Язык ПЛЭНЕР

Плэнер – один из потомков языка Лисп, созданный специально для реализации систем ИИ. Основные объекты языка (как и в Лиспе) – списки и атомы. Плэнер – очень громоздкий и сложный (для реализации и изучения) язык не нашедший, к сожалению, более или менее широкого практического применения.

Вспомним «Особенности задач ИИ (с точки зрения программирования)»:

1. сложные и динамически меняющиеся структуры данных;
2. большие по объему хранилища данных (базы знаний) и средства эффективной работы с ними;
3. символьные (в основном) данные;
4. модели, отражающие состояние проблемной среды;
5. переборные алгоритмы;
6. алгоритмы поиска по образцу;
7. гибкие структуры управления.

Основные составляющие языка Плэнер:

- средства для работы со списками (*лисповская часть*, используется для работы со списками)
- плэнерская база данных (используется для моделирования ситуаций проблемной среды)
- встроенный режим возвратов (реализует один из основных методов перебора – бэктрекинг)
- аппарат работы с образцами (используется для анализа структуры списков)
- аппарат теорем (используется при планировании решения)

позволяют достаточно эффективно (и быстро) реализовать соответствующие возможности.

Три типа процедур языка Плэнер:

- функции;
- сопоставители (для работы с образцами);
- теоремы (процедуры, вызываемые по образцу).

Примеры конструкций и процедур Плэнера:

Обращение к функции:

[ELEM -2 (A B C D)] → C

Выдать второй (2), считая с конца списка (знак "минус" при 2), элемент списка (A B C D).

Сопоставление образца с выражением:

[IS (*X ПРИШЕЛ !*Y) (САША ПРИШЕЛ ИЗ МАГАЗИНА)] → T

Сопоставление успешно, то есть образец (*X ПРИШЕЛ !*Y) соответствует выражению (САША ПРИШЕЛ ИЗ МАГАЗИНА). Побочный эффект: переменная X получает значение САША, а переменная Y получает значение (ИЗ МАГАЗИНА)

Утверждение плэнерской базы данных:

(BOX A (3 7)) - ящик с названием A находится на плоском квадратном поле с пронумерованными клетками в третьей строке (по горизонтали) и в седьмом столбце (по вертикали).

Запись утверждения в плэнерскую базу данных:

[ASSERT (BOX A) (WITH COL RED)] – в базу данных записывается утверждение (BOX A), то есть "A является ящиком"; записывается также, что "цвет" (COL) этого ящика "красный" (RED).

Поиск утверждения в плэнерской базе данных:

[SEARCH (BOX [ ]) (TEST COL [NON GREEN])] – в базе данных ищется утверждение о некотором ящике, цвет которого не (NON) зеленый (GREEN); возможный ответ (результат поиска) – (BOX A).

Определение плэнерской теоремы:

```
[DEFINE  ОСВОБОДИЛИ (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).

Перейдем к более детальному рассмотрению основных составляющих Плэнера.

Основные объекты языка, средства для работы со списками

Выражения:

- атомарные:
  - обращения к переменным
  - атомы: идентификаторы, числа, строки
- списковые:
  - L-списки (списки в круглых скобках: **(** и **)**)
  - P-списки (списки в квадратных скобках: **[** и **]**)
  - S-списки (списки в «уголках»: **<** и **>**)

Ограничители: <пробел>, **(** , **)** , **[** , **]** , **<** , **>**; цифры; буквы; специальные литеры **+** , **-** , **.** , ***** , **:** , **!** .

Префиксы: **.** , ***** , **:** - простое обращение к переменным; **!.** , **!*** , **!:** - сегментное обращение к переменным.

Простые формы:

- атомы (значением атома является сам этот атом),
- обращения к переменным с префиксом **.** ,
- обращения к константам с префиксом **:** ,
- L-списки (значением L-списка является список из значений его элементов),
- P-списки (обращения к процедурам).

Сегментные формы:

- обращения к переменным с префиксом **!.** ,
- обращения к константам с префиксом **!:** ,
- S-списки (сегментные обращения к процедурам).

В языке Плэнер, как и в Лиспе, программа и обрабатываемые ею данные строятся из *выражений*, к которым относятся: *атомы*, *обращения к переменным*, состоящие из префикса и имени переменной (например, .X) и *списки* всех трех видов. Некоторые выражения (*формы*) можно вычислять, получая в результате значения (ими являются выражения). Программа на Плэнере представляет собой последовательность форм, ее выполнение заключается в вычислении этих форм.

В языке имеется много встроенных (стандартных) функций.

Определение новых функций

Для определения новой функции следует обратиться к встроенной функции define:

[**define** *f dexp*]

Вычисление функции define в качестве побочного эффекта приводит к появлению в программе новой функции с именем *f* и т.н. определяющим выражением *dexp*, которое должно иметь вид

(**lambda** (v_1 v_2 ... v_n) *e*) (n≥0)

где v_i - формальные параметры новой функции, а *e* - форма, зависящая от v_i.

При последующем обращении к этой новой функции

[*f* a_1 a_2 ... a_n]

сначала вычисляются аргументы (фактические параметры) a_i, затем вводятся локальные переменные v_i, которым присваиваются значения соответствующих аргументов a_i, и далее вычисляется форма *e* при этих значениях переменных v_i, после чего эти переменные уничтожаются. Значение данной формы становится значением функции *f* при аргументах a_i.

Операции над списками

[**elem** *n l*]

Значением аргумента *n* должно быть ненулевое целое число (обозначим его *N*), а значением второго аргумента - список (*L*) с любыми скобками. Значением функции является *N*-й от начала элемент списка *L*, если *N*>0, или |*N*|-й от конца элемент этого списка, если *N*<0. В случае, когда в качестве *n* явно указано число, имя функции в обращении можно опустить; например, [1 .X] - это сокращение от [elem 1 .X], т.е. выделяется первый от начала элемент списка, являющего значением переменной X.

[**rest** *n l*]

Значением аргумента *n* должно быть ненулевое целое число (*N*), а значением второго аргумента - список (*L*) с любыми скобками. Значением функции является результат отбрасывания *N* первых элементов списка *L*, если *N*>0, или |*N*|-й последних его элементов, если *N*<0.

[**head** *n l*]

В такой же ситуации значением функции является список из *N* первых элементов списка *L*, если *N*>0, или |*N*|-й последних его элементов, если *N*<0.

Пример: [rest 2 [head 5 (1 2 3 4 5 6)]] → (3 4 5)

Если требуется вычислить список в круглых скобках

(e_1 e_2 ... e_n) (n≥0)

т.е. если он рассматривается как форма, то все его элементы должны быть формами. Значением такого списка является список (с круглыми скобками) из значений его элементов. Например, если переменная X имеет значение (a b), то значением списка (.X c [-1 .X]) является список ((a b) c b).

В таких списках можно использовать *сегментные формы* (сегментные обращения к переменным и сегментные обращения к процедурам, <elem 5 .X>). Сегментная форма вычисляется как обычная (простая) форма, но затем у ее значения, если это список, отбрасываются внешние скобки, и полученная таким образом последовательность элементов поставляется вместо сегментной формы. Например, если переменная X имеет значение (a (b c)), то:

(5 .X ( )) → (5 (a (b c)) ( )), (5 !.X ( )) → (5 a (b c) ( )), (!.X !.X) → (a (b c) a (b c)).

Если переменная X имеет значение (1 2), а переменная Y – значение (3 4):

(.X !.Y) → ((1 2) 3 4) – на Лиспе эти действия задаются так: (cons X Y)

(!.X !.Y) → (1 2 3 4) – на Лиспе эти действия задаются так: (append X Y)

Арифметические функции

[+ n_1 n_2 ... n_k] (k≥2)

Значениями аргументов должны быть числа. Результат вычисления функции – их сумма.

[**-** n_1 n_2]

Значениями обоих аргументов должны быть числа. Значение функции – их разность.

[**max** n_1 n_2 ... n_k] (k≥2)

Значениями аргументов должны быть числа. Результат вычисления функции – их максимум.

[**min** n_1 n_2 ... n_k] (k≥2)

Значениями аргументов должны быть числа. Результат вычисления функции – их минимум.

Предикаты

Предикатом называется форма, значение которой рассматривается как логическое значение «истина» или «ложь». При этом в Плэнере «ложью» считается пустой список ( ), а «истиной» – любое другое выражение (чаще всего в роли «истины» выступает атом T).

[**num** *e*]

Эта функция-предикат проверяет, является ли числом значение ее аргумента. Если да, то значение функции равно T («истина»), иначе – ( ) («ложь»).

[**empty** *e*]

Функция проверяет, является ли значение ее аргумента пустым списком (с любыми скобками). Если да, то значение функции равно T, иначе – ( ).

[**eq** e_1 e_2]

Функция сравнивает значения своих аргументов. Если они равны, то значение функции – T , иначе – ( ).

[**neq** e_1 e_2]

Аналог, но значения аргументов сравниваются на «не равно».

[**memb** e l]

Значением функции является номер первого вхождения формы E в список L.

[**gt** n_1 n_2], [**ge** n_1 n_2], [**lt** n_1 n_2] – сравнение, соответственно, на «больше», «больше или равно» и «меньше».

Предикаты, проверяющие состояние переменных

В Плэнере переменные (формальные параметры процедур или локальные переменные блоков) могут находиться в одном из трех состояний:

1) переменная не описана (в некоторой функции встретилась нелокальная переменная, которая не описана в динамически объемлющих процедурах/блоках),

2) переменная описана, но не имеет значения,

3) переменная описана и имеет некоторое значение.

Для анализа таких ситуаций используются предикаты: [**bound** *v*] и [**hasval** *v*].

Первый из этих предикатов проверяет, является ли значение его параметра описанной (в динамически объемлющих процедурах/блоках) переменной. Если да, возвращается значение T, иначе – ( ).

Значением аргумента предиката hasval должно быть имя переменной, существующей в данный момент. Функция проверяет, имеет ли сейчас эта переменная какое-либо значение или нет. Если имеет, то функция возвращает результат T, а иначе - результат ( ).

Функции для работы со списками свойств идентификаторов

Идентификаторы (и некоторые другие объекты) Плэнера могут иметь т.н. *списки свойств* – списки пар: *свойство-значение*. Если идентификатор является именем некоторого объекта проблемной среды (например, ящика, который может перемещаться роботом), в списке свойств могут быть указаны цвет этого ящика, его вес, линейные размеры и т.п.

Для задания списка свойств используется функция [**plist** *i pl*]. При выполнении обращения к ней идентификатор I получает список свойств PL = (ind_1 val_1 ... ind_n val_n). Отметим, что отсутствие некоторого свойства эквивалентно тому, что это свойство присутствует в списке со значением ( ).

Функция [**put** *i ind val*] позволяет изменить значение указанного свойства IND идентификатора I, а функция [**get** *i ind*?] – получить весь список свойств I (если параметр ind не задан), или же узнать значение указанного свойства.

Логические функции

Функции «отрицание», «конъюнкция», «дизъюнкция» и «условное выражение» аналогичны соответствующим лисп-функциям:

[**not** e]

[**and** e_1 e_2 ... e_k] (k≥1)

[**or** e_1 e_2 ... e_k] (k≥1)

[**cond** (p_1 $e_{1,1}$ $e_{1,2}$ ... $e_{1,k1}$) … (p_n $e_{n,1}$ $e_{n,2}$ ... $e_{n,kn}$)] (n≥1, k_i≥1)

Блочная и связанные с ней функции

[**prog** (v_1 v_2 ... v_n) e_1 e_2 ... e_k] (n≥0, k≥1)

Эту функцию называют «блочной», поскольку ее вычисление напоминает выполнение блоков в других языках программирования. Вычисление функции начинается с того, что вводятся локальные переменные, перечисленные в ее первом аргументе. Если v_i – идентификатор (имя), вводится локальная переменная с этим именем и без начального значения. Если же v_i – пара *(id val)*, то вводится локальная переменная с именем *id* и начальным значением *val* (выражение *val* при этом не вычисляется). После этого функция последовательно вычисляет остальные свои аргументы – формы e_i, которые принято называть операторами. Вычислив последний из них, функция с его значением заканчивает работу, уничтожив при этом свои локальные переменные.

Вычисленные значения всех операторов (кроме последнего) нигде не запоминаются, поэтому имеет смысл в качестве таких операторов использовать только функции/операторы с побочным эффектом. Некоторые из этих функций перечислены ниже.

[**set** *v e*]

Это аналог оператора присваивания. Сначала вычисляются оба аргумента, причем значением аргумента *v* должно быть имя переменной, существующей в данный момент. Функция присваивает этой переменной новое значение – значение аргумента *e*. Это же значение является значением функции set, но оно, как правило, не используется.

[**return** *e*]

Это оператор выхода из блока. Функция return может использоваться только внутри функции prog, поскольку она завершает вычисление ближайшей объемлющей блочной функции, объявляя ее значением значение своего аргумента *e*.

[**go** *e*]

Это оператор перехода. Отметим, что если в качестве оператора функции prog указан идентификатор, то он трактуется как метка. Значением аргумента функции go должен быть идентификатор – одна из меток ближайшей объемлющей блочной функции. Функция go полностью завершает выполнение того оператора этой блочной функции, в который она входит (на любом уровне), и передает управление на оператор, следующий за этой меткой.

В качестве операторов можно использовать составной оператор: [do e_1 e_2 ... e_k] (k≥1),

а также операторы цикла (**loop**, **while**, **until**, **for**). Мы рассмотрим только первый вид оператора цикла:

[**loop** *x l* e_1 e_2 ... e_k]

При выполнении этого оператора вводится параметр цикла *x*, локализованный внутри loop, затем *x* поочередно получает в качестве значения очередной элемент списка L (слева направо), к которому применяются операторы e_1 e_2 ... e_k.

Пример: [set L (1 2 3))]

[prog ((R ( ))) [loop E .L [set R (.E !.R)] .R]] - описанный блок переворачивает входной список (значение переменной L), помещая в конец списка R (в начальный момент пустого) элементы L:

сначала первый, затем второй, затем третий. Результатом работы prog будет значение переменной R на момент завершения цикла – список (3 2 1).

Константы

Как и в Лиспе, константами в Плэнере называются глобальные перемеенные. Они вводятся так: [**cset** c v] – константе с именем C присваивается V.

Функции ввода-вывода

В Плэнере имеется достаточно богатый набор функций ввода-вывода, включающий: функции для работы с файловой системой; функции ввода и вывода символов, текста, строк; функции установки цвета фона и символов, управления курсором и др.
Ниже приводится фрагмент программы, использующей некоторые средства ввода-вывода:

[open screen get]	файл screen открывается на ввод
[active screen get]	файл screen переводится в состояние "активный файл ввода"
[cset a [read]]	считывается очередное выражение (в данном случае, набираемое на клавиатуре) и его значение присваивается константе a; пусть это выражение – (N O T).
[cset b [rev :a]]	константе b присваивается (T O N).
[open ff put]	файл с именем ff открывается на вывод
[active ff put]	файл ff переводится в состояние "активный файл вывода"
[print :b]	в файл ff "печатается" (записывается) выражение (T O N).

Специальные функции

В Плэнере имеется более десятка т.н. специальных функций, реализующих взаимодействие с операционной средой, интерпретатором (перехват синтаксических ошибок с целью их анализа в программе пользователя), а также преобразование типов. Несколько примеров:

[**catch** e1 e2]

Функция вычисляет E1 и, если вычисление закончилось успешно, с этим значением заканчивает свою работу. Если же в процессе вычисления возникла ошибка, вычисляется значение второго параметра.

[**exit** e fn n?]

Выход из функции FN со значением E, если параметр n задан, то выход из N+1 динамически вложенных обращений к функции FN.
Частный случай – выход на верхний уровень программы: [exit ( ) ( )].

[**time**]

Значение функции – время от начала решения задачи.

[**clock**]

Дата и астрономическое время.

[**atl** a]

Функция преобразует идентификатор A в список составляющих его символов. Например, [atl PLAN] → (P L A N).

Образцы и сопоставители

Для анализа данных в Плэнере наряду с традиционными способами (применение функций для работы со списками, логических функций, предикатов, блоков) можно использовать:
- функцию is, реализующую сопоставление *образца* с выражением,
- *сопоставители* – особые процедуры (к которым можно обращаться только в образцах), проверяющие те или иные свойства выражения и его соответствие образцу в целом или отдельному элементу образца.

Соответствующие возможности Плэнера мы рассмотрим кратко и в основном на примерах.

Образец – выражение, которое используется как шаблон при анализе другого выражения (задает структуру и отдельные компоненты этого выражения).
Если выражение удовлетворяет требованиям шаблона, то говорят, что оно *соответствует* образцу, если нет, то *не соответствует*. В первом случае сопоставление образца с выражением *удачно*, во втором – *неудачно*.

Обращение к функции is выглядит так:

[**is** pat e], где pat – образец, e – выражение (сопоставление производится с E – значением e).

Алгоритм сопоставления в общем случае сложен, он предполагает возвраты. Важно, что удачное сопоставление может сопровождаться побочными эффектами – переменным, входящим в состав образца могут присваиваться в качестве значений соответствующие фрагменты выражения. Это позволяет резко сократить запись сложных действий по анализу данных и выделению важных для работы фрагментов.

Примеры (считаем, что используемые переменные описаны вне обращений к is, например, в каком- то объемлющем блоке):

[is (*X .X) (b b)] → T - сопоставление удачно, X (побочный эффект) получает значение b
отметим, что при сопоставлении первого элемента образца (*X) с первым элементом выражения (b) переменная X получила значение b; второй элемент образца (.X) требует, чтобы рассматривалось текущее значение переменной X (префикс *точка*), т.е. как раз b; поэтому удачным будет и сопоставление второго элемента образца со вторым элементом выражения.

[is (*X .X) (b (b))] → ( ) - сопоставление неудачно;

[is (*X !*Y) (1 2 3)] → T - сопоставление удачно; X:= 1, Y:= (2 3)

[is (*X *Y) (1 2 3)] → ( ) - сопоставление неудачно (образец описывает список длиной два).

Обращение к ***сопоставителю*** внешне выглядит как обращение к функции. Однако: значение не вырабатывается, проверяется соответствие выражения/подвыражения образцу/элементу образца, фактические параметры (аргументы) обычно уточняют вид проверки.

Пример: [**list** n] - сопоставитель, соответствующий списку из N элементов

[is [list 2] (a b)] → T - сопоставление удачно;

[is [list 2] (a b c)] → ( ) - сопоставление неудачно;

[is (a <list 2> a) (a b b a)] → T - сопоставление удачно (сегмент b b соответствует сегментному обращению к сопоставителю list;

[is (a [list 2] a) (a (b b) a)] → T - сопоставление удачно.

В Плэнере есть около 30 встроенных сопоставителей, в том числе: логические (типа and и or), типа cond, типа prog (вводят локальные переменные).

Сопоставитель [ ] соответствует любому выражению, сопоставитель < > - любому сегменту.

Некоторые встроенные сопоставители

Сопоставители (без параметров): **id** (идентификатор), **num** (число), **var.** (обращение к переменной с префиксом "."), **var***, **var:**, **var!.**, **var!***, **var!:**, **atomic** (атомарное выражение) – проверяют тип выражения.

[**aut** pat_1 ... pat_n] – сопоставитель "ИЛИ"; сопоставление с некоторым выражением успешно, если выражение соответствует одному из указанных образцов.

[**et** pat_1 ... pat_n] – сопоставитель "И"; сопоставление с некоторым выражением успешно, если выражение соответствует каждому из указанных образцов.

[**when** pat_1 $pat_{1,1}$ $pat_{1,2}$ ... $pat_{1,k1}$) … (pat_n $pat_{n,1}$ $pat_{n,2}$ ... $pat_{n,kn}$)] – "УСЛОВНЫЙ" сопоставитель; если выражение соответствует образцу pat_i, то применяется сопоставитель [et $pat_{i,1}$ $pat_{i,2}$ ... $pat_{i,ki}$].

[**star** *pat*] – сопоставитель соответствует списку, каждый элемент (верхнего уровня) которого соответствует образцу *pat*.

Определение новых сопоставителей

[**define** <имя> (***kappa*** <список параметров> pat)]

Пример:

```
[define pal (kappa ( )
      [aut ( ) [list 1] [same (X) (*X <pal> .X)]])]
```

- рекурсивный сопоставитель pal соответствует любому списку-палиндрому; в нем использованы: логический сопоставитель aut (аналог or), известный нам сопоставитель list, сопоставитель-блок same (вводящий локальную переменную X). Сопоставитель pal проверяет (последовательно): не является ли выражение пустым списком; списком длиной 1; списком, первый и последний элементы которого совпадают, а «середина» удовлетворяет требованиям pal.