Defining
Binary & Unary
Operators

English-French Dictionary

◊ Can use compound terms to represent a dictionary

> **list is a structure that contains an entry followed by the rest of the list**

> **For example**

list (entry (book , livre) ,

list (entry (man , homme) ,

list (entry (apple , pomme) ,

empty)))

◊ Illustrates how compound terms can be used

English-French Dictionary – 2

◊ Define a custom member function for the list structure

member (X , list (X , _)).

member (X , list (_ , L)) :- member (X , L).

English-French Dictionary – 3

◊ Here is a predicate that defines the correspondence between English and French words.

```
englishFrench1( English , French ) :-
    member ( entry ( English , French ) ,
        list ( entry ( book , livre ) ,
            list ( entry ( man , homme ) ,
                list ( entry ( apple , pomme ) ,
                    empty ) ) ) )
```

English-French Using Standard Lists

◊   We could use the standard list structure.

> **The standard member predicate**

member (X , [X | _]).

member (X , [_ | R]) :- member (X , R).

> **The translation predicate**

englishFrench2 (English , French) :-

member (entry (English , French),

[entry (book , livre) ,

entry (man , homme),

entry (apple , pomme)]).

English-French Different Dictionaries

◊ We could change the rule to use a dictionary that holds the list structure

> **It is easier to understand the rule**

englishFrench3 (English , French , Name) :-

   dictionary (Name , Dictionary) ,

   member (entry (English , French) , Dictionary)

> **where we have a fact defining the dictionary. It is easier to change the dictionary and to use it in other contexts**

Different Dictionaries

Dictionary (Name , D) :-

   Name = d1 , D = [entry (book , livre) ,

                      entry (man , homme) ,

                      entry (apple , pomme)] ;

   Name = d2 , D = [entry (book , koob) ,

                      entry (man , nam) ,

                      entry (apple , elppa)] .

Use an infix member function

◊ The previous definition is not a natural way of representing the member function

◊ A more "natural" use of member is as an infix operator, as in the following

> **Use the letter e to represent the mathematical symbol belongs-to ($\in$)**

englishFrench4 (English , French) :-

entry (English , French) e [entry (book , livre) ,

entry (man , homme) ,

entry (apple , pomme)

] .

Use an infix member function

◊ The infix operator e can be defined as follows

:- **op (500 , xfy , [e]).**

> **Later slides describe the meaning of the op predicate**

◊ **e** is a new operator (predicate) so we must create rules that define what it means

> **Since e is defined to be infix its rules use infix syntax**

> **Note the similarity with the definition of the member predicate**

X e [X | _].

X e [_ | L] :- X e L .

Use an infix member function – 3

◊ We can chose of the name of the operator

```
:-  op ( 500 , xfy , [ belongs_to ] ).

X belongs_to [ X | _ ].
X belongs_to [ _ | L ]   :-   X belongs_to L .

englishFrench5 ( English , French ) :-
    entry ( English , French )
        belongs_to
            [ entry ( book , livre ) ,
              entry ( man , homme ) ,
              entry ( apple , pomme )
            ].
```

Bird – Mammal example

◊ Define some properties of animals

> **Use syntax that is similar to natural language**

:- op (100 , xfx , [has , isa , flies]).

Animal has hair :- Animal isa mammal.

Animal has feathers :- Animal isa bird.

owl isa bird.

cat isa mammal.

dog isa mammal.

Example with mulitple precedence

◊ Plays and "and" are at different precedence levels.

◊ Define

      :- op (300 , xfx , plays).

      :- op (200 , xfy , and).

◊ Example use

      Term1 = jimmy plays football and squash.

      Term2 = susan plays tennis and basketball
                 and volleyball.

Example with mulitple precedence – 2

◊ What is the internal stucture when using operators as in the following?

Term1 = jimmy plays football and squash.

Term2 = susan plays tennis and basketball and volleyball.

◊ Recall that everything within Prolog is represented with compound terms, so we have ...

Term1 = plays (jimmy , and (football , squash))

Term2 = plays (susan , and (tennis , and (basketball , volleyball)))

Example with mulitple precedence – 3

◊ DeMorgan's law – make predicate syntax look similar to standard mathematics

    :- **op (800 , xfx , <==>).**
    :- **op (700 , xfy , v).**
    :- **op (600 , xfy , &).**
    :- **op (500 , fy , ~).**

◊ Consider representing the following

    ~ (A & B) <==> ~A v ~B .   **Uses the above**

◊ In standard Prolog, this could be represented as

    equivalence (not (and (A , B)) ,
                    or (not (A) , not (B))).

    > **or, directly use the internal form**

    '<==>' ('~' ('&' (A , B)) , 'v ' ('~' (A) , '~' (B))).

Why have operators?

◊ Introduce operators to improve the readability of predicates

　　» **Can be infix, prefix or postfix**

◊ Operator definitions do not define any action, they only introduce new notation

　　» **Operators are functors that hold together the components of compound terms or structures**

◊ A programmer can define their own operators

　　» **with their own precedence and associativity**

　　» **programmer defined operators can be merged in precedence and associativity with the Prolog builtin operators**

op Predicate

◊ Define one or more operators with a given precedence, associativity

op (precedence ,
associativity ,
symbol or symbol list
)

◊ Bratko page 77 gives a listing of the precedence of some of the standard operators in Prolog

op Precedence component

◊ Precedence

  » **between 0 and 1200 – the precedence class**

  » **lower class numbers have higher priority**

  » **higher priority implies do first**

  » **Example**

    3 + 4 * 5 = 3 + (4 * 5)

  » *** (precedence class 400) has lower number than + (precedence class 500) so times is done first**

  » **Can always use () to force the order of using operators**

    > **Useful when you do not know relative precedence or to make it clear to the reader**

Expression Precedence Class

◊ Precedence class of base operand is 0.

◊ Precedence class of expression with operator, oper, is the precedence class of oper

op Associativity component

◊ Associativity

   » **Defines which operands belong to which operator when several operators are used in sequence**

   » **For example in the following**

   A oper B

   > **is oper a unary operator with operand A**
     is oper a unary operator with operand B
     is oper a binary operator with operands A and B

◊ Can define oper as unary operator with ...

   op (100 , fy , oper).   **-- unary prefix**
   op (100 , fx , oper).   **-- unary prefix**
   op (100 , xf , oper).   **-- unary postfix**
   op (100 , yf , oper).   **-- unary postfix**

Unary prefix associativity

◊   f y

     oper oper a .   **-- legal syntax**

       > **oper a has equal precedence class with oper**

       > **y says operand of oper can have lower or equal precedence class**

◊   f x

     oper oper a.   **-- illegal syntax**

       > **oper a has equal precedence class with oper**

       > **x says operand of oper must have lower precedence class**

       > **must use ( ) as follows**

     oper (oper a) .

Unary postfix associativity

◊ y f

    a oper oper .   -- legal syntax

      > a oper has equal precedence class with oper

      > y says operand of oper can have lower or equal class

◊ x f

    a oper oper .   -- illegal syntax

      > a oper has equal precedence class with oper

      > x says operand of oper must have lower precedence class

      > must use ( )

    (a oper) oper .

op Associativity component – 2

◊ Given

A oper B

◊ Can define oper as a binary operator with ...

op (100 , xfy , oper). **-- right associative**

op (100 , yfx , oper). **-- left associative**

op (100 , xfx , oper). **-- evaluate both operands first**

op (100 , yfy , oper). **-- not defined, ambiguous**

Right associative operator

◊ Define

    :- op (100 , xfy , op1).

◊ Test

    > **C becomes the full structure, L shows the substructure**

    C = 1 op1 2 op1 3 op1 4 , C =.. L.

◊ Result

    C = 1 op1 2 op1 3 op1 4

    L = [op1 , 1 ,   2 op1 3 op1 4]

    > **Left most op1 is evaluated last**

    > **Apply recursively**

Left associative operator

◊ Define

    :- op (200 , yfx , op2).

◊ Test

    > **C becomes the full structure, L shows the substructure**

    C = 1 op2 2 op2 3 op2 4 , C =.. L.

◊ Result

    C = 1 op2 2 op2 3 op2 4

    L = [op2 , 1 op2 2 op2 3 , 4]

    > **Right most op2 is evaluated last**

    > **Apply recursively**

Evaluate both operands first

◊ Define

:- op (300 , xfx , op3).

◊ Test

C = 1 op3 2 op3 3 op3 4 , C =.. L.

◊ Result

C = 1 op3 2

« Syntax Error - check operator precedences » op3 3 op3 4 , C =.. L.

> Error because the middle op3 expects its operands to its left and right to have lower precedence class but they have equal precedence class

© Gunnar Gotshalks

Evaluate both operands first – 2

◊ Define

:- op (300 , xfx , op3).

◊ Test – with different operators to left and right of op3

C = 1 op1 2 op3 3 op2 4 , C =.. L.

◊ Result

C = 1 op1 2 op3 3 op2 4

L = [op3 , 1 op1 2 , 3 op2 4]

> **op1 and op2 are done first (higher priority, lower precedence class)**

> **op3 is done last**