Logic Programming

Sukree Sinthupinyo

Outline

- Introduction to Prolog
- Knowledge representation
- Prolog’s search strategy

Introduction to Prolog

- PROgramming in LOGic.
- Prolog is based on First-Order logic.
- Prolog is known to be a difficult language to master.
  - The system does not give too much help to the programmer to employ structured programming concepts.

Introduction to Prolog *(cont')*

- The special attention should be given to understanding variables in Prolog.
- The programmer should pay careful attention to the issue of backtracking.

Knowledge Representation

- How do we represent what we know?
  - Knowledge how *procedural knowledge such as how to drive a car*.
  - Knowledge that *declarative knowledge such as knowing the speed limit for a car on a motorway.*

Knowledge Representation *(cont')*

- Propositional calculus.
  - The propositional calculus is based on statements which have truth values (true or false).
  - An example:
    - If p stands for "fred is rich" and q for "fred is tall".
    - Connection symbol $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$, $\neg$

Knowledge Representation *(cont')*

- First order predicate calculus.
  - Permits the description of relations and the use of variables.
  - Variables: `X,Y,Z`
  - Constant: `x,y,john,jean`
  - Predicate: `brother(x,y)`
  - Formulae: `brother(x,y) ^brother(y,z)`
  - Sentence: `brother(john,jean)`

We turn to Prolog

- “the capital of france is paris“
  - `has_capital(france,paris).`
- “jim is tall”
  - `jim(tall).`
  - `tall(jim).`
- There must be no space between the predicate name and “(” . The whole thing also ends in a “.”

Prolog Constants

- Constant is a string of characters starting with a lower case letter.
  - `loves(jane,jim).`
  - `jane` and `jim` are constant.
- No predicate may be variable
  - `X(jane,jim).`

Examples of statement in Prolog form

- bill likes ice-cream.
- bill is tall
- jane hits jimmy with the cricket bat
- john travels to london by train

Multiple Clauses

- Examples
  - bill only eats chocolate, bananas or cheese.
  - the square root of 16 is 4 or -4.
  - wales, ireland and scotland are all countries.

Rules

- The format is:

```
divisible_by_two:- even.
```

  - This is a non-unit clause.
  - The head is `divisible_by_two`.
  - The body is `even`.
  - `divisible_by_two` is **true** if even is **true**

Rules *(continue)*

- No more than one goal is allowed in the head.
- We cannot translate `rich(fred) ⇒ happy(fred) ∧ powerful(fred)` directly into the Prolog version `happy(fred),powerful(fred) :- rich(fred).`

Rules *(continue)*

- A number is divisible by two if it is even.
- We can express this with the help of the logical variable. Here is the improved rule:
- `divisible_by_two(X):- even(X).`

Semantics

ถ้า x เป็นคู่แล้วจะ | 2 ลงตัว

header rule

body rule

- `divisible_by_two(X):- even(X).`
- If we can find a value of X that satisfies the goal `even(X)` then we have also found a number that satisfies the goal `divisible_by_two(X).`

The Logical Variable

- If an object is referred to by a name starting with a capital letter then the object has the status of a *logical variable*.
- The same variables in one rule refer to the same object.

The Logical Variable *(continue)*

- The logical variable cannot be overwritten with a new value.
- For example, in Pascal:
  - `X:= 1; X:= 2;`
- In Prolog:
  - `X=1 ∧ X=2.`
  - cannot be true as X cannot be both ‘2’ and ‘1’ simultaneously. An attempt to make a logical variable take a new value will fail.

Rules and Conjunction

- A man is happy if he is rich and famous
- might translate to:

```
happy(Person):-
    man(Person),
    rich(Person),
    famous(Person).
```

Rules and Disjunctions

- Someone is happy if they are healthy, wealthy or wise.
- translates to:

```
happy(Person):- healthy(Person).
happy(Person):- wealthy(Person).
happy(Person):- wise(Person).
```

Both Disjunctions and Conjunctions

ถ้า

```
happy(Person):-healthy(Person),woman(Person).
happy(Person):-wealthy(Person),woman(Person).
happy(Person):-wise(Person),woman(Person).
```

Query ด้วย ?

Prolog's Search Strategy

- Prolog use *depth-first search.*
- Prolog enables the programmer to implement other search methods quite easily.
- Prolog is an interactive system.

Queries and Disjunctions

- A query is a goal which is submitted to Prolog in order to determine whether this goal is true or false.
- Prolog normally expects queries it prints the prompt:

?–

Queries and Disjunctions *(continue)*

- Perhaps we would like to determine whether or not

```
woman(jane)
```

- We can type this

```
?- woman(jane).
```

A Simple Conjunction

Program Database

```
woman(jean).
man(fred).
wealthy(fred).
happy(Person):- woman(Person),
                wealthy(Person).
```

A Simple Conjunction *(continue)*

- From the knowledge base, if we would like to find “Is Jean happy?”.
- Prolog try to match `happy(jean)` with the knowledge base.

```
happy(jean):- woman(jean),
              wealthy(jean).
```

A Simple Conjunction *(continue)*

- So two sub goals are:
  - `woman(jean)`
  - `wealthy(jean)`
- There is a possible match but we cannot *unify*
  - `wealthy(fred)` with `wealthy(jean)`

Unification

- Unification is a two way matching process
  - `book(waverley,X)` and `book(Y,scott)`
  - `X/scott, Y/waverley`
- Examples
  - `X=fred.      succeeds`
  - `jane=fred      fail`
  - `Y=fred,X=Y     succeeds`
  - `X=happy(jim)  succeeds`

Recursion

- Prolog depends heavily upon it
- Examples
  - One of my ancestors is one of my parents or one of their ancestors.
  - A string of characters is a single character or a single character followed by a string of characters

ancestor(bob,X) = ? bob เป็นบรรพบุรุษใครบ้าง

Recursion *(continue)*

- Examples

  X เป็นบรรพบุรุษ Y

  - ```
    ancestor(X,Y) :- parent(X,Y).
    ancestor(X,Y) :- parent(X,Z),
                     ancestor(Z,Y).
    ```
    V
  - Program Database

    ```
    ancestor(X,Y) :- parent(X,Y).
    ancestor(X,Y) :- parent(X,Z),
                     ancestor(Z,Y).
    parent(bob,bill).   parent(bill,jane).
    parent(jane,jim).   parent(bob,babe).
    ```
    ```

Lists

- Lists can be regarded as special Prolog structures.
- Lists can be used to represent an ordered sequence of Prolog terms.
- Examples
  - `[ice_cream,coffee,chocolate]`
  - `[a,b,c,c,d,e]`

Lists *(continue)*

- List destruction
  - `[X|Y] = [f,r,e,d].`
    - will result in `X=f`.
      - The first element of the list is known as the HEAD of the list.
    - `Y=[r,e,d].`
      - The list formed by deleting the head is the TAIL of the list.

Lists *(continue)*

- List construction
  - `X=[r,e,d].`
  - `Result_Wanted=[b|X].`
  - will result in `Result_Wanted=[b,r,e,d].`

Lists *(continue)*

- Bigger Chunks:
  - Suppose you want to stick the elements `a`, `b` and `c` onto the front of the list `X` to make a new list `Y`.
  - This can be done with `Y=[a,b,c|X]`

Lists *(continue)*

- Bigger Chunks:
  - Conversely, suppose you want to take three elements off the front of a list `X`.
  - This can be done with `X=[A,B,C|Y]`.
  - `Y` is available for use as the remaining list.

Lists *(continue)*

- Some Possible Matches
  1. `[b,a,d]=[d,a,b]` fails
  2. `[X]=[b,a,d]` fails
  3. `[X|Y]=[he,is,a,cat]` succeeds
  4. `[X,Y|Z]=[a,b,c,d]` succeeds
  5. `[X|Y]=[]` fails
  6. `[X|Y]=[[a,[b,c]],d]` succeeds
  7. `[X|Y]=[a]` succeeds

Lists *(continue)*

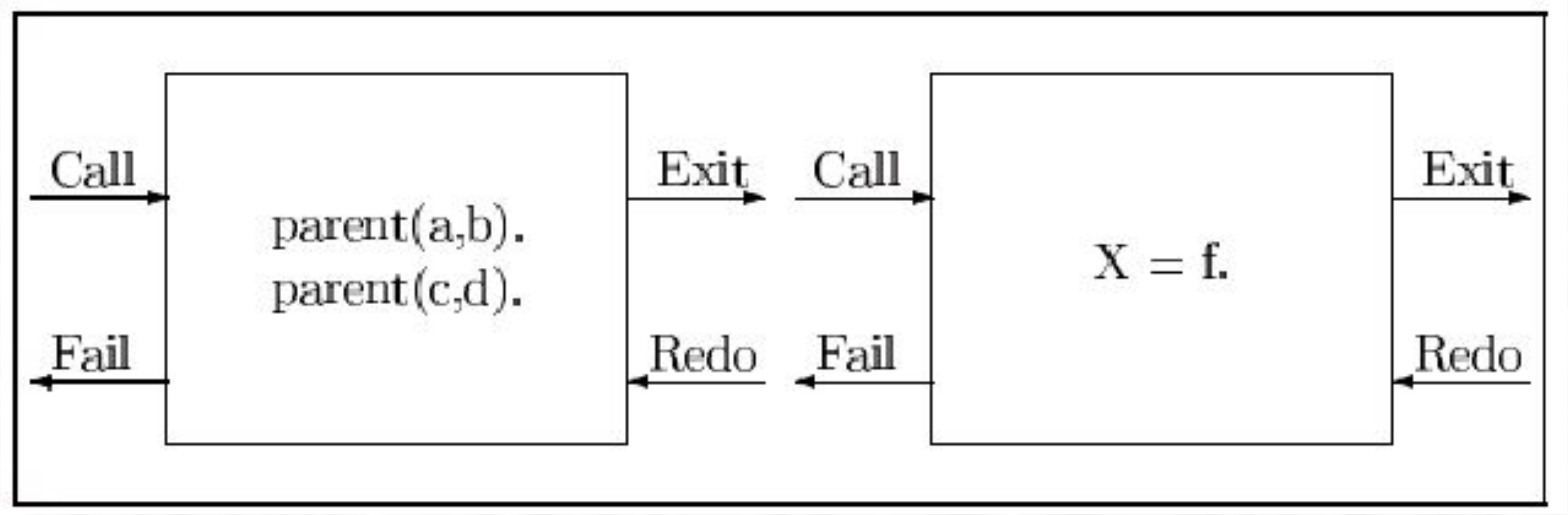
- A recursive program using list

```
print_a_list([]).
print_a_list([H|T]):- write(H),
                      print_a_list(T).
```

- `write/1` is a build in predicate, the argument is Prolog term.
- `nl/0` is new line command

The Box Model of Execution

- We can think in term of *procedure* rather than *predicate.*



The Box Model of Execution

(continue)

- The control flows into the box through the *Call* port.
- Then, Prolog seek a clause with a **head** that unify with the goal.
- Seek solutions to all the subgoals in the body of the successful clause.
- If the unification fails for all clauses then control would pass out of the *Fail* port.

The Box Model of Execution

(continue)

- Control reaches the *Exit* port if the procedure succeeds.
- The *Redo* port can only be reached if the procedure call has been successful and some subsequent goal has failed.

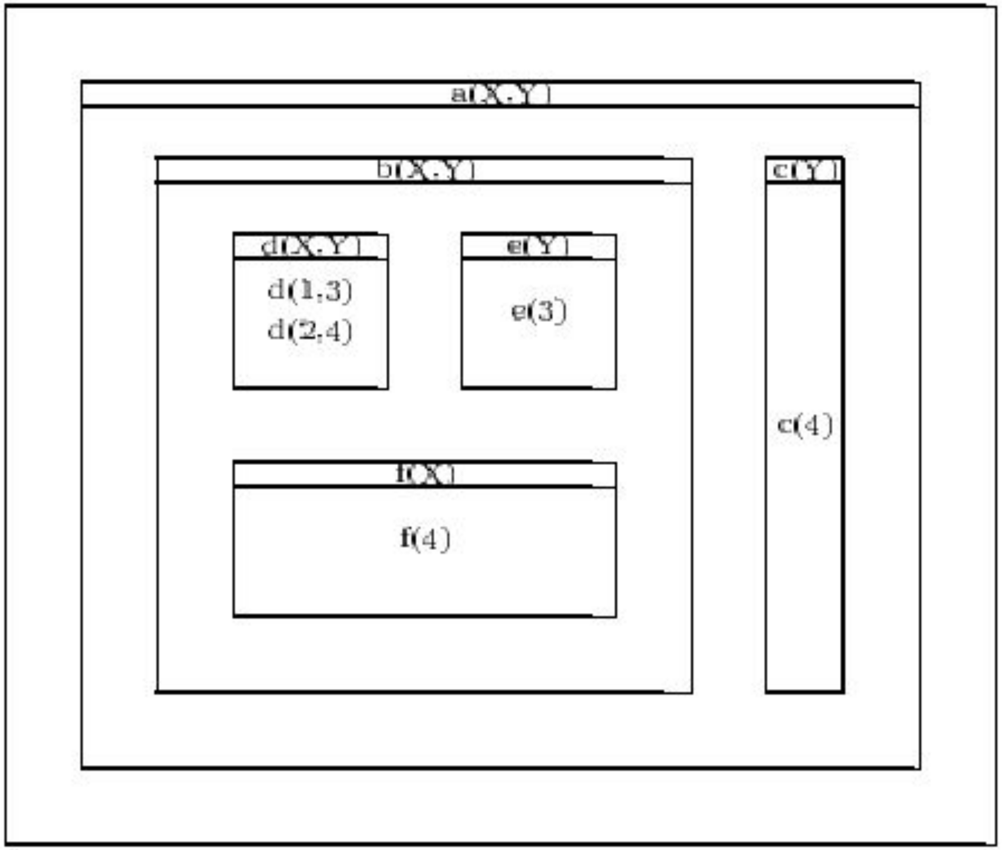
The Flow of Control

```
Call:   parent(X,Y)
Exit:   parent(a,b)
                                          Call:   a=f
                                          Fail:   a=f
                        Now backtracking
Redo:   parent(X,Y)
Exit:   parent(c,d)
                                          Call:   c=f
                                          Fail:   c=f
                        Now backtracking
Redo:   parent(X,Y)
Fail:   parent(X,Y)
```

The Flow of Control *(continue)*

Program Database

```
a(X,Y):-
        b(X,Y),
        c(Y).
b(X,Y):-
        d(X,Y),
        e(Y).
b(X,Y):-
        f(X).
c(4).
d(1,3).
d(2,4).
e(3).
f(4).
```



The Flow of Control *(continue)*

Call:	a(X,Y)				
		Call:	b(X,Y)		
				Call:	d(X,Y)
				Exit:	d(1,3)
				Call:	e(3)
				Exit:	e(3)
		Exit:	b(1,3)		
		Call:	c(3)		
		Fail:	c(3)		
			Now backtracking		
		Redo:	b(X,Y)		
				Redo:	e(3)
				Fail:	e(3)
				Redo:	d(X,Y)
				Exit:	d(2,4)
				Call:	e(4)
				Fail:	e(4)
			Now backtracking		
				Call:	f(X)
				Exit:	f(4)
		Exit:	b(4,Y)		
		Call:	c(Y)		
		Exit:	c(4)		
Exit:	a(4,4)				

Practical Matter

- Exit Prolog by the following commands
  - `?- halt.`
  - `^D`
  - `end_of_file.`
- Loading file(s)
  - `?- consult(filename).`
    - `?- consult(‘test.pl’).`
    - `?- consult(‘test’).`
    - `?- consult(test).`

Practical Matter *(continue)*

- Undeclared predicates could be handled by
  - `:- dynamic` *`predicate/arity`*`.`
    - `:- dynamic foo/1.`
    - `:- dynamic two_hand/1.`
  - `?- unknown(X,Y).`
    - `?- unknown(X,trace).`
    - `?- unknown(X,fail).`

Practical Matter *(continue)*

- Singleton variable
  - A singleton variable occurs in a clause if a variable is mentioned once and once only.
    - `mammal(Animal):- dog(Aminal).`
    - `Warning: singleton variable Animal in procedure mammal/1`
    - `Warning: singleton variable Aminal in procedure mammal/1`

Practical Matter *(continue)*

- Singleton variable
  - You can use the *anonymous* variable.
  - The *anonymous* variables can be written by the string starting with the underscore (_).
    - `member(X,[X|_]).`

Programming Techniques and Lists Processing

- The reversibility of Prolog program

```
Program Database
square(1,1).
square(2,4).
square(3,9).
square(4,16).
square(5,25).
square(6,36).
```

Programming Techniques and Lists Processing *(continue)*

- We can ask all the following queries.

```
?- square(2,X).
?- square(X,5).
?- square(X,Y).
?- square(2,3).
```

Programming Techniques and Lists Processing *(continue)*

- Evaluation in Prolog
  - Pascal: Y:=2+1;
  - Prolog: Y=2+1.
    - This command only binds the variable Y with the clause 2+1 and gives the value of Y as 2+1
  - Prolog: Y is 2+1.
    - After the above command, the value of Y is 3.

Programming Techniques and Lists Processing *(continue)*

- `successor(X,Y):- Y is X + 1.`
  - `?- successor(3,X).`
  - `?- successor(X,4).`
  - `?- successor(X,Y).`
  - `?- successor(3,5).`

Programming Techniques and Lists Processing *(continue)*

- Calling patterns
  - For any given predicate with arity greater than 0, each argument may be intended to have one of three calling patterns:
    - Input |indicated with a +
    - Output |indicated with a -
    - Indeterminate |indicated with a ? (+ or -)

Programming Techniques and Lists Processing *(continue)*

- For example, `successor/2` above requires a calling pattern of
  - 1st argument must be `+`
  - 2nd argument can be `+` or `-` and is therefore `?`
  - We write this as
    - `:- mode successor(+,?).`

Lists Processing

- Program Patterns
  - Test for Existence
    - `list_existence_test(Info,[Head|Tail]):-`
      `element_has_property(Info,Head).`
    - `list_existence_test(Info,[Head|Tail]):-`
      `list_existence_test(Info,Tail).`
  - Example
    - `nested_list([Head|Tail]):- sublist(Head).`
    - `nested_list([Head|Tail]):- nested_list(Tail).`
    - `sublist([]).`
    - `sublist([Head|Tail]).`

Lists Processing *(continue)*

- Example
  - `member(Element,[Element|Tail]).`
  - `member(Element,[Head|Tail]):-`
    `member(Element,Tail).`
- Test all elements
  - `test_all_have_property(Info,[]).`
  - `test_all_have_property(Info,[Head|Tail]):-`
    `element_has_property(Info,Head),`
    `test_all_have_property(Info,Tail).`

Lists Processing *(continue)*

- Example
  ```
  all_digits([]).
  all_digits([Head|Tail]):-
        member(Head,[0,1,2,3,4,5,6,7,8,9]),
        all_digits(Tail).
  ```
- Return a Result – Having processed one element
  ```
  return_after_event(Info,[H|T],Result):-
        property(Info,H),
        result(Info,H,T,Result).
  return_after_event(Info,[H|T],Ans):-
        return_after_event(Info,T,Ans).
  ```

Lists Processing *(continue)*

- Example
  - ```
    everything_after_a([Head|Tail],Result):-
          Head = a,
          Result = Tail.
    ```
  - ```
    everything_after_a([Head|Tail],Ans):-
          everything_after_a(Tail,Ans).
    ```
- Return a Result – Having processed all elements
  - ```
    process_all(Info,[],[]).
    ```
  - ```
    process_all(Info,[H1|T1],[H2|T2]):-
          process_one(Info,H1,H2),
          process_all(Info,T1,T2).
    ```

Lists Processing *(continue)*

- Example
  - `triple([],[]).`
  - ```
    triple([H1|T1],[H2|T2]):-
        H2 is 3*H1,
        triple(T1,T2).
    ```
    ```

Control and Negation

- Some useful predicate for control
  - `true/0`
    - `father(jim,fred).`
    - `father(jim,fred) :- true.`
  - `fail/0`
    - `lives_forever(X) :- fail.`
  - `repeat/0`
    - `test :- repeat, write(hello), fail.`

พอ fail จะย้อนไปทำ block ก่อนหน้า

redo

forever loop

fail

Control and Negation *(continue)*

- `call/1`
  - `call(write(hello)).`

\+ คือ ¬

- Negation
  - `\+/1`
- Equivalent to ¬

close for assumption ไม่ได้ระบุว่ามี ผช. 2 คนบนโลก

  - `man(jim).`
  - `man(fred).`
  - `woman(X) :- \+ man(X).`

Some General Program Schemata

- Generate – Test
  - ```
    generate_and_test(Info,X):-
        . . .
        generate(Info,X),
        test(Info,X),
        . . .
    ```
  - Finite generator
  - Infinite generator
    ```

Generate – Test

```
integer_with_two_digit_square(X) :- int(X),
     test square(X).
test square(X) :- Y is X*X, Y >= 10, Y < 100.
```

- Finite generator

```
int(1).  int(2).  int(3).  int(4).  int(5).
```

- Infinite Generator

```
int(1).
int(N) :- int(N1), N is N1+1.
```

Test – Process

- ```
  test_process(Info,X,Y):-
          test(Info,X),
          process(Info,X,Y).
  ```
- Example
  - `parity(X,Y) :- odd(X), Y=odd.`
  - `parity(X,Y) :- \+ (odd(X)),Y=even.`
  ```

Failure-Driven Loop

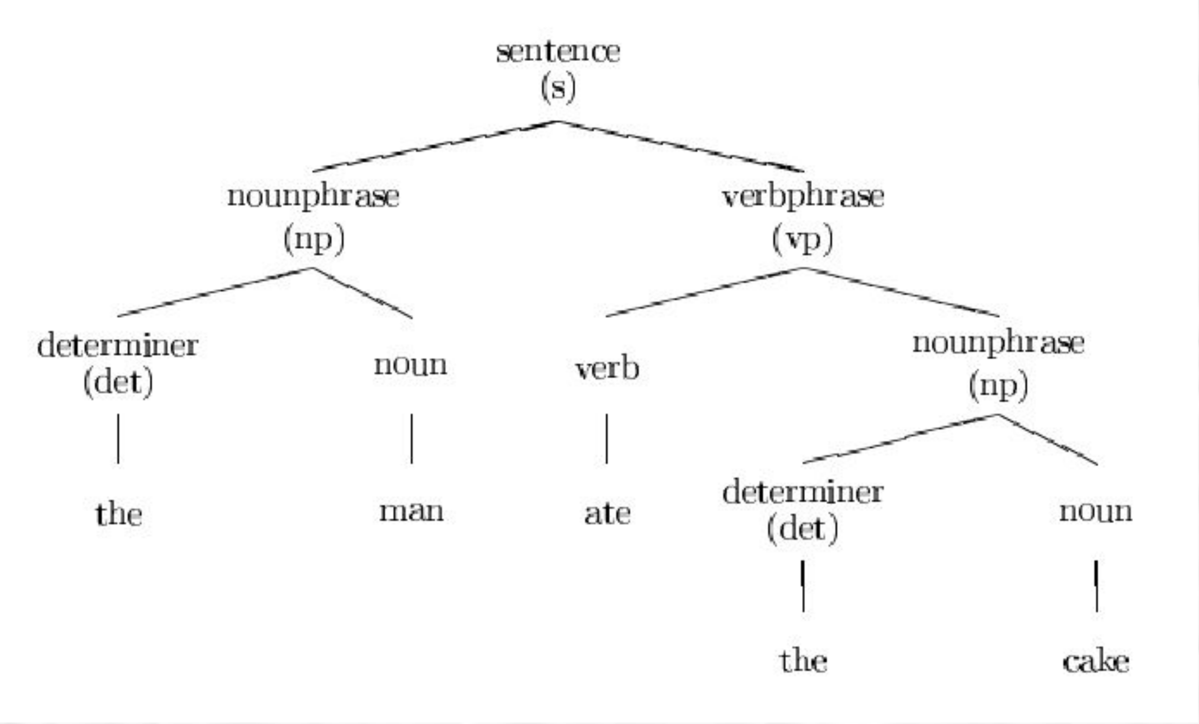
- ```
  failure_driven_loop(Info):-
        generate(Info,Term),
        fail.
  ```
- `failure_driven_loop(Info).`
- An example
  - `int(1). int(2). int(3). int(4). int(5).`
  - `print_int :- int(X),write(X),nl,fail.`
  - `print_int.`
  ```

Parsing in Prolog

- Simple English Syntax
  - Unit: sentence Constructed from noun_phrase followed by a verb_phrase
  - Unit: noun phrase Constructed from: proper noun or determiner followed by a noun
  - Unit: verb phrase Constructed from: verb or verb followed by noun_phrase
  - Unit: determiner Examples: a, the
  - Unit: noun Examples: man, cake
  - Unit verb: Examples: ate

Simple English Syntax

- The man ate the cake.



First Attempt at Parsing

- `sentence(S):-append(NP,VP,S),`
  `noun_phrase(NP),verb_phrase(VP).`
- `noun_phrase(NP):-append(Det,Noun,NP),`
  `determiner(Det),noun(Noun).`
- `verb_phrase(VP):-append(Verb,NP,VP),`
  `verb(Verb),noun_phrase(NP).`
- `determiner([a]). determiner([the]).`
- `noun([man]). noun([cake]).`
- `verb([ate]).`

A Second Approach

```
• sentence(S,S0):-noun_phrase(S,S1),
      verb_phrase(S1,S0).
• noun_phrase(NP,NP0):-
      determiner(NP,NP1),noun(NP1,NP0).
• verb_phrase(VP,VP0):-verb(VP,VP1),
      noun_phrase(VP1,VP0).
• determiner([a|Rest],Rest).
• determiner([the|R],R).
• noun([man|R],R). noun([cake|R],R).
• verb([ate|R],R).
```

Prolog Grammar Rules

- `sentence --> noun_phrase, verb_phrase.`
- `noun_phrase --> determiner, noun.`
- `verb_phrase --> verb, noun_phrase.`
- `determiner --> [a].`
- `determiner --> [the].`
- `noun --> [man].`
- `noun --> [cake].`
- `verb --> [ate].`
- `noun_phrase --> determiner, adjectives, noun.`
- `adjectives --> adjective.`
- `adjectives --> adjective, adjectives.`
- `adjective --> [young].`

Prolog Grammar Rules

- Prolog enables you to write in:
  - `sentence --> noun phrase, verb phrase.`
- and Prolog turns this into:
  - ```
    sentence(S,S0):-noun_phrase(S,S1),
        verb_phrase(S1,S0).
    ```
- And
  - `adjective --> [young].`
- into
  - `adjective(A,A0):-'C'(A,young,A0).`
  - `'C'([H|T],H,T).`
    ```

Prolog Grammar Rules

- How to Extract a Parse Tree
  - `sentence([[np,NP],[vp,VP]]) --> noun_phrase(NP), verb_phrase(VP).`
  - `noun_phrase([[det,Det],[noun,Noun]]) --> determiner(Det), noun(Noun).`
  - `determiner(the) --> [the].`
- So what structure is returned from solving the goal:
  - `sentence(Structure,[the,man,ate,a,cake],[])`
- The result is:
  - `[[np,[[det,the],[noun,man]]],[vp,[…`

Special Control Predicate

- Cut (!/0)
  - On backtracking, all attempts to redo a subgoal to the left of the cut results in the subgoal immediately failing.
  - On backtracking, into the predicate once the call had exited: if one of the clauses defining the predicate had previously contained a cut that had been executed then no other clauses for that predicate may be used to resatisfy the goal being redone.

Special Control Predicate *(continue)*

- Commit
  - `skin(X,Y) :- thai(X),!,Y=yellow.`

    `skin(X,Y) :- asia(X),!,Y=yellow.`

จะไม่สน thai(x)

Special Control Predicate *(continue)*

- Make determination
  - ```
    sum(1,1).
    sum(N,Ans):- NewN is N-1,
                 sum(NewN,Ans1),
                 Ans is Ans1+N.
    ```
  - ```
    sum(1,1):-!.
    sum(N,Ans):- NewN is N-1,
                 sum(NewN,Ans1),
                 Ans is Ans1+N.
    ```

Special Control Predicate *(continue)*

- Fail Goal Now
  - `\+ (Goal):- call(Goal),!,fail.`
    `\+ (Goal).`
  - `woman(X):- man(X),! ,fail.`
    `woman(X).`

Changing the Program

- `assert(C)`: Assert clause C
- `asserta(C)`: Assert C as first clause
- `assertz(C)`: Assert C as last clause
- `retract(C)`: Erase the first clause of form C
- `abolish(Name,Arity)`: Abolish the procedure named F with arity N

Changing the Program *(continue)*

- `assert(man(bill)).`
- `asserta(man(bob)).`
- `assertz(man(jim)).`
- `retract(man(bob)).`
- `abolish(man,1).`

Input/Output

- `read/1`
- ```
  test:-read(X),double(X,Y),
          write(Y),nl.
  ```
- ```
  ?- test.
  |: 2.
  4
  yes
  ```

Input/Output and File

- `see/1:` Take input from the named file
- `seen/0:` Close the current input stream and take input from user
- `seeing/1:` Returns name of current input stream

Input/Output and File *(continue)*

- `test:-read(X),\+(X = -1), double(X,Y),write(Y),nl,test.`
- `go:-see(in),test,seen.`

Input/Output and File *(continue)*

- `tell/1:` Send output to the named file
- `told/0:` Close the current output stream and send output to user
- `telling/1:` Returns name of current output stream

Input/Output and File *(continue)*

- `test:-read(X),\+(X = -1),`
  `double(X,Y),write(Y),nl,test.`
- `go:-tell(out),see(in),test,seen,`
  `told.`

Prolog Examples

- `move(state(middle,onbox,middle,hasnot),grasp, state(middle,onbox,middle,has)).`
- `move(state(P,onfloor,P,H),climb,state(P,onbox, P,H)).`
- `move(state(P1,onfloor,P1,H),push(P1,P2), state(P2,onfloor,P2,H)).`
- `move(state(P1,onfloor,B,H),walk(P1,P2), state(P2,onfloor,B,H)).`

Prolog Examples *(continue)*

```
• depth_first(State,State,[]).
• depth_first(StartState,GoalState,Ans) :-
     findsuccessor(StartState,Successor,Operator),
     depth_first(Successor,GoalState,Ans1),
     Ans=[Operator|Ans1].

• findsuccessor(OldState,NewState,Operator) :-
     move(OldState,Operator,NewState).
```