Плэнерская база данных

База данных Плэнера не имеет никакого отношения к базам данных как объекту традиционного программного обеспечения (реляционным или сетевым, СУБД). Она похожа на т.н. *доску объявлений.*

Плэнерская база данных – это область памяти системы программирования на Плэнере, в которой хранятся *утверждения*. Утверждение представляет собой произвольный L-список, семантику которого (как и семантику всего наполнения базы данных) определяет пользователь. Рекомендуется использовать базу данных для моделирования динамически меняющейся проблемной среды: текущий набор утверждений отражает текущее состояние проблемной среды, запись или вычеркивание утверждений соответствует происходящим в среде изменениям. Динамика базы данных отражает динамику поиска /планирования решения задач.

Основные операции над базой данных: запись утверждения, вычеркивание утверждение (по образцу), поиск утверждения (по образцу).

Запись

[**assert** *asrt with? rec? else?*] – функция assert служит для записи явно заданного (в виде L-списка) утверждения asrt в базу данных. Факультативный параметр with задает список свойств записываемого утверждения (он имеет такую же структуру, что и список свойств идентификатора, но связан с утверждением в целом). Факультативный параметр rec указывает, какие теоремы следует вызвать при записи данного утверждения (некоторые теоремы могут вызываться в этот момент автоматически). Факультативный параметр else содержит рекомендации по поводу того, что следует делать, если записать указанное утверждение в базу данных не удалось (например, потому, что такое утверждение в базе данных уже хранится).

Пример: пусть переменная X имеет значение green
[assert (box A) (with col .X) (else)] - параметр rec в этом примере опущен
При выполнении этого обращения к функции assert происходит следующее:

1) в базу данных записывается утверждение (box A) - имеется ящик A,
2) с ним связывается список свойств (col green) - его цвет – зеленый,
3) если утверждение записать не удалось, то (в соответствии с конкретным видом последнего параметра – else – вырабатывается неуспех.

Поиск

[**search** *pat test?*] – основная функция для поиска в базе данных утверждений по образцу pat. Параметр test (факультативный) позволяет задать требования, предъявляемые к списку свойств утверждения.

Функция search ставит развилку; ищет утверждение, соответствующее образцу; проверяет его список свойств (если он не удовлетворяет требованиям параметра test, ищется другое утверждение соответствующее образцу). Если подходящее (соответствует образцу, удовлетворяет требованиям test, параметр test не задан) утверждение найдено, оно является результатом обращения к search; развилка не уничтожается, следовательно, если в динамике до этого обращения к search «доберется» неуспех, начнется поиск новых утверждений, соответствующих pat. Если найти утверждение, соответствующее pat и удовлетворяющее test, не удалось, развилка отменяется и вырабатывается неуспех.

Пример: пусть после выполнения обращения к assert из предыдущего примера мы задаем:
[search (box [ ]) (test col [non red])]
Результат поиска – утверждение (box A).

Поиск утверждений можно вести поэтапно – находить утверждения, «частично соответствующие» образцу (функция candidates, обращение к которой имеет вид [**candidates** *pat type?*]), а затем более детально анализировать полученный «список кандидатов». Можно искать только одно подходящее утверждение (функция **search1**).

Вычеркивание

[**erase** *pat test?*] – функция для удаления из базы данных утверждений, соответствующих образцу pat и удовлетворяющих требованиям параметра test (если он задан).

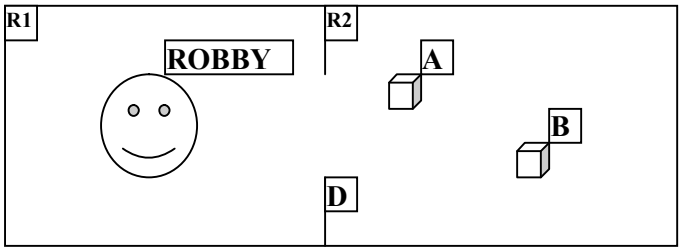
Пример: стереть утверждение (box A) со списком свойств (col green) можно так:
[erase (box [ ]) (test col [non red])]

Плэнерская база данных, отображающая состояние проблемной среды:

(room R1)
(room R2)
(conn R1 R2 D)

(door D)
(box A) – (col green)
(box B) – (col red)

(at ROBBY R1)
(at A R2)
(at B R2)



В этой ситуации можно выполнять операции поиска:
[search (at ROBBY R1)] → T
[search (at ROBBY R2)] → ( )
[search (at ROBBY *X)] → T, X:= R1
[find all (Y) .Y [search (room *Y)]] → (R1 R2)

Режим возвратов

В язык Плэнер встроен т.н. *режим возвратов*, который упрощает реализацию различных поисковых алгоритмов, использующих перебор вариантов. Суть этого режима в следующем. В любом месте программы может быть установлена т.н. *развилка*, от которой возможно несколько вариантов продолжения работы программы. Выбирается один из вариантов, и программа продолжает свою работу. Если затем окажется, что этот вариант неуспешен, вырабатывается т.н. *неуспех*, по которому программа автоматически «откатывается» к последней (по времени) развилке. При этом отменяются все изменения (в значениях переменных и т.п.), произведенные на неуспешном пути, и в этой развилке выбирается следующий вариант, после чего программа снова «идет вперед». Если в развилке уже не оказалось нерассмотренных альтернатив, то неуспех возвращает программу к предыдущей развилке.

В каких местах программы ставить развилки и с какими альтернативами, считать ли выбранный путь вычисления неуспешным и когда вырабатывать неуспех – за все это отвечает автор программы. Встроенный же режим возвратов обеспечивает запоминание мест развилок и то, какие альтернативы в них еще не рассматривались, обеспечивает возврат программы по неуспеху к последней развилке и отмену ранее произведенных изменений в значениях переменных.

Ниже перечислены некоторые из встроенных функций языка Плэнер, позволяющих реализовать режим возвратов.

[**among** *e*]

Значением аргумента должен быть список. Если этот список пуст, функция вырабатывает неуспех, который автоматически возвращает программу к предыдущей ее развилке. Иначе функция запоминает развилку, альтернативами которой является то, что функция в качестве своего значения может выдать любой элемент из этого списка. Вначале функция выдает как свое значение первый элемент списка, завершает на этом работу, и программа продолжает свои вычисления. Но если позже в программе возникнет неуспех, который вернет ее к данной развилке, то функция возобновит свою работу и теперь как свое значение выдаст второй элемент списка, после чего программа снова «идет вперед». И так далее, пока в списке остаются нерассмотренные элементы. После выдачи в качестве своего значения последнего элемента списка, функция уничтожает свою развилку и потому последующий неуспех уже не будет здесь остановлен.

[**alt** e_1 e_2 … e_n]

Функция ставит развилку, *i*-я альтернатива которой – вычисление формы e_i. Если вычислить значение удалось, функция заканчивает работу со значением E_i. Перед вычислением e_n развилка уничтожается.

[**fail**]

Эта функция вырабатывает неуспех, по которому программа автоматически возвращается к последней (по времени) развилке. (Если развилок нет, то вычисление всего выражения самого верхнего уровня программы считается окончившимся неуспешно.)

[**pset** *v e*]

Это аналог функции set, т.е. переменной, имя которой является значением аргумента *v*, присваивается новое значение – значение аргумента *e*. Однако, если присваивание, осуществленное функцией set, отменяется при неуспехе, то действие функции pset при неуспехе не будет отменено. Функция pset (и ей подобные) применяется, когда надо сохранить информацию, полученную на неуспешном пути вычисления программы, для последующих путей.

Для управления режимом возвратов помимо использования процедур, результаты которых не отменяются при неуспехе, в Плэнере есть и другие средства:
- уничтожение развилок и/или обратных операторов,
- использование *именованных* развилок.

Пример плэнер-программы, решающей переборную задачу на основе бэктрекинга:

Пусть задан список L положительных целых чисел. Нужно подобрать набор чисел из L (они могут повторяться), сумма которых равна заданному числу N.

```
[define sum (lambda (L N) [prog ( K (M ( )) (S 0))
      A     [set K [among .L]] [set M ( .K !.M)] [set s [+ .K .S]]
            [cond ([eq .S .N] .M]
                  ([lt .S .N] [go A])
                  (T [fail])] ])]
```

Трассировка выполнения программы при L = (6 3 2 1) и N = 5:
Вход: K – без значения, M = ( ), S = 0

[among (6 3 2 1)] → 6		[among (6 3 2 1)] → 3
[set K 6]	обр.оператор [unassign K]	[set K 3]
[set M (6)]	обр.оператор [set M ( )]	[set M (3)]
[set S 6]	обр.оператор [set S 0]	[set S 3]
S > 5 → неуспех		S < 5 → переход по метке A и новый вызов among:

[among (6 3 2 1)] → 6	[among (6 3 2 1)] → 3	[among (6 3 2 1)] → 2
неуспех	неуспех	S = 5, выход со значением M.

Отметим, что развилки в первом и втором обращениях к функции among остаются. Если в описанное обращение к функции sum откуда-то извне «придет» неуспех, вычисление может возобновиться (при этом будут выбираться не исследованные ранее альтернативы). Например:

```
[prog (X) [set X [sum (6 3 2 1) 5]] [cond ([neq [length .X 3]] [fail])] .X] → (1 1 3).
```

Напечатать (поочередно) все решения рассматриваемой задачи можно с помощью такой конструкции:

```
[prog ( ) [alt ( ) [return T]]
          [print [sum (6 3 2 1) 5]] [fail]]
```

А собрать все решения (в списке Y) и затем выдать этот результат на печать можно так:

```
[prog (X (Y ( )) ) [alt ( ) [return .Y]]
                   [set X [sum (6 3 2 1) 5]]
                   [pset Y (!.Y .X)] [fail]]
```

Теоремы

Теорема Плэнера – процедура, *вызываемая по образцу*.
В языке существуют три типа теорем:

- «целевые» (типа conseq),
- «при записи» (типа antec),
- «при вычеркивании» (типа erasing).

Целевые теоремы используются при планировании решения задач.
Если плэнерская база данных может рассматриваться как модель проблемной ситуации, то набор целевых теорем – как набор средств решения соответствующей задачи. Отобранный и упорядоченный набор теорем может трактоваться как план решения: отдельная теорема из этого набора описывает некоторое элементарное действие (перемещение робота из одной точки в другую, применение некоторой формулы интегрирования и т.п.). Примечательно, что мы можем не знать имена теорем, перебираемых в ходе планирования решения задачи и/или попавших в окончательный вариант плана решения. Автоматически выбираются такие теоремы, которые приводят к достижению цели, описываемой в образце этой теоремы. Вызов теорем происходит в сочетании с режимом возвратов; распространение неуспеха может влиять на процесс планирования решения.
Все теоремы – любого из трех указанных типов – пользователь должен определять сам (встроенных теорем в языке нет).

Пример определения целевой теоремы:

```
[define TRAN-R (conseq (x y)
            (at R *y)
      [search1 (AT R *x)]
      [erase   (AT R .x)]
      [assert  (AT R .y)] )]
```

Эта теорема (с именем TRAN-R) описывает перемещение робота (R) из точки x в точку y. Теорема может быть вызвана по образцу – (at R *y) – в ситуации, когда ставится цель «робот R должен попасть в некоторую точку проблемной среды», скажем в точку G (такая целевая ситуация описывается выражением (at R G), которое соответствует образцу теоремы). Добиться этого можно, применив данную теорему (выполнив соответствующее ей действие в предметном мире) или, возможно, какие-то другие теоремы из числа описанных в программе.

Тело этой теоремы предписывает:

- найти точку, в которой находится R,
- вычеркнуть из базы данных утверждение о том, что R находится в этой точке,
- записать в базу данных утверждение о новом местонахождении R.

Вызов целевых теорем осуществляется с помощью функции achive (или goal):

[**achive** *pat rec?*] - pat – образец, rec? – факультативная рекомендация;

[**goal** *pat test? rec?*] - pat – образец, test? – факультативный набор требований к списку свойств утверждения, rec? – факультативная рекомендация;

Функция goal перед тем, как начать вызов теорем, проверяет, не представлена ли целевая ситуация в базе данных в виде утверждения (это означает, что цель на самом деле достигнута, никакого вызова теорем, никакого планирования решения не нужно).

Факультативный параметр rec (рекомендации) позволяет влиять на процесс перебора теорем, отдавать приоритет некоторым теоремам, учитывать их «стоимость» и т.п. Более того, можно "редактировать" (динамически менять) рекомендации с учетом попыток вызова других теорем.

Примеры рекомендаций:

(use T1 [ ]) - вызвать теорему с именем T1, а если вызов неуспешен, вызывать все остальные,

(try [NOT T3]) - вызывать все теоремы кроме T3,

(use1) - если найдена одна успешная теорема, отменить все развилки.

Пример определения теоремы типа «при вычеркивании»:

```
[define КУБИК_ОСВ (erasing (X)
            (=ЗАНЯТ= *X)
      [assert (=СВОБОДЕН= .X)])]
```

Если из базы данных вычеркивается утверждение о том, что "поверхность некоторого кубика X занята" (если так, то на него нельзя поставить другой кубик), то автоматически будет вызвана и выполнена эта теорема. Она запишет в базу данных утверждение: "поверхность этого кубика X освободилась" (теперь на него можно поставить другой кубик).

Вызов теорем типа «при вычеркивании» осуществляется с помощью функции change – либо явно: [**change** *pat rec?*], либо неявно (из функции erase).

Пример определения теоремы типа «при записи»:

```
[define ПРИШЕЛ (antec (X Y Z)
            (*X =ПРИШЕЛ= *Y)
      [search1 (.X =НАХОДИТСЯ_В= *Z)]
      [erase   (.X =НАХОДИТСЯ_В= .Z)]
      [assert  (.X =НАХОДИТСЯ_В= .Y)])]
```

Если в базу данных записывается утверждение о том, что "X пришел в Y (из Z)", то автоматически будет вызвана и выполнена эта теорема. Она найдет в базе данных утверждение о прежнем местонахождении X, вычеркнет это утверждение и запишет, что "X находится вY".

Вызов теорем типа «при записи» осуществляется с помощью функции draw – опять же либо явно: [**draw** *pat rec?*], либо неявно (из функции assert).