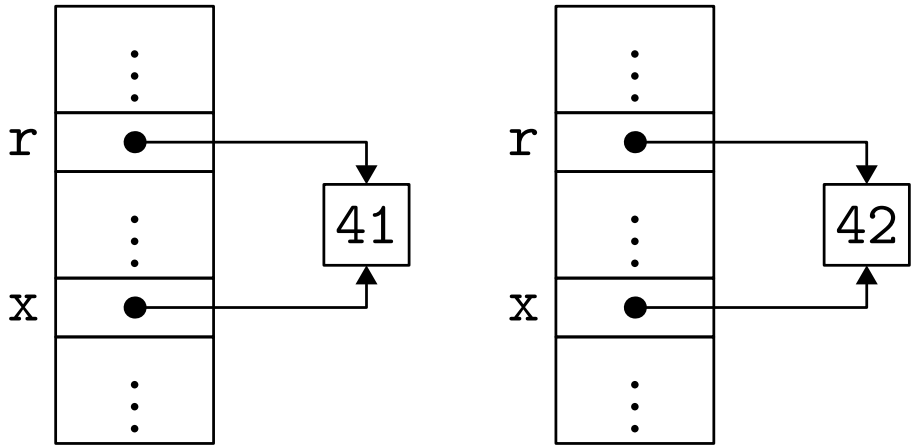
lembrete : uma referência é uma estrutura

```
type 'a ref = { mutable contents: 'a }
```

e as operações `:=` e `!` estão definidos por

```
let (!)  r   = r.contents
let (:=) r v = r.contents <- v
```

```
let f x =
  x := !x + 1

let main () =
  let r = ref 41 in
  f r
  (* !r vale agora 42 *)
```



permanece uma passagem por valor,
dum valor que é um apontador (para um valor mutável)

podemos **simular o *call-by-name*** em OCaml, substituindo os argumento em causa por funções

assim a função

```
let f x y =
  if x = 0 then 42 else y + 1
```

pode ser reescrita em

```
let f x y =
  if x () = 0 then 42 else y () + 1
```

e chamada da seguinte forma

```
let v = f (fun () -> 0) (fun () -> failwith "oops!")
```

de forma mais súbtil podemos igualmente **simular o *call-by-need*** em OCaml

começamos or introduzir um tipo para representar os cálculos preguiçosos

```
type 'a value = Value  of 'a
              | Frozen of (unit -> 'a)

type 'a by_need = 'a value ref
```

e uma função de avaliação destes cálculos, se estes já não se encontram realizados

```
let force l = match !l with
  | Value  v -> v
  | Frozen f -> let v = f () in l := Value v; v
```

definimos então a função `f` da seguinte forma

```
let f x y =
  if force x = 0 then 42 else force y + 1
```

e utilizamo-la assim

```
let v = f (ref (Frozen (fun () -> 0)))
          (ref (Frozen (fun () -> failwith "oops")))
```

nota : a construção `lazy` de Ocaml faz algo de semelhante,
mas de forma ainda mais súbtil

compilação de um fragmento de Pascal

porque encontramos ali simultaneamente

- funções aninhadas
- passagem por valor e por referência

consideramos o fragmento de Pascal seguinte

$$
\begin{array}{rcl}
E & \to & n \\
& | & x \\
& | & E \texttt{ + } E \mid E \texttt{ - } E \\
& | & E \texttt{ * } E \mid E \texttt{ / } E \\
& | & \texttt{ - } E
\end{array}
$$

$$
\begin{array}{rcl}
C & \to & E \texttt{ = } E \mid E \texttt{ <> } E \\
& | & E \texttt{ < } E \mid E \texttt{ <= } E \mid E \texttt{ > } E \mid E \texttt{ >= } E \\
& | & C \texttt{ and } C \\
& | & C \texttt{ or } C \\
& | & \texttt{not } C
\end{array}
$$

$$
\begin{array}{rcl}
S & \to & x \texttt{:= } E \\
& | & \texttt{if } C \texttt{ then } S \\
& | & \texttt{if } C \texttt{ then } S \texttt{ else } S \\
& | & \texttt{while } C \texttt{ do } S \\
& | & p(E, \ldots, E) \\
& | & B
\end{array}
$$

$$
\begin{array}{rcl}
D & \to & \texttt{var } x, \ldots, x\texttt{: integer;} \\
& | & \texttt{procedure } p(F; \ldots; F)\texttt{;} \\
& & \quad D \ldots D\ B\texttt{;}
\end{array}
$$

$$
\begin{array}{rcl}
F & \to & x\texttt{: integer} \\
& | & \texttt{var } x\texttt{: integer}
\end{array}
$$

$$
B \to \texttt{begin } S; \ldots; S \texttt{ end}
$$

$$
P \to \texttt{program } x\texttt{; } D \ldots D\ B\texttt{.}
$$

```
program fact;

var f : integer;

procedure fact(n : integer);
begin
  f := 1;
  while n > 1 do begin
    f := n * f;
    n := n - 1
  end
end;

begin
  fact(10);
  writeln(f)   { mostra 3628800 }
end.
```

```
program fib;
var f : integer;

procedure fib(n : integer);
   procedure soma();
   var tmp : integer;
   begin fib(n-2); tmp := f;
         fib(n-1); f := f + tmp end;
begin
   if n <= 1 then f := n else soma()
end;

begin fib(10); writeln(f) end.  { mostra 55 }
```

exemplo 3

```
program syracuse;

procedure syracuse(max : integer);
var i : integer;
   procedure length();
   var v,j : integer;
      procedure step();
      begin
        v := v+1; if j = 2*(j/2) then j := j/2 else j := 3*j+1
      end;
   begin
     v := 0; j := i; while j <> 1 do step(); writeln(v)
   end;
begin
   i := 1;
   while i <= max do begin length(); i := i+1 end
end;

begin syracuse(100) end.  { mostra 0 1 7 2 5 ... }
```

os procedimentos podendo ficar aninhados, introduzimos algumas definições

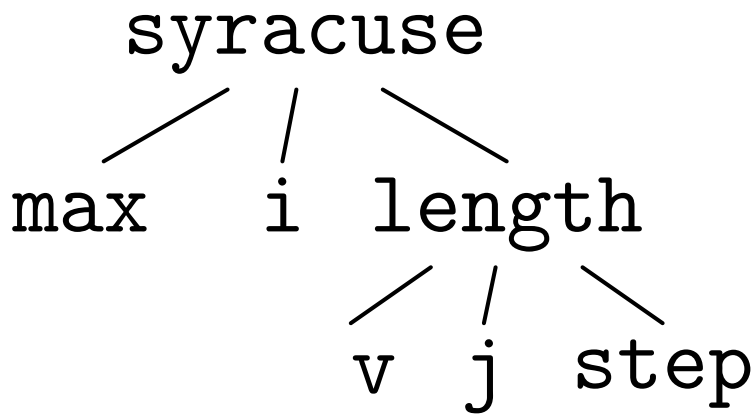
Definição (nível)

*O **nível** de uma declaração (de variável ou de procedimento) é o número de procedimentos por baixo da qual está declarada. O programa principal tem por nível 0.*

Definição (pai, antepassado, irmãos)

*Dizemos que p é o **pai** de um identificador y se y está declarado em p. Dizemos que p é um **antepassado** de y se p é ou y ou o pai de um antepassado de y. Dizemos que dois identificadores são **irmãos** se tem o mesmo pai.*

```
program syracuse;

procedure syracuse(max : integer);
var i : integer;
   procedure length();
   var v,j : integer;
      procedure step();
      begin
        ...
      end;
   begin
     ...
   end;
begin
   ...
end;

begin syracuse(100) end.
```



lembrete: fala-se de **porte** ou equivalentemente de **âmbito** para designar a área de influência ou de visibilidade de um identificador..
o porte é aqui a usual :
se o corpo de um procedimento p menciona um identificador então este é ou uma declaração local de p, ou um antepassado de p (incluído p ele próprio), ou um irmão de um antepassado de p

a análise de porte é realizada antes da tipagem

a sintaxe abstracta arquiva o resultado desta analise para *memória futura* ; para cada identificador do programa, convém por exemplo arquivar o seu nível de declaração

árvores de sintaxe abstracta provenientes da análise sintáctica :

```
type pint_expr =
  | PEconst of int
  | PEvar   of string
  | PEbinop of binop * pint_expr * pint_expr
  ...
```

(os identificadores são **strings**)

árvores de sintaxe abstracta provenientes da tipagem :

```
type ident = { ident: string; level: int; ... }

type int_expr =
  | Econst of int
  | Evar   of ident
  | Ebinop of binop * int_expr * int_expr
```

(os identificadores são agora do tipo `ident`)

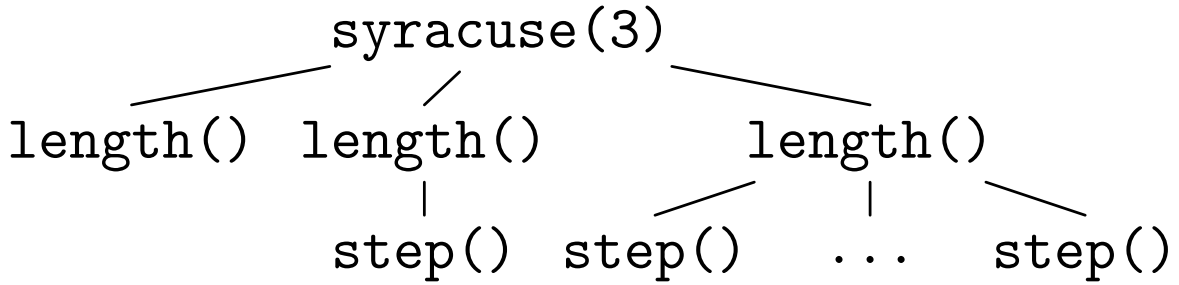
porte (exemplo)

```
program syracuse;

procedure syracuse0(max1 : integer);
var i1 : integer;
   procedure length1();
   var v2,j2 : integer;
      procedure step2();
      begin
        v2 := v2+1; if j2 = 2*(j2/2) then j2 := j2/2 else j2 := 3*j2+1
      end;
   begin
     v2 := 0; j2 := i1; while j2 <> 1 do step2(); writeln(v2)
   end;
begin
   i1 := 1;
   while i1 <= max1 do begin length1(); i1 := i1+1 end
end;

begin syracuse0(100) end.
```

Definição (árvore de activação)

Para uma dada execução definimos a ***árvore de activação*** *da forma seguinte : todo o nodo corresponde a uma chamada de procedimento* $p(e_1, \ldots, e_n)$ *e os sub-nodos correspondem às chamadas directamente efectuadas por* $p(e_1, \ldots, e_n)$.

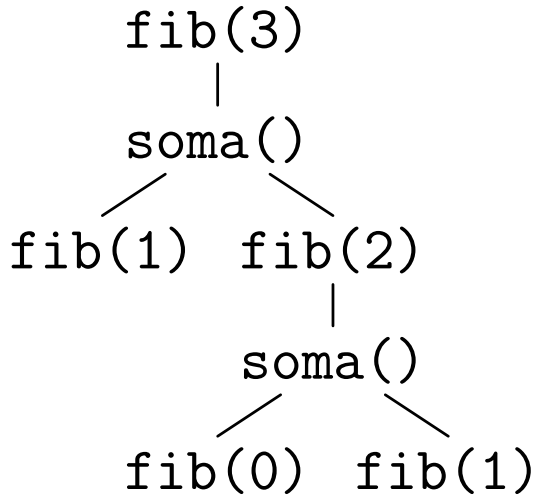
exemplo :



atenção : a profundidade da árvore de activação não coincide necessaramente com o nível de declaração

exemplo :

```
program fib;
var f : integer;

procedure fib(n : integer);
   procedure soma();
   var tmp : integer;
   begin fib(n-2); tmp := f;
         fib(n-1); f := f + tmp end;
begin
   if n <= 1 then f := n else soma()
end;

begin fib(3); writeln(f) end.
```



Definição (procedimento activo)

Dizemos que um procedimento está ***activo*** *num determinado momento da execução se ainda não acabamos a execução do corpo deste procedimento.*

Proposição

Se um procedimento p está activo então todos os seus antepassados estão activos

demonstração : por indução estrutural sobre a profundidade de p na árvore de activação

é verdade para o programa principal (só tem ele próprio como antepassado)

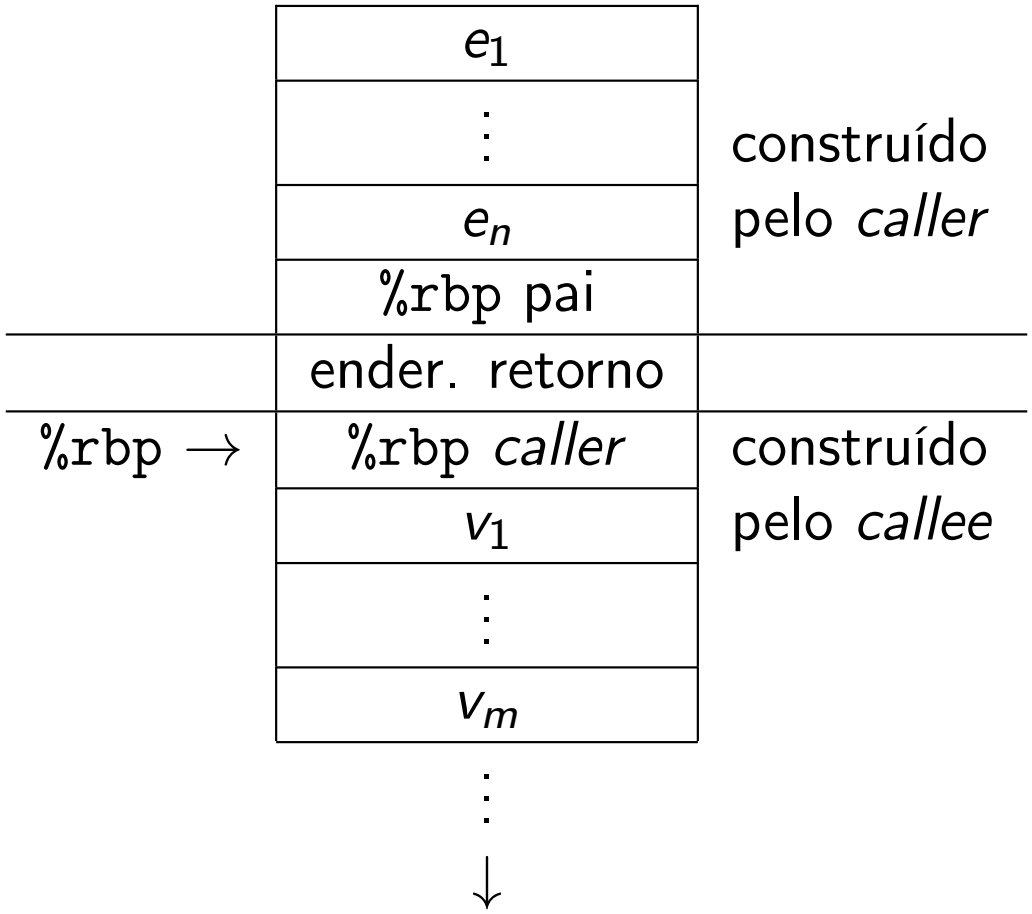
se p foi activado por um procedimento q então q está ainda activo (por definição) e todos os antepassados de q estão activos por HI ; ora, as regras de visibilidade implicam que ou q é o pai de p, ou p é o irmão de um antepassado de q; nos dois casos, todos os antepassados de p estão de facto activos □

é preciso escolher o local memória para cada variável e ser capaz de **calcular** este local em tempo de compilação

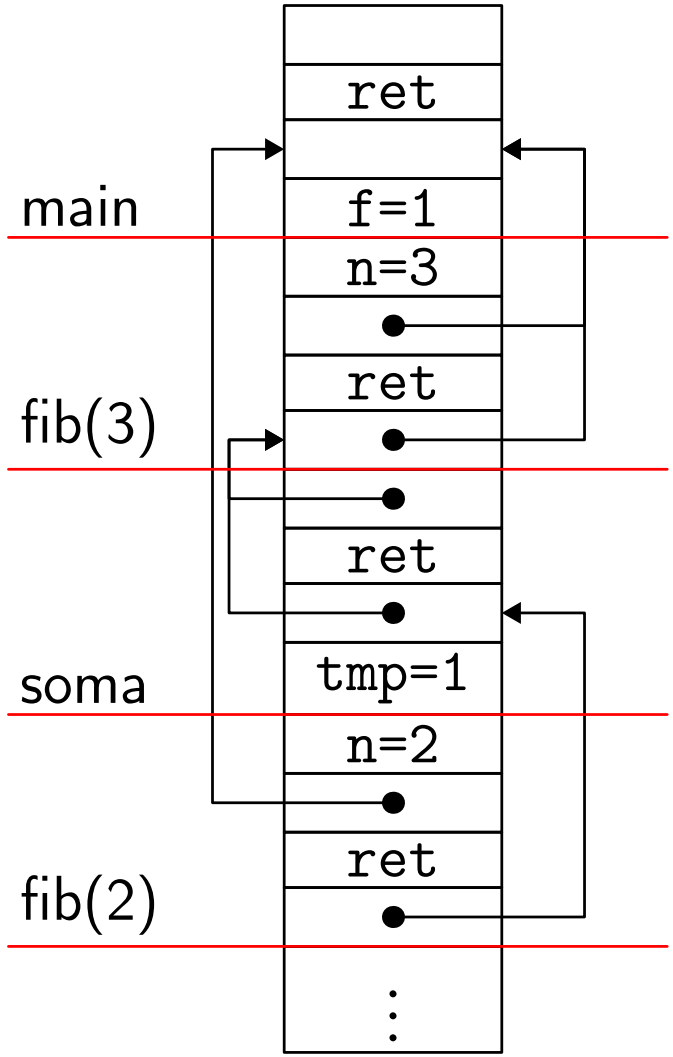
procedemos da seguinte forma : a cada procedimento activo corresponde uma porção da **pilha** chamada de **tabela de activação**, que contém

- os seus parâmetros efectivos
- as suas variáveis locais
- um apontador para a tabela de activação do seu procedimento pai (que existe, em virtude do resultado anterior !)
- um apontador para a tabela de activação do procedimento *caller*
- o endereço de retorno

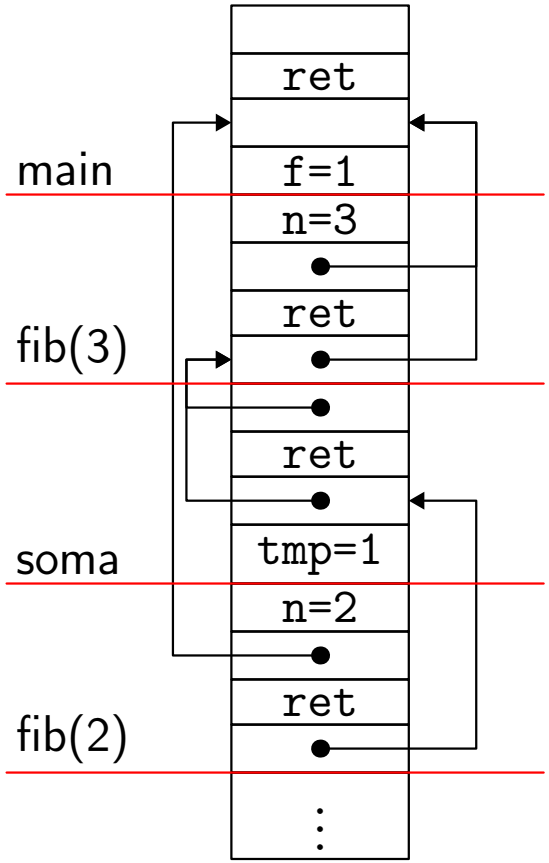
tabela de activação que corresponde à chamada $p(e_1, \ldots, e_n)$ de um procedimento $p(x_1, \ldots, x_n)$; `var` $v_1, \ldots, v_m$; `begin ... end`

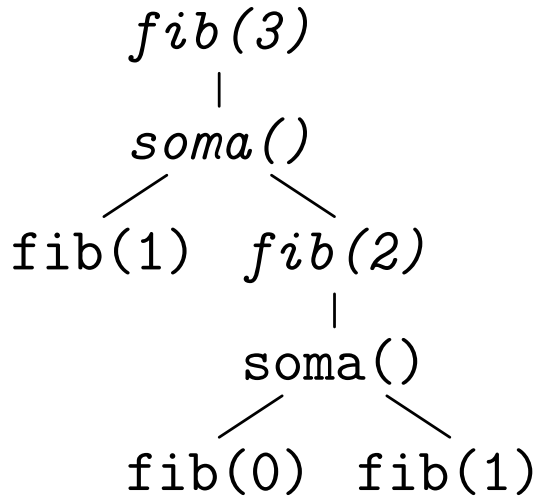


```
program fib;

var f : integer;

procedure fib(n : integer);
   procedure soma();
   var tmp : integer;
   begin
     fib(n-2); tmp := f;
     fib(n-1); f := f + tmp
   end;
begin
   if n <= 1 then f := n else soma()
end;

begin fib(3); writeln(f) end.
```



a cada momento as tabelas de activação na pilha correspondem a um caminho desde a raiz na árvore de activação





estendemos o tipo `ident` para identificar a posição da variável na tabela de activação :

```
type ident = { ident: string; level: int; offset: int }
```

supomos que esta informação foi calculada
(por exemplo durante a análise de porte)

vamos a isso !

para produzir código X86-64, utilizemos o módulo OCaml `X86_64` fornecido na página web da UC

com este módulo, escrevemos por exemplo

```
movq (imm 42) (reg rax)
```

para construir a instrução *assembly*

```
movq $42, %rax
```

e concatenamos estes pedaços de *assembly* com a operação ++

seguimos um esquema - por enquanto - simplista, utilizando a pilha para guardar os resultados intermédios (veremos mais tarde como utilizar eficientemente os registos)

escrevemos uma função `int_expr` que compila uma expressão aritmética

```
val int_expr : int -> int_expr -> X86_64.text
```

aquando do fim da execução, o resultado da expressão deve ficar em `%rdi`

o inteiro passado em argumento é o nível dentro do qual se encontra a expressão, ou seja $n + 1$ se a expressão se encontra num procedimento de nível n

começamos pelas constantes inteiras

```
let rec int_expr lvl = function
  | Econst n ->
      movq (imm n) (reg rdi)
```

e as operações aritméticas

```
  | Ebinop (Badd, e1, e2) ->
      int_expr lvl e1 ++
      pushq (reg rdi) ++
      int_expr lvl e2 ++
      popq rsi ++
      addq (reg rsi) (reg rdi)
  | Ebinop (Bsub, e1, e2) ->
      ...
```

funciona, mas claro, é particularmente ingénuo e simplista ; o código para 1+2 é

```
        movq  $1, %rdi
        pushq %rdi
        movq  $2, %rdi
        popq  %rsi
        addq  %rsi, %rdi
```

enquanto dispomos de 16 registos

o caso interessante é o de uma **variável** x

sejam `l` o seu nível e `ofs` a sua posição na tabela de activação

para encontrar a tabela de activação de x, devemos **seguir** `lvl - l` **vezes** o apontador de activação do pai

```
  | Evar { level = l; offset = ofs } ->
      assert (l <= lvl);
      movq (reg rbp) (reg rsi) ++
      iter (lvl - l) (movq (ind ~ofs:16 rsi) (reg rsi)) ++
      movq (ind ~ofs rsi) (reg rdi)
```

com

```
let rec iter n code =
  if n = 0 then nop else code ++ iter (n - 1) code
```

de igual forma, as expressões booleanas são compiladas com recurso a função

```
val bool_expr : int -> bool_expr -> X86_64.text
```

de forma análoga

```
let rec bool_expr lvl = function
  | Bcmp (Beq, e1, e2) ->
      int_expr lvl e1 ++ pushq (reg rdi) ++
      int_expr lvl e2 ++ popq rsi ++
      cmpq (reg rdi) (reg rsi) ++
      sete (reg dil) ++ movzbq (reg dil) rdi
  | ...
      (* deixado em exercício *) ...
```

cuidado : os operadores `and` e `or` devem ser avaliados de forma preguicosa (*lazy*), *i.e.* e_2 não é avaliada em e_1 `and` e_2 (resp. e_1 `or` e_2) se e_1 se avalia em `false` (resp. `true`)

as instruções são compiladas com a função

```
val stmt : int -> stmt -> X86_64.text
```

```
let rec stmt lvl = function
  | Swriteln e ->
      int_expr lvl e ++ call "print_int"
```

com

```
print_int:
        movq %rdi, %rsi
        movq $.Sprint_int, %rdi
        movq $0, %rax
        call printf
        ret
.data
.Sprint_int:
        .string "%d\n"
```

```
| Sif (e, s1, s2) ->
    (* deixado como exercício *)

| Swhile (e, s) ->
    (* deixado como exercício *)

| Sblock sl ->
    List.fold_left
      (fun code s -> code ++ stmt lvl s) nop sl
```

para uma chamada a um procedimento `p` de nível `l`, é necessário

1. empilhar os argumentos
2. empilhar o apontador para a tabela de activação do pai : para tal, basta seguir `lvl - l` vezes o apontador para a tabela do pai
3. chamar o código situado na etiqueta `p`
4. desempilhar os argumentos e o apontador para a tabela do pai

```
| Scall ({pident = p; plevel = l}, el) ->
    List.fold_left
      (fun acc e -> acc ++ int_expr lvl e ++ pushq (reg rdi))
      nop el ++
    movq (reg rbp) (reg rsi) ++
    iter (lvl - l) (movq (ind ~ofs:16 rsi) (reg rsi)) ++
    pushq (reg rsi) ++
    call p ++ addq (imm (8 + 8 * List.length el)) (reg rsp)
```

resta a **atribuição** `x := e`

consideramos aqui que o elemento esquerdo é reduzido a uma variável `x`

de forma geral, o membro esquerdo de uma atribuição deve ser um **valor esquerdo**, isto é uma expressão que designa uma localização memória

assim `3+1 := e` não tem sentido,
tanto como `f(x) := e` numa linguagem que dispõe de funções

é importante realçar que o significado do identificador `x` não é o mesmo à esquerda e à direita de `:=`
(é por isso que introduzimos e destacamos o conceito de valor esquerdo do conceito de valor direito)

como para o valor direito, seguimos `lvl - l` vezes o apontador para a tabela de activação do pai

```
| Sassign ({ level = l; offset = ofs }, e) ->
    int_expr lvl e ++
    movq (reg rbp) (reg rsi) ++
    iter (lvl - l) (movq (ind ~ofs:16 rsi) (reg rsi)) ++
    movq (reg rdi) (ind ~ofs rsi)
```

por enquanto, passamos os parâmetros **por valor**

i.e. o parâmetro formal é uma **nova variável** que toma por valor inicial o valor do parâmetro efetivo

em Pascal, o qualificativo `var` permito especificar que pretendemos uma passagem **por referência**

neste caso o parâmetro formal designa a **mesma variável** que o parâmetro efectivo, que deve assim ser igualmente uma variável (um valor esquerdo de forma mais abrangente)

```
program test;

procedure fact(n : integer; var f : integer);
begin
   f := 1;
   while n > 1 do begin
      f := n * f;
      n := n - 1
   end
end;

var x : integer;

begin
   fact(10, x);
   writeln(x)    { mostra 3628800 }
end.
```

para poder tomar em conta a passagem por referência, estendemos o tipo `ident` para que possa indicar se se trata de uma variável passada por referência

```
type by_reference = bool

type ident =
  { ident : string;
    level : int;
    offset : int;
    by_reference : by_reference }
```

(nota : vale sempre `false` no caso das variáveis locais)

numa uma chamada tal que `p(e)`, o parâmetro efectivo e não é nem tipado nem compilado da mesma forma quando passado por referência ou por valor

quando o parâmetro é passado por referência, a tipagem vai então

1. verificar que se trata bem de uma variável (valor esquerdo)
2. dar indicação de que pode ser passada por referência

uma forma de prosseguir consiste em juntar uma construção de « cálculo de valor esquerdo » na sintaxe das expressões

```
type int_expr =
  ...
  | Eaddr of ident
```

e substituir, caso necessário, o parâmetro efectivo `e` por `Eaddr e`

é preciso juntar o código correspondente em `int_expr` :

```
let rec int_expr lvl = function
  | Eaddr { level = l; offset = ofs; by_reference = br } ->
      assert (l <= lvl);
      movq (reg rbp) (reg rdi) ++
      iter (lvl - l) (movq (ind ~ofs:16 rdi) (reg rdi)) ++
      addq (imm ofs) (reg rdi) ++
      if br then movq (ind rdi) (reg rdi) else nop
```

nota : o caso `br = true` corresponde ao caso de uma variável ela própria passada por referência

é preciso também modificar o cálculo dos valores direitos :

```
  | Evar { level = l; offset = ofs; by_reference = br } ->
      assert (l <= lvl);
      movq (reg rbp) (reg rsi) ++
      iter (lvl - l) (movq (ind ~ofs:16 rsi) (reg rsi)) ++
      movq (ind ~ofs rsi) (reg rdi) ++
      if br then movq (ind rdi) (reg rdi) else nop
```

assim como o da atribuição:

```
| Sassign ({level=l; offset=ofs; by_reference=br}, e) ->
    int_expr lvl e ++
    movq (reg rbp) (reg rsi) ++
    iter (lvl - l) (movq (ind ~ofs:16 rsi) (reg rsi)) ++
    (if br then movq (ind ~ofs rsi) (reg rsi)
     else addq (imm ofs) (reg rsi)) ++
    movq (reg rdi) (ind rsi)
```

contudo, não há alterações por fazer na chamada (graças à nova construção Eaddr)

resta-nos a compilação das declarações

```
type pident = { pident: string; plevel: int }

type procedure =
  { name    : pident;
    formals: (string * by_reference) list;
    locals  : decl list;
    body    : stmt; }

and decl =
  | Var       of string list
  | Procedure of procedure
```

compilamos uma declaração com

```
val decl : decl -> X86_64.text
```

```
let rec decl = function
  | Var _         -> nop
  | Procedure p -> procedure p ++ decls p.locals

and decls dl =
  List.fold_left (fun code d -> code ++ decl d) nop dl
```

```
let frame_size dl = (* 8 vezes o número de variáveis de dl *)

let procedure p =
  let fs = 8 + frame_size p.locals in
```

```
p:
        subq $fs, %rsp              # alloca a tabela
        movq %rbp, fs-8(%rsp)       # salva %rbp
        leaq fs-8(rsp), %rbp        # posiciona rbp
```

```
  ++ stmt (p.name.plevel + 1) p.body ++
```

```
        movq %rbp, %rsp             # de-aloca a tabela
        popq %rbp                   # restaura %rbp
        ret                         # retorno ao caller
```

basta considerar o programa como um procedimento de nível -1

```
let prog p =
  let fs = 8 + frame_size p.locals in

  main:
       subq $fs, %rsp          # aloca a tabela
       leaq fs-8(rsp), %rbp    # posiciona rbp

  ++ stmt 0 p.body ++

       addq $fs, %rsp          # de-aloca a tabela
       movq $0, %rax           # código de retorno 0
       ret

  ++ decls p.locals
```

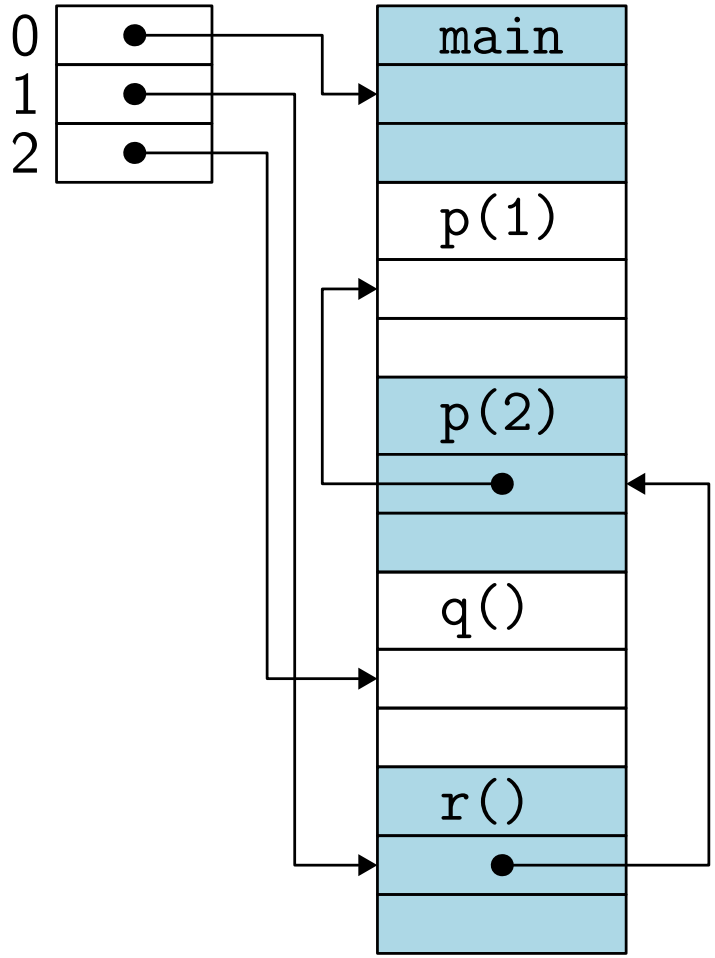
podemos melhorar o acesso às variáveis

a ideia é conservar numa tabela um apontador para a **última** tabela de activação de cada nível ; e as tabelas de activação de um mesmo nível formam uma lista encadeada

- quando chamamos um procedimento de um nível *n*
  - fazemo-lo apontar para o último procedimento de nível *n*
  - ele próprio torna-se o ultimo procedimento de *n*
- quando retornamos da chamada, desempilhamos

o acesso a uma variável de nível *n* faz-se agora em tempo constante

```
program foo;

procedure r();
begin ... end;

procedure p(n : integer);
  procedure q();
  begin r() end;
begin
  if n = 1 then p(2) else q()
end;

begin p(1) end.
```



a tabela tem um tamanho conhecido em tempo de compilação (nível máximo) e pode ser alocado na base da pilha, por exemplo

correção da compilação

o compilador deve respeitar a **semântica** da linguagem (correção)

se a linguagem fonte está munido de uma semântica $\rightarrow_s$ e a linguagem máquina de uma semântica $\rightarrow_m$, e se a expressão e é compilada em $C(e)$ então devemos ter um « diagrama que comuta » :

$$\begin{array}{ccc} e & \xrightarrow{\star}_s & v \\ \downarrow & & \approx \\ C(e) & \xrightarrow{\star}_m & v' \end{array}$$

onde $v \approx v'$ expressa que os valores v e v' coincidem

consideramos aqui as expressões aritméticas sem variáveis

$$e ::= n \mid e + e \mid e - e$$

e mostremos a correção da compilação

damo-nos uma semântica por redução (i.e. *small step*) para a linguagem fonte

$$\begin{array}{lcl} v & ::= & n \\ E & ::= & \Box \mid E + e \mid v + E \mid E - e \mid v - E \end{array}$$

$$n_1 + n_2 \xrightarrow{\epsilon} n \quad \text{avec } n = n_1 + n_2$$
$$n_1 - n_2 \xrightarrow{\epsilon} n \quad \text{avec } n = n_1 - n_2$$

damo-nos igualmente uma semântica por redução para a linguagem alvo

$$
\begin{array}{rcl}
m & ::= & \texttt{movq}\ \$n, r \\
& | & \texttt{addq}\ \$n, r \mid \texttt{addq}\ r, r \mid \texttt{subq}\ \$n, r \mid \texttt{subq}\ r, r \\
& | & \texttt{movq}\ (r), r \mid \texttt{movq}\ r, (r) \mid \\
r & ::= & \texttt{\%rdi} \mid \texttt{\%rsi} \mid \texttt{\%rsp}
\end{array}
$$

um estado S é o mapa dos valores para os registos, R,
e de um estado da memória, M

$$
\begin{array}{rcl}
R & ::= & \{\texttt{\%rdi} \mapsto n; \texttt{\%rsi} \mapsto n; \texttt{\%rsp} \mapsto n\} \\
M & ::= & \mathbb{N} \to \mathbb{Z}
\end{array}
$$

definimos a semântica de uma instrução m por uma redução da forma

$$R, M, m \xrightarrow{m} R', M'$$

e de uma sequência de instrução $m_1, \ldots, m_k$ como o fecho transitivo

a redução $R, M, m \xrightarrow{m} R', M'$ é definida por

$$R, M, \texttt{movq } \$n, r \xrightarrow{m} R\{r \mapsto n\}, M$$

$$R, M, \texttt{addq } \$n, r \xrightarrow{m} R\{r \mapsto R(r) + n\}, M$$

$$R, M, \texttt{addq } r_1, r_2 \xrightarrow{m} R\{r_2 \mapsto R(r_1) + R(r_2)\}, M$$

idem para subq

$$R, M, \texttt{movq } (r_1), r_2 \xrightarrow{m} R\{r_2 \mapsto M(R(r_1))\}, M$$

$$R, M, \texttt{movq } r_1, (r_2) \xrightarrow{m} R, M\{R(r_2) \mapsto R(r_1)\}$$

pretendemos mostrar que se

$$e \xrightarrow{\star} n$$

e se

$$R, M, code(e) \xrightarrow{m}{}^{\star} R', M'$$

então $R'(\texttt{\%rdi}) = n$

procedemos por indução estrutural sobre e

estabelecemos um resultado mais forte (um **invariante**), ou seja :

se $e \xrightarrow{\star} n$ e $R, M, code(e) \xrightarrow{m}{}^{\star} R', M'$ então

$$\begin{cases} R'(\texttt{\%rdi}) = n \\ R'(\texttt{\%rsp}) = R(\texttt{\%rsp}) \\ \forall a \geq R(\texttt{\%rsp}),\ M'(a) = M(a) \end{cases}$$

- caso $e = n$

  temos $e \xrightarrow{\star} n$ et $code(e) =$ `movq $`n`,%rdi` e o resultado é imediato

- caso $e = e_1 + e_2$

  temos $e \xrightarrow{\star} n_1 + e_2 \xrightarrow{\star} n_1 + n_2 \xrightarrow{\star} n$ com $n = n_1 + n_2$, e

$$
\begin{aligned}
code(e) = \quad & code(e_1) \\
& \texttt{addq \$-8,\%rsp} \\
& \texttt{movq \%rdi,(\%rsp)} \\
& code(e_2) \\
& \texttt{movq (\%rsp),\%rsi} \\
& \texttt{addq \$8,\%rsp} \\
& \texttt{addq \%rsi,\%rdi}
\end{aligned}
$$

	R, M	
$code(e_1)$	R_1, M_1	por HI $R_1(\texttt{\%rdi}) = n_1$ e $R_1(\texttt{\%rsp}) = R(\texttt{\%rsp})$ $\forall a \geq R(\texttt{\%rsp}),\ M_1(a) = M(a)$
`addq $-8,%rsp` `movq %rdi,(%rsp)`	R'_1, M'_1	$R'_1 = R_1\{\texttt{\%rsp} \mapsto R(\texttt{\%rsp}) - 8\}$ $M'_1 = M_1\{R(\texttt{\%rsp}) - 8 \mapsto n_1\}$
$code(e_2)$	R_2, M_2	por HI $R_2(\texttt{\%rdi}) = n_2$ e $R_2(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8$ $\forall a \geq R(\texttt{\%rsp}) - 8,\ M_2(a) = M'_1(a)$

`movq (%rsp),%rsi`
`addq $8,%rsp`
`addq %rsi,%rdi`
	R', M_2	$R'(\texttt{\%rdi}) = n_1 + n_2$
		$R'(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8 + 8 = R(\texttt{\%rsp})$
		$\forall a \geq R(\texttt{\%rsp}),$
		$M_2(a) = M'_1(a) = M_1(a) = M(a)$

e as outras linguagens imperativas ?

existe outros modos de passagem de parâmetros

- **cópia-restauração** (*copy-restore*)

  mistura de avaliação por referência e por valor : na chamada, uma nova variável é criada, esta recebe o valor direito do argumento ; no retorno do procedimento o valor final é copiado no local designado pelo valor esquerdo do argumento

  usado em certo compiladores Fortran

- **chamada por nome** (*call-by-name*)

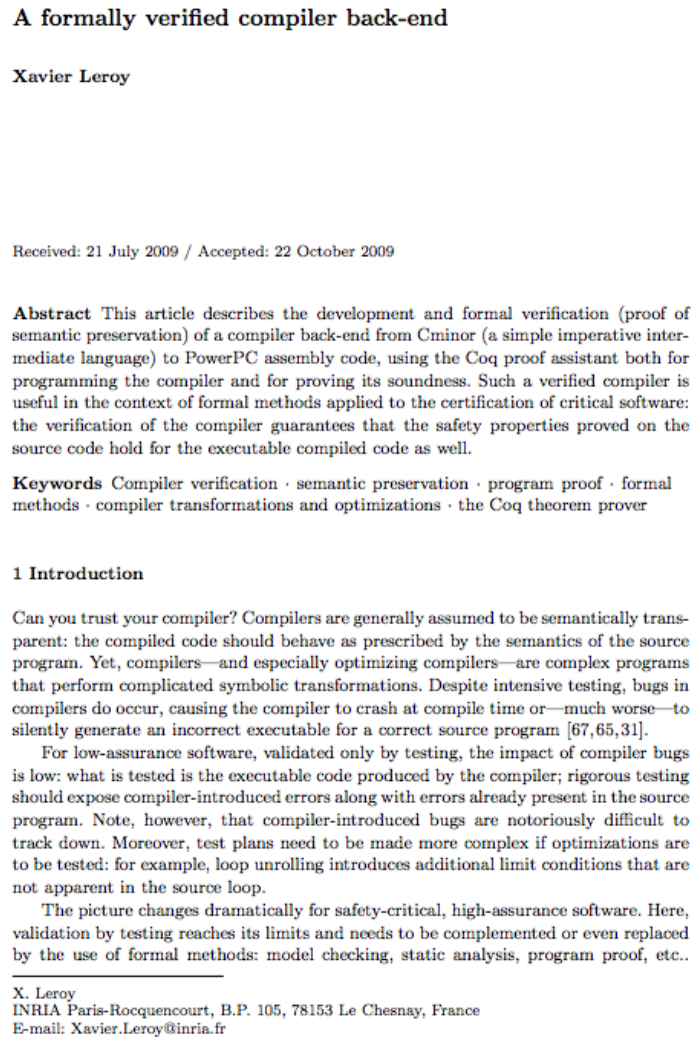
  os argumentos efectivos substituem textualmente os argumentos formais (com o cuidado acrescido de evitar as capturas de variáveis, mas sem evitar alguns problemas (e.g. `swap(i, t[i])`)

  avaliação por omissão de Algol 60

conclusão

artigos na base de muito do que aqui se falou:

- Xavier Leroy. A formally verified compiler back-end. Journal of Automated Reasoning, 43(4):363-446, 2009. (link)
- versão mais compacta na POPL'06: Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In 33rd ACM symposium on Principles of Programming Languages, pages 42-54. ACM Press, 2006. (link)



A formally verified compiler back-end

Xavier Leroy

Received: 21 July 2009 / Accepted: 22 October 2009

Abstract This article describes the development and formal verification (proof of semantic preservation) of a compiler back-end from Cminor (a simple imperative intermediate language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its soundness. Such a verified compiler is useful in the context of formal methods applied to the certification of critical software: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Keywords Compiler verification · semantic preservation · program proof · formal methods · compiler transformations and optimizations · the Coq theorem prover

1 Introduction

Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compiler to crash at compile time or—much worse—to silently generate an incorrect executable for a correct source program [67,65,31].

For low-assurance software, validated only by testing, the impact of compiler bugs is low: what is tested is the executable code produced by the compiler; rigorous testing should expose compiler-introduced errors along with errors already present in the source program. Note, however, that compiler-introduced bugs are notoriously difficult to track down. Moreover, test plans need to be made more complex if optimizations are to be tested: for example, loop unrolling introduces additional limit conditions that are not apparent in the source loop.

The picture changes dramatically for safety-critical, high-assurance software. Here, validation by testing reaches its limits and needs to be complemented or even replaced by the use of formal methods: model checking, static analysis, program proof, etc..

X. Leroy
INRIA Paris-Rocquencourt, B.P. 105, 78153 Le Chesnay, France
E-mail: Xavier.Leroy@inria.fr

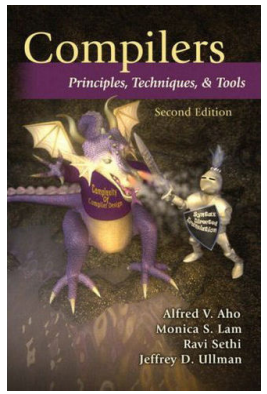
- aulas práticas
  - continuação da prática anterior
- próxima aula teórica
  - compilação das linguagens funcionais

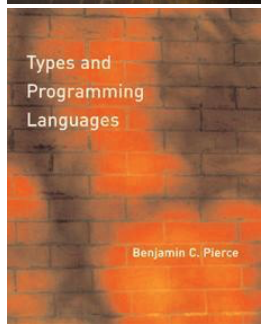
leituras de referência

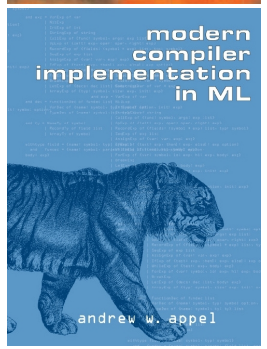
estes acetatos resultam essencialmente de uma adaptação do material pedagógico gentilmente cedido pelo Jean-Christophe Filliâtre (link1, link2)

adicionalmente poderá consultar as obras seguintes

- **Modern Compilers: Principles, Techniques, and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman
- **Types and Programming Languages**, Benjamin C. Pierce
- **Modern Compiler Implementation**, Andrew W. Appel (3 versões: ML, C, Java)







material adicional

Vectores

comecemos pelos **vectores estáticos**, isto é vectores cujo tamanho é conhecido em tempo de compilação

em Pascal, podemos introduzir vectores unidimensionais

```
var t : array [2..10] of integer;
```

ou multidimensionais

```
var u : array [-5..5, 1..10] of integer;
```

ou mesmo vectores de vectores

```
var v : array [0..11] of array [0..31] of integer;
```

o tamanho de um dado (em byte) calcula-se facilmente por recorrência sobre o seu tipo :

$$\begin{aligned} \mathit{tamanho}(\texttt{integer}) &= 4 \\ \mathit{tamanho}(\texttt{array}\ [l_1..u_1, \ldots, l_k..u_k]\ \texttt{of}\ \tau) &= \prod_{i=1}^{k}(u_i - l_i + 1) \times \mathit{tamanho} \end{aligned}$$

exemplo : o vector

```
var u : array [-5..5, 1..10] of integer;
```

ocupa 440 bytes em memória

para *arrumar* os elementos de um vector multi-dimensional, temos *a priori* a escolha de uma arrumação por linhas ou por colunas

no entanto, se declaramos

```
var u : array [-5..5, 1..10] of integer;
```

podemos pretender que `u[i]` seja visto como um elemento de tipo

```
array [1..10] of integer
```

e é então preferível privilegiar uma arrumação por linhas

seja um vector

$$t : \texttt{array}\ [l_1..u_1, \ldots, l_k..u_k]\ \texttt{of}\ \tau$$

arrumado em memória no endereço *base*

o local memória a do elemento $t[i_1, \ldots, i_k]$ é então

$$\begin{aligned} a = \mathit{base} \quad & + \quad (i_1 - l_1) \times \mathit{tamanho}(\texttt{array}\ [l_2..u_2, \ldots, l_k..u_k]\ \texttt{of}\ \tau) \\ & + \quad (i_2 - l_2) \times \mathit{tamanho}(\texttt{array}\ [l_3..u_3, \ldots, l_k..u_k]\ \texttt{of}\ \tau) \\ & \vdots \\ & + \quad (i_{k-1} - l_{k-1}) \times \mathit{tamanho}(\texttt{array}\ [l_k..u_k]\ \texttt{of}\ \tau) \\ & + \quad (i_k - l_k) \times \mathit{tamanho}(\tau) \end{aligned}$$

seja $d_i \stackrel{\text{def}}{=} u_i - l_i + 1$

então

$$\begin{aligned} a = base \quad & + \quad (i_1 - l_1) \times d_2 \times \cdots \times d_k \times tamanho(\tau) \\ & + \quad (i_2 - l_2) \times d_3 \times \cdots \times d_k \times tamanho(\tau) \\ & \quad \vdots \\ & + \quad (i_{k-1} - l_{k-1}) \times d_k \times tamanho(\tau) \\ & + \quad (i_k - l_k) \times tamanho(\tau) \end{aligned}$$

ou seja

$$\begin{aligned} a \quad = \quad & (\ldots(i_1 \times d_2 + i_2) \times d_3 + i_3) \cdots + i_k) \times tamanho(\tau) + \\ & (base - (\ldots(l_1 \times d_2 + l_2) \times d_3 + l_3) \cdots + l_k) \times tamanho(\tau)) \end{aligned}$$

o segundo termo pode ser calculado durante a compilação ; o segundo pode ser calculado dinamicamente com recurso ao método de Horner para limitar o número de multiplicação

Passagem dos vectores em parâmetro

o princípio de passagem dos parâmetros não muda para os vectores estáticos

notemos no entanto que

- as dimensões do vector parâmetro formal deve ser dado

```
procedure p(t : array[2..5] of integer);
```

  para permitir que o cálculo do endereço no interior de `p`
- os argumentos não ocupam todos o mesmo espaço na tabela de activação (este depende agora do tipo)
- um vector passado por valor deve ser copiado, o que já não é mais uma operação atómica
- um elemento de um vector é um valor esquerdo

```
var t : array[1..10] of integer;
procedure p(var x : integer); begin x := 42 end;
begin p(t[1]) end.
```

impor um tamanho estático aos vectores é uma restrição (era um defeito apontado a concepção original de Pascal)

em prática, podemos querer declarar vectores cujo tamanho só é conhecido durante a execução, tais que

```
procedure p(n : integer);
var t : array [1..n*n] of integer;
begin ... end;
```

```
procedure tri(n : integer; var t : array [1..n] of integer);
...
```

somente o número de dimensão é conhecido estaticamente

ocorrem dois problemas

- o tamanho em memória do vector já não é conhecido em tempo de compilação ; em particular, um vector já não pode ficar alojado na tabela de activação como qualquer outra variável
- o cálculo do endereço de um elemento $t[i_1, \ldots, i_k]$ necessita dos l_i e dos d_i, que já não são conhecidos estaticamente

a solução : a tabela de activação contém um **descritor de tabela**, que tem a forma seguinte

$base$
tamanho total da tabela
$base - (\ldots(l_1 \times d_2 + l_2) \times d_3 + l_3) \cdots + l_k) \times tamanho(\tau)$
d_2
d_3
$\vdots$
d_k
$tamanho(\tau)$

em particular, o tamanho deste descriptor é conhecido em tempo de compilação (só depende do número de dimensões)

(nota : se pretendemos testar a boa gestão dos índices dos vectores, é preciso também arquivar os valores l_i e u_i neste descriptor)

podemos continuar o arquivo do vector na pilha, mais neste caso é necessário conservar as localizações conhecidas estaticamente para os parâmetros e as variáveis locais ; a tabela de activação toma então a forma seguinte

	conteúdo dos vectores passados em parâmetro	tamanho desconhecido
	parâmetro 1	(descritor se vector)
	...	
	parâmetro n	tamanho conhecido
`$fp`→	`$fp` pai, etc.	
	variável local 1	(descritor se vector)
	...	
	variável local m	tamanho conhecido
	conteúdo dos vectores locais	tamanho desconhecido

Estruturas

em Pascal, introduzimos estruturas da seguinte forma

```
var e : record x : integer; y : integer end
```

de forma geral, um tipo estrutura tem a seguinte forma

$$\tau = \texttt{record}\ x_1 : \tau_1\,;\ x_2 : \tau_2\,;\ \ldots x_n : \tau_n\ \texttt{end}$$

onde os x_i são designados de campos da estrutura

acede-se a um campo com a notação `e.x`, que é simultaneamente valor esquerdo e valor direito

uma estrutura é naturalmente representado por um bloco de memória onde os valores dos diferentes campos são arquivados contiguamente

o tamanho ocupado é conhecido em tempo de compilação :

$$\mathit{tamanho}(\texttt{record } x_1 : \tau_1 \texttt{;} \ x_2 : \tau_2 \texttt{;} \ \ldots x_n : \tau_n \texttt{ end}) = \sum_{i=1}^{n} \mathit{tamanho}(\tau_i)$$

nota : se um campo é um vector dinâmico, então o valor deste campo é o descriptor deste vector (cujo tamanho é conhecido estaticamente)

o valor esquerdo de uma estrutura é o endereço do início da sua localização memória, que coincide com o endereço do seu primeiro campo

para cada campo x_i podemos calcular estaticamente a **desvio** $\delta(x_i)$ deste campo relativamente ao início

$$\delta(x_i) = \sum_{j=1}^{i-1} tamanho(\tau_j)$$

o valor esquerdo de e . x_i é obtido acrescentando $\delta(x_i)$ ao valor esquerdo de e