Implementação de Linguagens de Programação Lógica

Execução de Prolog com Alto Desempenho

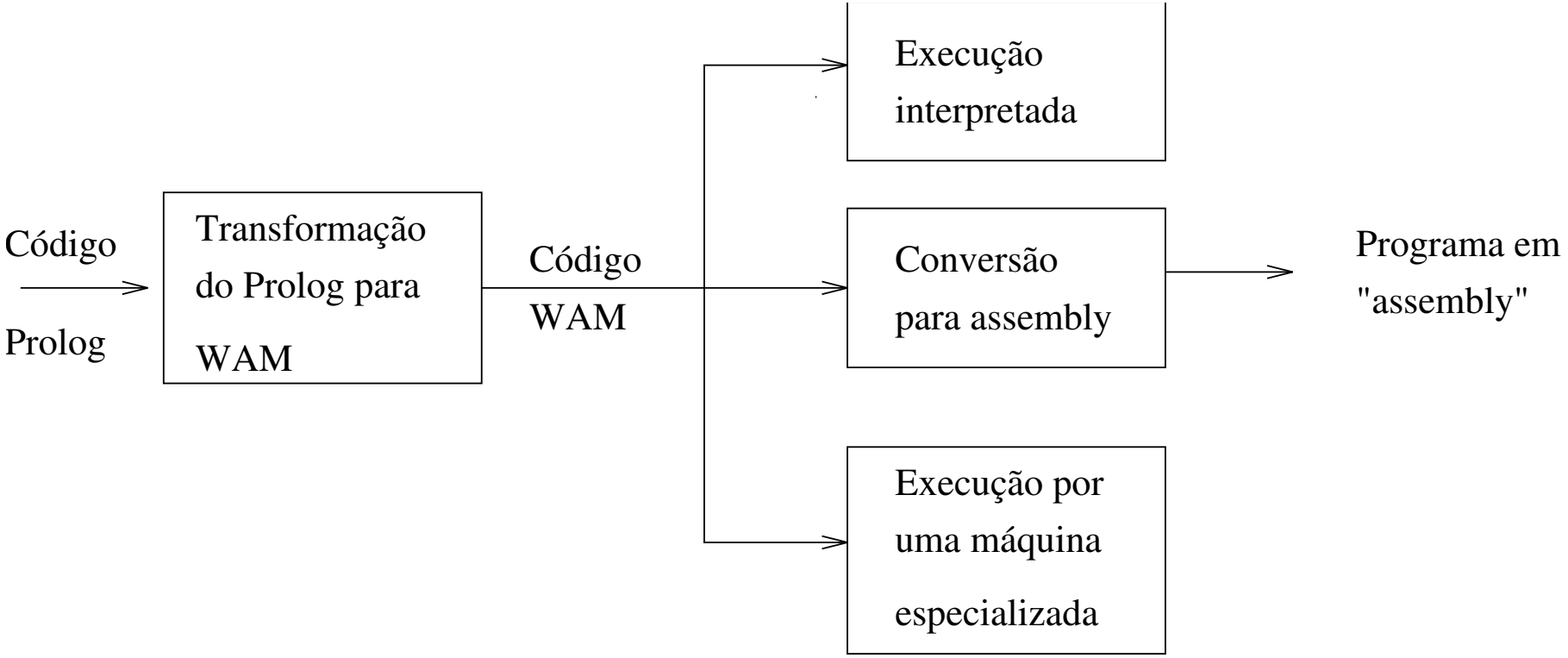
Ricardo Lopes
rslopes@ncc.up.pt
DCC-FCUP

Tópicos Avançados de Informática
Mestrado em Informática 2003/04

Warren Abstract Machine (WAM)

- Em 1983, David Warren criou a WAM um modelo para a execução de Prolog que rapidamente se tornou norma nas implementações de Prolog.
- São três as principais formas de implementação de Prolog usando a WAM:
  - ⋆ implementações por hardware;
  - ⋆ implementações usando um emulador;
  - ⋆ e implementações que traduzem o código WAM para código nativo.

Warren Abstract Machine (WAM)



WAM: Implementações por Hardware

- A motivação original no desenho de máquinas abstractas era a construção de hardware para suportar Prolog eficientemente.
- As implementações por hardware não tiveram muito sucesso devido à falta de mercado, pois são poucas as pessoas interessadas em comprar uma máquina apenas capaz de correr Prolog.
- Mesmo que o Prolog fosse muito popular, o hardware especializado teria muita dificuldade em conseguir competir pois os microprocessadores beneficiam de investimentos maiores.
- De notar que as instruções WAM realizam operações muito complexas, como seja a unificação, e o desenvolvimento das novas arquitecturas tem caminhado na direcção oposta, aparecendo cada vez mais arquitecturas com um pequeno número de instruções simples.

WAM: Implementações usando um Emulador

- As dificuldades em obter performance superior à possível em arquitecturas tradicionais e o custo de desenvolver hardware para suportar Prolog justificam que a WAM tenha sido principalmente usada em **implementações por software**.

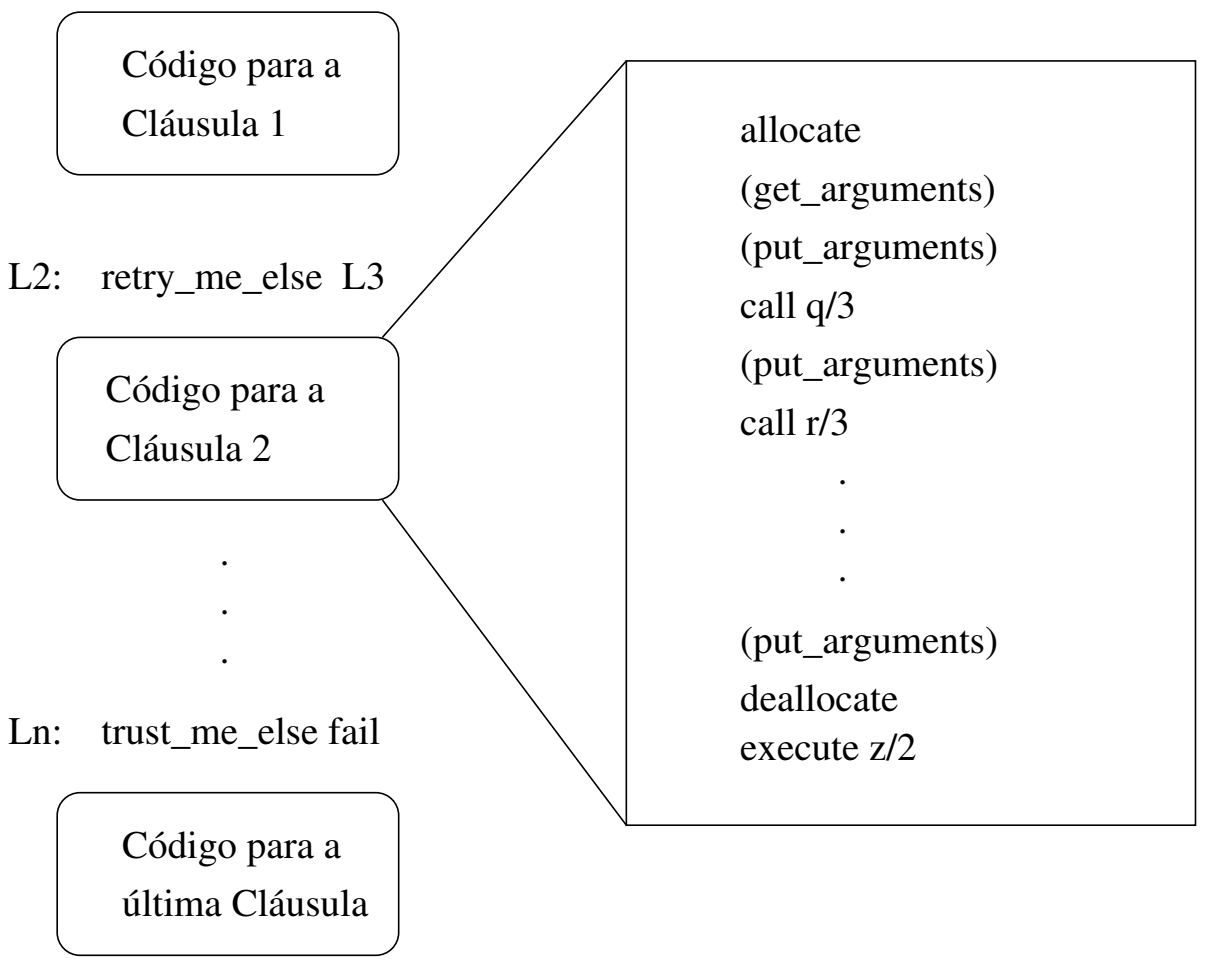
- Estas implementações, tradicionalmente compilam o código Prolog para o código de uma máquina abstracta que é depois emulado em tempo de execução (exemplo: YAP, SICStus Prolog).

- As implementações usando um emulador de WAM tornaram-se muito populares, pois são relativamente fáceis de implementar e têm um bom desempenho.

```
p(X,Y,Z):- ...
p(A,B,C):- q(A,Z,W), r(W,T,B), ... , z(A,X).
   ...
p(Q,W,E):- ...
```

Código Prolog

Código WAM



```
do {
   switch(*PC) {

   case get_struct :           /* get_struct f, Xi */
                     ...
                    PC = PC + 3;
                    break;

   case get_list   :           /* get_list Xi */
                     ...
                    PC = PC + 2;
                    break;

   case proceed    :
                     ...
                    PC = PC + 1;
                    break;
  ...

  }
} while(*PC!=END);
```

WAM: Implementações usando um Emulador

- Contudo este tipo de sistemas não aproveitam ao máximo o desempenho da máquina.
- São duas as razões principais para tal facto:
  - **o overhead** intrínseco em emular as instruções de outra arquitectura, já que todas as instruções da máquina abstracta no fim têm de realizar um salto para o código da próxima instrução.
  - **a complexidade das instruções WAM**, que não permitem que muitas optimizações se possam realizar. Note-se que é impraticável ter instruções optimizadas para todos os casos possíveis.

WAM: Implementações de código nativo

- As implementações de Prolog de código nativo têm vindo a ganhar popularidade porque têm vindo a conseguir aumentos significativos de performance na execução de programas Prolog.
- O bom desempenho destes sistema deve-se à geração de código que corre directamente no hardware, sem precisar de ser emulado.
- No entanto, a transformação de código Prolog para código nativo é um problema complexo.
- Uma alternativa à geração de código nativo, é a geração de código *C*:
  - Um exemplo deste tipo de implementações é a `wamcc`.
  - A filosofia subjacente a este sistema, é deixar o trabalho de optimização para o compilador de *C*.
  - Infelizmente o desempenho deste sistema não é muito impressionante, mesmo comparando com sistemas que usam emuladores.

WAM: Implementações de código nativo

- As implementações de código nativo podem ser divididas em duas grandes classes:
  - **Sistemas que compilam Prolog directamente para código nativo.** A compilação directa para Prolog é um problema muito complexo.
  - **Sistemas que geram o código nativo a partir da WAM.** Este tipo de aproximação é bastante útil quando já existe à partida um sistema baseado na WAM, evitando que se crie um sistema a partir do zero.

WAM: Implementações de código nativo

- Considerando a grande separação entre o Prolog e as instruções de uma máquina convencional, é evidente que para gerar bom código nativo, a transição tenha que ser feita usando várias linguagens intermédias.
- uma solução usada por alguns sistemas, consiste em fazer a transição do Prolog para o código nativo, usando três linguagens intermédias distintas:
  - **WAM**: conseguimos ter uma optimização num âmbito mais superior, classificação de variáveis (globais ou locais a nível da WAM), optimização das variáveis temporárias, e eliminação de instruções desnecessárias do tipo `unify_var` e `pop`.
  - **WAM Two-Streams**: consegue-se simplificar o código reduzindo o número de saltos existentes no seio das instruções da WAM.
  - **Simple Intermediate Language**: uma linguagem intermédia de mais baixo nível. Com este passo conseguimos ter uma optimização de código a mais baixo nível, eliminando entre outros, atribuições desnecessárias e fazendo um uso mais eficaz dos registos da máquina destino.

WAM: Implementações de código nativo

- A maior vantagem em usar três linguagens intermédias de níveis diferentes é a de permitir obtermos um maior leque de possíveis optimizações do que seria possível se tivéssemos usado apenas uma linguagem intermédia.
- Outra vantagem de dividir a compilação em várias fases, é a de se tornar mais simples o processo de alterar uma fase sem ter que para isso remodelar as outras.

WAM Two-Streams

A WAM compila a unificação numa simples sequência de instruções. Esta técnica tem alguns problemas:

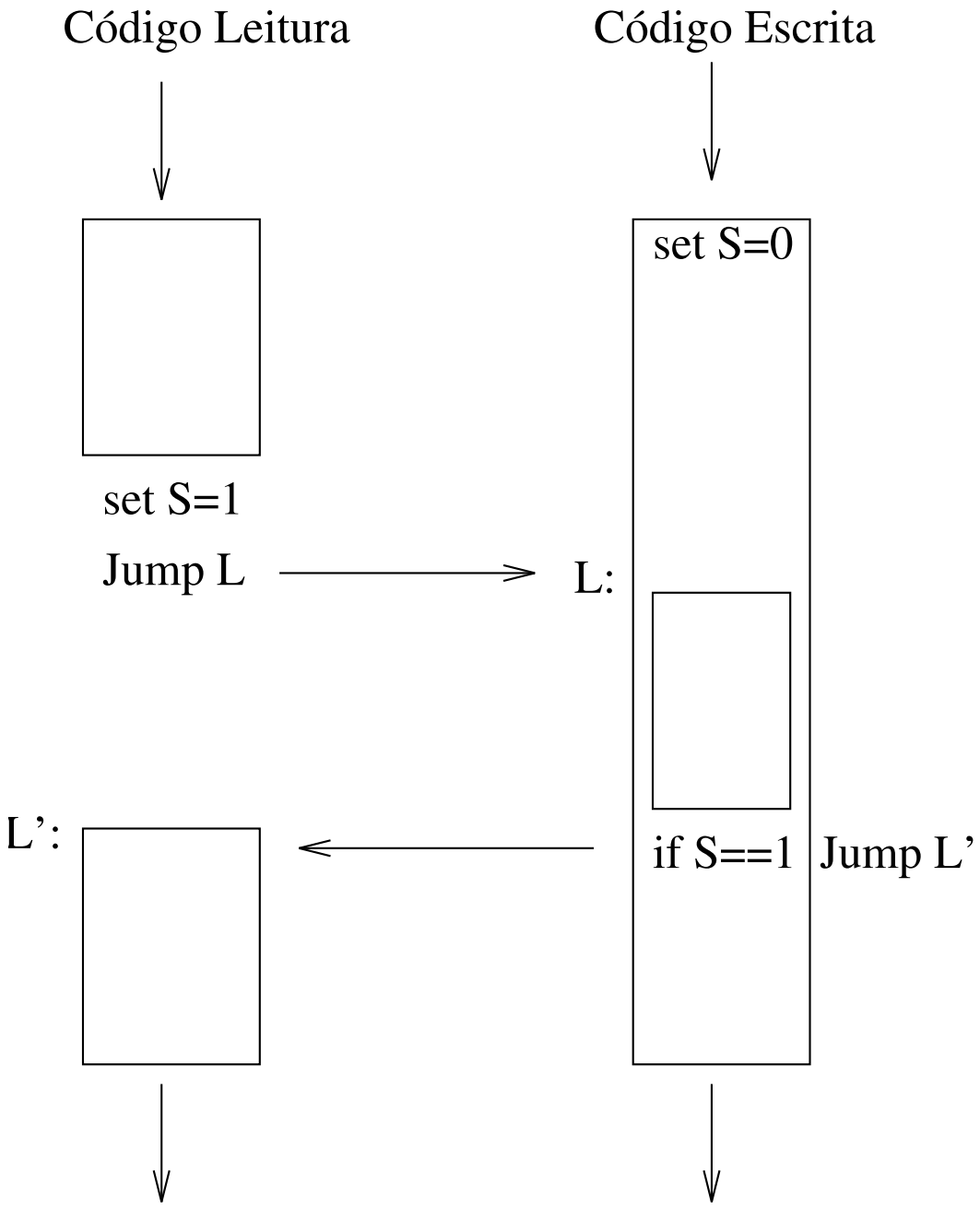
- O modo de escrita não é propagado aos subtermos. Por exemplo, a unificação de M=h(f(a)) é compilada em duas unificações independentes, como M=h(N), N=f(a). Ora, se a variável M ainda não foi inicializada, N será também uma variável livre, facto este que não é propagado para a segunda unificação. Isto obriga a algumas operações desnecessárias, como seja a desreferenciação de N, a sua atribuição (binding) e verificação na trilha.
- As instruções têm dois modos de execução, modo de leitura e modo de escrita. O modo corrente é guardado num registo, que é inicializado na instrução `get_structure/list` e verificado em todas as instruções `unify`.
- Pobre tradução para código nativo. Como as instruções `unify` têm dois modos o código de escrita e o de leitura ficam intercalados, resultando num grande número de saltos (`jumps`).

WAM Two-Streams

- O algoritmo 2-Streams constitui um elegante método de fazer unificação mais eficiente do que a WAM.

- A ideia é compilar a unificação em duas áreas de código, uma para o código de escrita e outra para o código de leitura, com saltos condicionais entre elas.

- Durante a execução a unificação segue o código em modo de leitura saltando para o código de escrita quando os subtermos precisem de ser criados. Porém, antes de efectuar o salto do código de leitura para o código de escrita, uma variável, chamemos-lhe S, deverá ser actualizada com um identificador que permita que no código de escrita se possa determinar o momento em que se deverá retornar ao código de leitura. Este identificador pode ser um inteiro e deve ser criado de forma a reflectir o nível da profundidade da estrutura em questão.


Código Leitura
Código Escrita
set S=0
set S=1
Jump L
L:
L':
if S==1 Jump L'

Código WAM para a unificação W=f(g(A),h(B))

Instruções WAM	Operações
get_structure f/2,A1	W=(f/2)
unify_variable X4	W.1=U
unify_variable X5	W.2=Z
get_structure g/1,X4	U=(g/1)
unify_value A2	U.1=A
get_structure h/1,X5	Z=(h/1)
unify_value A3	Z.1=B

Código WAM Two-Streams para a unificação W=f(g(A),h(B))

Instruções do modo de leitura	Instruções do modo de escrita
`if var(W) set S=1, jump Lx`	`Lx:`
`W=(f/2)`	`W=(f/2)`
`W.1=U`	`W.1=U`
`W.2=Z`	`W.2=Z`
`if var(U) set S=2, jump Ly`	`Ly:`
`U=(g/1)`	`U=(g/1)`
`U.1=A`	`U.1=A`
`Ly'`	`if S=2 jump Ly'`
`if var(Z) set S=2, jump Lz`	`Lz:`
`Z=(h/1)`	`Z=(h/1)`
`Z.1=B`	`Z.1=B`
`Lz'`	`if S=2 jump Lz'`
`Lx'`	`if S=1 jump Lx'`

Vantagens da WAM Two-Streams

- **Propagação do modo de escrita.** O modo de escrita de um termo é propagado para todos os seus subtermos durante a compilação. Não existem assim desreferenciações, verificações da trilha ou atribuições (*bindings*) desnecessários.

- **Tamanho do código linear.** O número de instruções geradas é cerca do dobro das geradas pela WAM, mas as instruções têm metade (ou menos) da complexidade.

- **Expansão para código nativo mais eficiente.** Ora como o código de cada instrução é especifico para modo de leitura ou escrita, não necessita de ter saltos dentro das instruções `unify` para escolher o modo, pois este é seleccionado pelas instruções `get`, que fazem o salto para o código a ser executado. Saliente-se que um salto é uma operação com penalização muito elevada em termos de execução nas máquinas actuais.

Uma linguagem intermédia de mais baixo nível

- No conjunto de instruções das linguagens intermédias do tipo WAM existe uma série de optimizações que não são possíveis de se realizar.
- Considere-se por exemplo a estrutura `irmão(manuel,tiago).` e o resultado da sua compilação para YAAM:

```
get_struct  irmão
unify_atom  manuel
unify_atom  tiago
```

- O conjunto de instruções WAM fazem com que seja difícil obter código mais optimizado. Sabemos porém que a inicialização do `Heap` feita pela instrução `get_struct` quando entra em modo de escrita é desnecessária, pois as instruções `unify` que se lhe seguem realizam essa operação.
- Esta optimização já pode ser facilmente realizada a nível de uma linguagem intermédia de mais baixo nível constituída por instruções muito simples.

Uma linguagem intermédia de mais baixo nível

- A criação de uma nova linguagem intermédia normalmente pretende:
  - isolar muitas das dependências do compilador em relação à máquina a que se destina.
  - criar uma representação capaz de melhorar as transformações e optimizações que são difíceis (mesmo impossíveis) na linguagem de origem e destino.
- As instruções devem ser o mais independentes possível umas das outras, para se poder alterar a sua ordem, por forma a permitir um maior leque de possíveis optimizações.
- Deve-se também ter em conta as instruções da arquitectura destino, de forma a que a transformação para essa arquitectura seja feita com o menor custo possível, aproveitando ao máximo os seus registos e as suas instruções. **No entanto deve-se tentar não inviabilizar uma possível implementação numa outra máquina.**

YAIL: Yet Another Intermediate Language

A nível de exemplo, neste módulo será apresentada a linguagem intermédia YAIL, usada no protótipo do sistema YAP c/ código nativo.

Instruções que acedem a posições de memória.

Instrução	Significado
`Load(Ra,Rb,disp)`	Ra = *(Rb+disp)
`Store(Ra,Rb,disp)`	*(Rb+disp)=Ra

Instruções simples que trabalham apenas com registos

Instrução	Significado
`test(cond,Ra, label)`	if cond(Ra) jump label
`compare(cond,Ra,Rb,Rc)`	Rc= (Ra cond Rb)
`comparei(cond,Ra n,Rc)`	Rc= (Ra cond n)
`move_cond(cond,Ra,Rb,Rc)`	if cond(Ra) Rc=Rb
`move_condi(cond,Ra,n,Rc)`	if cond(Ra) Rc=n
`set(Ra,Rb,n)`	Ra=Rb+n (n com 16 bits)
`add(Ra,Rb,Rc)`	Ra=Rb+Rc
`addv(Ra,Rb,n)`	Ra=Rb+n
`mul(Ra,Rb,Rc)`	Ra=Rb*Rc
`mulv(Ra,Rb,n)`	Ra=Rb*n
`shift(Ra,Rb,n)`	Ra=Rb$\gg$ n

Instruções complexas

Instrução	Significado
`deref(Ra,Rb)`	Ra fica com o resultado da desreferenciação de Rb
`failz(Ra)`	Se Ra for zero falha
`failnz(Ra)`	Se Ra for diferente de zero falha
`trail(Ra)`	faz o trail de Ra se Ra for menor que `BH` e maior que `B`
`trailglobal(Ra)`	faz o trail de Ra se Ra for menor do que `BH`
`traillocal(Ra)`	faz o trail de Ra se Ra for maior do que `B`
`unify(Ra,Rb)`	unifica as células Ra e Rb
`unifybound(Ra,Rb)`	tenta unificar Ra com Rb e retorna 1 se conseguir
`push(Ra)`	push(Ra)
`pop(Ra)`	pop(Ra)

Instruções relacionadas com o esquema de etiquetas (Tags)

Instrução	Significado
`IsVar(Ra,Rb)`	Retorna 0 em Ra se Rb não for uma variável
`IsAppl(Ra,Rb)`	Retorna 0 em Ra se Rb não for uma estrutura
`IsPair(Ra,Rb)`	Retorna 0 em Ra se Rb não for uma lista
`RepAppl(Ra,Rb)`	Ra toma o valor de Rb sem a etiqueta de estrutura
`RepPair(Ra,Rb)`	Ra toma o valor de Rb sem a etiqueta de lista
`AbsAppl(Ra,Rb)`	Ra toma o valor de Rb mais a etiqueta de estrutura
`AbsPair(Ra,Rb)`	Ra toma o valor de Rb mais a etiqueta de lista

Organização dos registos na YAIL

- A YAIL tem por definição um número ilimitado de registos, contudo na última fase de compilação, depois da optimização, os registos são restringidos aos existentes na máquina destino.
- Os registos na YAIL têm a seguinte organização:
  - `R0`, registo com o valor sempre em zero.
  - `R1` - `R10`, registos de âmbito geral, para controle do estado da execução.
  - `R11`,`R12` - registos temporários, são na maior parte das vezes usados para receber o resultado de operações de desreferenciação.
  - `R13` - registo temporário usado principalmente em comparações.
  - `R14` em diante, registos `X[i]`.

Optimização do código YAIL

- Segue-se a descrição das oito transformações diferentes que podem ser usadas para melhorar código intermédio.
- Para cada transformação é apresentado um simples exemplo.
- Saliente-se que todos estes passos não funcionam isoladamente mas sim como um conjunto.
- Os primeiros cinco passos têm como objectivo reduzir o tamanho do código, eliminando código que nunca chega a ser usado.
- As últimas três fases da optimização concentram-se no facto de que algumas instruções podem usar o valor de um registo mais um inteiro como parâmetro, eliminando assim a necessidade de efectuar certas atribuições.

Eliminação de instruções `label`

```
<...>                          <...>
labely:                        labelx:
labelx:            --->        <...>
<...>                          jump labelx
jump labely
```

Eliminação de código morto

```
<...>                                  <...>
jump labelx                            jump labelx
<ins. diferentes de label>             labely:
labely:                                <...>
<...>                        --->      <...>
labelw:                                labelw:
addv Rx,Rx,0                           <...>
<...>                                  <...>
```

Redução de saltos

```
<...>                          <...>
labely:                        <...>
jump labelx                    jump labelx
<...>              --->        <...>
jump labely                    jump labelx
<...>                          <...>
```

Eliminação de saltos

```
<...>                          <...>
jump labelx        --->        <...>
labelx:                        labelx:
<...>                          <...>
```

Substituição de instruções `jump`

```
<...>                          <...>
<...>                          set Rx,Ry,m
jump labelx                    jump labely
<...>            --->          <...>
labelx:                        labelx:
set Rx,Ry,m                    set Rx,Ry,m
jump labely                    jump labely
<...>                          <...>
```

Eliminação de atribuições não usadas

```
<...>                                  <...>
set Rn,Ra,3                  --->      <...>
<ins. que não usam Rn>                 <sem alteração>
set Rn,Rb,m                            set Rn,Rb,m
<...>                                  <...>
```

Redução de atribuições para o mesmo registo

```
<...>                                   <...>
set Rx,Rw,n1                            <...>
set Ry,Rx,m                  --->       set Ry,Rw,m+n1
set Rx,Rx,n2                            set Rx,Rw,n1+n2
<...>                                   <...>
```

Alteração da ordem das instruções

```
<...>                          --->      <...>
set Rx,Ry,n                              store Rw,Ry,n+m
store Rw,Rx,m                            set Rx,Ry,n
<...>                                    <...>
```

Exemplo: código YAIL não Optimizado

```
WAM Two-Streams  |  Código YAIL s/ opt.
                 |
...              \
unifyListWrite    > set   R_S,R_H,0
                 /  set   R_H,R_H,2
                 \  addv  R_T,R_Z,valor_1
unifyAtomWrite 1  > store R_T,R_S,0
                 /  set   R_S,R_S,1
                 \  addv  R_T,R_Z,valor_[]
unifyAtomWrite [] > store R_T,R_S,0
                 /  set   R_S,R_S,1
```

Exemplo: código YAIL Optimizado

```
WAM Two-Streams  |  Código YAIL s/ opt.      | YAIL Optmizado
                 |                          |
...              \                          |
unifyListWrite    > set   R_S,R_H,0         | addv  R_T,R_Z,valor_1
                 /  set   R_H,R_H,2         | store R_T,R_H,0
                 \  addv  R_T,R_Z,valor_1   | addv  R_T,R_Z,valor_[]
unifyAtomWrite 1  > store R_T,R_S,0         | store R_T,R_H,1
                 /  set   R_S,R_S,1         | set   R_S,R_H,2
                 \  addv  R_T,R_Z,valor_[]  | set   R_H,R_H,2
unifyAtomWrite [] > store R_T,R_S,0         |
                 /  set   R_S,R_S,1         |
```

Compilação e utilização do código assembly gerado

- Depois de optimizar o código YAIL, é necessário transformar o código em assembly da máquina destino...
- Depois de compiladas todas as cláusulas do programa em Prolog para código assembly, torna-se necessário invocar o assemblador para esse código de forma a ser possível executar o código nativo dentro do Sistema de Prolog.
- O ficheiro com o código nativo é compilado para um ficheiro objecto, que é carregado dinamicamente (durante a execução) para a memória usando a função `dlopen`

```
system("gcc code.c -c -fpic");
system("ld codigo.o -o code.so -shared -expect_unresolved "*" ");
handle=dlopen("./code.so",RTLD_NOW);
endereço=dlsym(handle,"funcao");
```

O endereço da nova função é guardado na variável `endereço`.

Compilação e utilização do código assembly gerado

- O código da função fica disponível para ser chamado a partir de qualquer área do sistema Prolog, como se de uma função já implementada se tratasse. Basta para tal invocar o comando `(*endereço)(args);`.

- Para que a máquina abstracta do sistema Prolog possa usar o código nativo é ainda necessário:

  - criar uma nova instrução no emulador que estabeleça a ligação com o código nativo `RunNative clausula`.
  - durante a compilação de um programa Prolog, o código WAM de cada cláusula deverá ser substituído por essa instrução.

Código das Cláusulas Pi para a máquina abstracta

P1

Run_Native 1

P2

Run_Native 2

P3

Run_Native 3

Código Nativo

```
funcao(int n)
{    labels[]={...}
<...>
goto labels[n];
label1:
        código de P1
            <...>
         return;
label2:
        código de P2
            <...>
          return;
label 3:
        código de P3
            <...>

          return;
```

Desvantagens do Código Nativo

- menor portabilidade
- compilador mais complexo
- compilador mais lento
- tamanho do código nativo pode crescer demasiado