

Practical Session E4

Metaprogramming in Prolog *Prolog (3)*

a.a. 2011/2012

Stefano Benedettini

Outline

- What we do in this lab lesson is to build interpreters for simple languages, whose syntax is basically Prolog plus new unary/binary operators
- Part 1: an example for you to analyse
- Part 2: an extension for you to implement on a step-by-step basis
- Part 3: a new language (do/finish at home)

Part 1.1

```
:-op(100,xfx,':').  
:-op(90,fx,add).  
:-op(90,fx,sub).
```

```
1: add 5.  
2: sub 1.  
3: add 1.
```

```
execute(RI,RO) :- exec(1,RI,RO).  
exec(N,RI,RO):-clause(':',(N,add(V)),true),!,  
                RO1 is RI+V, N1 is N+1, exec(N1,RO1,RO).  
exec(N,RI,RO):-clause(':',(N,sub(V)),true),!,  
                RO1 is RI-V, N1 is N+1, exec(N1,RO1,RO).  
exec(_,R,R).
```

- Look at this code and try to execute:
 - execute(10,X).

Description

- The first three lines include directives to specify syntactic aspects of the language
 - The language is an assembler-like language with one register and operations to add and subtract an integer to the register
- The 1,2,3 lines are the current program
- execute/2 is an I/O executor, namely, an interpreter for the language
- See the actual Prolog representation of the program with goal:
 - `clause(':',(X,Y),Z).`

Part 1.2

```
code {  
    1: add 5;  
    2: sub 1;  
    3: add 1  
}.
```

- We want to support a slight change of the language, which is now represented by a single clause
- Look and analyse the interpreter in the next page

Interpreter (1.2)

```
% parsing
:-op(100,fx,code).
:-op(100,xfx,':').
:-op(90,fx,add).
:-op(90,fx,sub).
% execution
execute(RI,RO) :- exec(1,RI,RO).
instruction(N,I):-clause(code({L}),true),search(L,N,I).
search(';')(A,_),N,I):-search(A,N,I),!.
search(';')(_,B),N,I):-search(B,N,I).
search(':')(N,I),N,I).
exec(N,RI,RO):-instruction(N,add(V)),!,
                RO1 is RI+V, N1 is N+1, exec(N1,RO1,RO).
exec(N,RI,RO):-instruction(N,sub(V)),!,
                RO1 is RI-V, N1 is N+1, exec(N1,RO1,RO).
exec(_,R,R).
```

Part 2: A New Language

```
code {  
    1 : dec r0;  
    2 : inc r1;  
    3 : jmpifzero r0 towards 1;  
    4 : add r0 to r1;  
    5 : sub r0 to r1  
}.
```

- the input is a list of integers, that will fill a set of registers (r0,r1,r2,...)
- e.g: if the input list is **(1,4,3,7,10)**, it means that, before starting the program, the state of registers is initialized to: **r0=1, r1=4, r2=3, r3=7, r4=10**.
- When an operation is executed on a register, the corresponding element of the input list will be modified: e.g. **inc r0** leads to list **(2,4,3,7,10)**
- **add r0 to r1** means " $r1 = r1 + r0$ "
- **jmpifzero r0 towards N** means "*if $r0=0$ goto instruction N*"

Do not implement it yet!

2.1 Reading from file

- Desiderata:
 - `load('C:/code.prg'),execute(10,X).`
 - **Important:** use `\\` or `/` as path separator!
- Predicate `load(+File)`
 - drop any clause of the kind `code(X)`
 - open file `File`
 - get one term `code(X)` from it
 - close the file
 - assert the new program
- From now, always execute program on files
- Implementation in the next slide!

2.1 Predicate Load

```
load(F):-retractall(code(X)),      % delete an existing prg.  
        see(F),                   % opens the file  
        read(PRG),                % reads one term  
        seen,                     % closes the file  
        assert(PRG).              % stores the prg.
```

- Take a look at predicates `see/1`, `seen`, `read/1` on the tuProlog User's Guide
- Test the behavior of `load` by loading a file containing the term **`simpleterm(1)`**.

2.2 Handling registers

- Implement a `register/2` predicate for the new language
- Desiderata:
 - `register(r5,N).` → `N/5`
- Suggestion. Use the following predicates
 - **`atom_codes(?atom,?list_of_codes)`** → where ***atom*** is a Prolog atom and ***list_of_codes*** the corresponding sorted list of codes of the chars forming ***atom***.
 - **`num_atom(?number,?atom)`** → where ***number*** is a string representing a number and ***atom*** its corresponding representation as a Prolog atom.
 - See next page for more suggestions →

2.2 Handling registers

- (More) suggestions
 - **register(r5,N)** has to return **N/5**, not N/'5' (i.e. the string representation of Prolog constant **5**).
 - Apply two times **atom_codes/2**: the first to obtain the list of codes corresponding to the numeric part of the register name; the second to obtain the string representation of the codes (note that **atom_codes(N,[49,50])** → **N/'12'**, not **N/12**).
 - Finally, use **num_atom/2** to obtain the number as a Prolog constant given its representation as a string.

2.3 Accessing memory

- Implement a predicate `access/5` for the new language
 - `access(+list,+pos,-elem,+newelem,-newlist)`
- Desiderata:
 - `access([10,20,30],0,X,_,_).` $\rightarrow$ `X/10`
 - `access([10,20,30],2,X,_,_).` $\rightarrow$ `X/30`
 - `access([10,20,30],0,_,11,L).` $\rightarrow$ `L/[11,20,30]`
 - `access([10,20,30],2,_,31,L).` $\rightarrow$ `L/[10,20,31]`
 - `access([10,20,30],4,_,_,_).` $\rightarrow$ `no`
- Its implementation needs 2 clauses only

2.4 only dec and inc

```
code {  
    1 : dec r0;  
    2 : inc r1  
}.
```

```
?- load("C:/code.prg"),execute([10,20],L). -> L/[9,21]
```

- By modifying the language in part 1, support only the version with the two above instructions

2.5 support the entire lang

```
code {  
    1 : jmpifzero r0 towards 3;  
    2 : sub r1 to r1;  
    3 : add r1 to r1  
}.
```

```
?- load("C:/code.prg"),execute([0,10],L). -> L/[0,20]  
?- load("C:/code.prg"),execute([1,10],L). -> L/[1,0]
```

- Suggestion, add a new instruction at a time

Part 3: Additional Exercise

```
function prod(v1,v2) do {  
    variable v3 <- 0;  
    whilenotzero(v1) do {  
        v1 <- v1-1;  
        v3 <- v3+v2  
    };  
    output v3  
}.
```

```
?-execute(prod(10,20),X). -> X/200
```

- Interpreting this language is now a mere prolog exercise

3.1 Matching

- Implement a match/3 predicate
- Desiderata:
 - `match(prod(v1,v2),prod(10,20),L).`
 - `L/[var(v1,10),var(v2,30)]`
- Suggestion:
 - use predicate `=../2` seen at lesson
 - `prod(v1,v2) =.. L .` → `L/[prod,v1,v2]`

3.2 Parsing

```
: -op(800,xfx,'<-').  
: -op(45,fx,function).  
: -op(50,xfx,do).  
: -op(40,fx,[variable,whilenotzero,output]).
```

- Check if it works and how it represents programs
- We used priority 800 to handle existing operators for + and - properly

3.3 Execution

- When the function is invoked, the memory is prepared
 - it is a list with elements $\text{var}(V, N)$
- Its body is executed
 - the body is of the kind $I;B$, where I is the first instruction and B is the rest
- Instructions:
 - variable creates a new element in the memory
 - $\leftarrow$ evaluates the right-expression and updates the memory

3.4 While

- How can we support while construct?
- When the program is
 - `whilenotzero(V) do B1;B`
- proceed as follows:
 - if V is zero proceed by executing B
 - otherwise, proceed by executing
 - `B1;whilenotzero(V) do B1;B`

3.5 Evaluating expressions

- Just support simple expressions
 - X , $'+'(X,Y)$, $'-'(X,Y)$
- Where X and Y are either
 - numbers
 - variable identifiers (in this case you should check what the current value of the variable is)

3.5 Evaluating Functions

- Now an expression can be a function call
 - `prod(x,y)`, `prod(x,prod(x, x))`, `prod(10,x+y)`, ...
- Reuse the execute predicate
- Argument variables must be evaluated beforehand