Программное обеспечение работ по ИИ

Экспериментальный и эволюционный характер разработок систем ИИ, требования к программному обеспечению

Система Искусственного интеллекта (**Интеллектуальная Система** = **ИС**) – программно-аппаратный комплекс, способный к решению таких задач, решая которые человек использует свои интеллектуальные способности.
Создание ИС – процесс сложный, он предполагает моделирование интеллектуальной деятельности человека (а мы знаем, что она чрезвычайно сложна и слабо изучена)
→
приходится много экспериментировать, создавать все новые и новые варианты, версии
(иногда очень сильно отличающиеся друг от друга)
→
нужны языки высокого/сверхвысокого уровня для быстрого прототипирования

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.

Языки, реально используемые в работах по ИИ:

лиспоподобные языки высокого уровня	- различные диалекты/версии Лиспа
языки логического программирования	- Пролог
языки объектно-ориентированного программирования	- SMALL-TALK
языки на основе фреймов	- KRL,
http://en.wikipedia.org/wiki/KRL_(programming_language)	
универсальные языки программирования	- C/C++, Java, Паскаль
языки, ориентированные на доступ	- LOOPS
(используют специальные процедуры - ***демоны***,	
которые активируются при получении СООБЩЕНИЙ)	
экспертные системы-оболочки	- EMYCIN

Другие средства программной поддержки

Редакторы баз знаний:
штатные текстовые редакторы
автоматические журналы
средства синтаксического контроля
средства контроля содержания

Средства отладки: трассировка, остановы, автоматизация тестирования

Язык ЛИСП

В языке Лисп все действия описываются в виде *функций*. С точки зрения синтаксиса, обращения к функциям, как и обрабатываемые ими данные, представляют собой так называемые *S-выражения*. В простейшем случае S-выражением является *атом* (идентификатор или число), в более сложном – *список*, который представляет собой заключенную в круглые скобки последовательность элементов списка. Отметим, что обращение к функции всегда является списком. Лисповские списки имеют рекурсивную структуру, поскольку элементом списка может быть произвольное S-выражение – как атом, так и список. Примеры списков:

```
(( )  (a  b  1  (c))  class)
(car (quote (( )  ( ))))
(AT  ROBOT  (a  6))
(1  2  #  4  5  6  7  8  3).
```

Пустой список `()` имеет в Лиспе и другое обозначение – `nil`, причем он одновременно относится и к атомам, и к спискам.

Некоторые S-выражения имеют *значения*, которые можно *вычислять*. Такие выражения называются *формами*. Простейшими формами являются числа, идентификаторы `T` и `nil`, а также переменные с уже определенными значениями; значением такой формы служит соответственно само число, сам идентификатор `T` и `nil` или текущее значение переменной.

Формой является также обращение к функции, т.е. список вида

(f a_1 a_2 . . . a_n) ($n \geq 0$)

где f - имя функции, а a_i - ее аргументы, которые для *обычной функции* должны быть формами. В Лиспе есть и *специальные функции*, у которых может быть произвольное число аргументов и аргументы которых либо не вычисляются, либо вычисляются особым образом.

Программа на Лиспе представляет собой последовательность форм, и ее выполнение заключается в последовательном вычислении этих форм. Как правило, в начале программы определяются новые функции, а затем следуют обращения к ним.

Языка Лисп предоставляет большой набор *встроенных* (стандартных) *функций*, на основе которых составляется пользовательская программа. Ниже кратко описаны некоторые из встроенных функций. Все эти функции входят в наиболее известные версии Лиспа - *Common Lisp* и *MuLisp*.

Отметим, что комментарием в Лиспе считается любой текст между двумя точками с запятой (;)

Определение новых функций

Для этого служит встроенная функция defun, к которой можно обратиться так:

(**defun** f (lambda (v_1 v_2 . . . v_n) e)) ($n \geq 0$).

Сокращенный вариант обращения – (defun f (v_1 v_2 . . . v_n) e).

Значением этой функциональной формы является имя определяемой функции, т.е. f. Побочным же эффектом вычисления этой формы будет определение новой лисповской функции с именем f, аргументами (формальными параметрами) v_i и телом e - формой, зависящей от v_i.

Отметим, что таким образом определяется обычная лисп-функция, т.е. функция с фиксированным количеством аргументов, которые всегда вычисляются при обращении к ней.

При последующем обращении в программе к 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, а далее вычисляется форма-тело e при этих значениях переменных v_i, после чего эти переменные уничтожаются. Вычисленное значение формы становится значением функции f.

Основные операции над списками

(**car** l)

Значением аргумента l должен быть непустой список, тогда значением функции является первый элемент (верхнего уровня) этого списка. Например: (car '(AT ROBOT (a 6))) → AT.

Примечание: символ ' имеет особое значение, он показывает, что следующее за ним S-выражение берется в том виде, в каком оно записано (значение его – как формы – не вычисляется) – см. ниже.

(**cdr** l)

Значением аргумента l должен быть непустой список, и значением функции является «хвост» этого списка, т.е. список, полученный отбрасыванием первого элемента. Например, (cdr '(AT ROBOT (a 6))) → (ROBOT (a 6)).

Кроме этих двух функций, называемых селекторами элементов списка, часто используются функции, эквивалентные определенной их суперпозиции. Имена всех таких функций начинаются на букву c, а заканчиваются на букву r, между ними же может стоять произвольная комбинация из не более чем пяти букв a и d, причем буква a означает наличие в суперпозиции функции car, а буква d – функции cdr . Сама же последовательность букв соответствует порядку следования функций car и cdr в эквивалентной суперпозиции.

Например, (caddr l) ≡ (car (cdr (cdr l))) .

Предполагается, что список-аргумент l всех этих функций, так же как и ниже описываемой функции nth, содержит необходимое число элементов (в противном случае вычисление программы прерывается сообщением об ошибке).

(**nth** n l)

Значением аргумента n должно быть положительное целое число (обозначим его N), а значением аргумента l – список. Значением функции является N-й от начала элемент этого списка. Например,

(nth 2 '(AT ROBOT (a 6))) → ROBOT.

(**last** *l*)

Функция выбирает последний (от начала) элемент списка, являющегося значением ее аргумента. Например, (last '(AT ROBOT (a 6))) → (a 6).

(**cons** *e l*)

В отличие от предыдущих функций эта функция является конструктором, т.е. строит новый список, который и выдает в качестве своего результата. Первым элементом этого списка будет значение аргумента *e*, а хвостом списка – значение аргумента *l*. Например, (cons '(A B C) '(D E F)) → ((A B C) D E F).

(**append** l_1 l_2)

Эта функция осуществляет конкатенацию двух списков, являющихся значением ее аргументов. Например, (append '(A B C) '(D E F)) → (A B C D E F).

(**list** e_1 e_2 . . . e_n) ($n \geq 1$)

Эта функция имеет произвольное количество аргументов, из значений которых она строит список (количество элементов на верхнем уровне результирующего списка равно количеству аргументов).

Например, если переменная X имеет своим значением список (p (q r)), а переменная Y – список (t), то

значение формы (cons X Y) равно ((p (q r)) t),
значение формы (append X Y) равно (p (q r) t),
значение формы (list X Y) равно ((p (q r)) (t)).

Арифметические функции

(**length** *l*)

Значением аргумента *l* должен быть список, функция вычисляет количество элементов (верхнего уровня) этого списка. К примеру, значение (length *X*) равно 2, если переменная X имеет значение (p(q r)).
Значениями аргументов нижеследующих функций должны быть числа, над которыми производятся арифметические операции.

(**add1** *n*)

Функция прибавляет число 1 к числу-аргументу и выдает результат в качестве своего значения.

(**sub1** *n*)

Эта функция вычитает 1 из значения своего аргумента и выдает результат в качестве своего значения.

(+ n_1 n_2)

Значением функции является сумма значений ее аргументов.

(- n_1 n_2)

Значением этой функции является разность значений ее аргументов.

(**mod** n_1 n_2)

Функция выполняет деление нацело значение первого аргумента на значение второго, и результат выдает в качестве своего значения.

(**rem** n_1 n_2)

Результат вычисления этой функции – остаток от деления первого числа на второе.

Предикаты

Предикатом обычно называется форма, значение которой рассматривается как логическое значение «истина» или «ложь». Особенностью языка Лисп является то, что «ложью» считается пустой список, записываемый как `()` или nil, а «истиной» – любое другое выражение (часто атом T).

(**null** *e*)

Эта функция проверяет, является ли значение ее аргумента пустым списком: если да, то значение функции равно T, иначе – `()`.

(**eq** e_1 e_2)

Функция сравнивает значения своих аргументов, которые должны быть атомами-идентификаторами. В случае их совпадения (идентичности) значение функции равно T, иначе – `()`.

(**eql** e_1 e_2)

В отличие от предыдущей функции, данная функция сравнивает значения своих аргументов, которыми могут быть не только атомы-идентификаторы, но и атомы-числа. Если они равны, то значение функции равно `T`, иначе – ( ). Отметим, что во многих версиях Лиспа для сравнения любых атомов используется функция **eq**.

(**neq** e_1 e_2)

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

(**equal** e_1 e_2)

Функция производит сравнение двух произвольных S-выражений – значений своих аргументов. Если они равны (имеют одинаковую структуру и состоят из одинаковых атомов), то значение функции равно T, иначе – ().

(**member** a l)

Значением первого аргумента должен быть атом, а второго – список. Функция производит поиск заданного атома на верхнем его уровне заданного списка. В случае успеха поиска значение функции равно T, иначе – ( ).

(**gt** n_1 n_2) или (> n_1 n_2)

Значениями аргументов этой функции должны быть числа. Если первое из них больше второго, то значение функции равно T, иначе – ().

(**lt** n_1 n_2) или (< n_1 n_2)

Аналог предыдущей функции, но числа сравниваются на «меньше».

Логические функции

Так называются три функции, реализующие основные логические операции.

(**not** e)

Эта функция, реализующая «отрицание», является дубликатом функции null: если значение аргумента равно ( ) («ложь»), то функция выдает результат T («истина»), а при любом другом значении аргумента выдает результат ( ).

(**and** e_1 e_2 ... e_k) (k≥1)

Это «конъюнкция». Функция по очереди вычисляет свои аргументы. Если значение очередного из них равно ( ) («ложь»), то функция, не вычисляя оставшиеся аргументы, заканчивает свою работу со значением ( ), а иначе переходит к вычислению следующего аргумента. Если функция дошла до вычисления последнего аргумента, то с его значением она и заканчивает свою работу.

(**or** e_1 e_2 ... e_k) (k≥1)

Это «дизъюнкция». Функция по очереди вычисляет свои аргументы. Если значение очередного из них не равно ( ) («ложь»), то функция, не вычисляя оставшиеся аргументы, заканчивает свою работу со значением этого аргумента, в противном случае она переходит к вычислению следующего аргумента. Если функция дошла до вычисления последнего аргумента, то с его значением она и заканчивает свою работу.

К числу логических функций можно отнести и лисповское условное выражение:

(**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)

Функция cond последовательно вычисляет первые элементы своих аргументов – обращения к предикатам p_i. Если все они имеют значение ( ) («ложь»), тогда функция заканчивает свою работу со значением ( ). Но если был обнаружен предикат p_i, значение которого отлично от ( ), т.е. он имеет значение «истина», тогда функция cond уже не будет рассматривать остальные предикаты, а последовательно вычислит формы $e_{i,j}$ из этого i-го аргумента и со значением последнего из них закончит свою работу. Заметим, что поскольку значения предыдущих форм из этого аргумента нигде не запоминаются, то в качестве этих форм имеет смысл использовать только такие, которые имеют побочный эффект, например, функции присваивания значений переменным или печати.

Специальные функции

(**quote** e) или 'e

Это функция блокировки вычислений: она выдает в качестве значения свой аргумент, не вычисляя его. Например, значением формы '(car (2)) является само выражение (car (2)).

(**gensym**)

Это функция генерации уникальных атомов (символов): при каждом обращении к ней выдается новый атом-идентификатор. Этот идентификатор получается склеиванием специального префикса и очередного номера (целого числа). Префикс и целое число, от которого начинается нумерация генерируемых атомов, могут быть установлены заранее, как, например, в языке MuLisp:

(setq *gensym-prefix* 'S) (setq *gensym-count* 2)

После этого при последовательных обращениях к функции gensym она будет выдавать атомы S2, S3, S4 и т.д.

Блочная и связанные с ней функции

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

Эту специальную функцию называют «блочной», поскольку ее вычисление напоминает выполнение блоков в других языках программирования. Вычисление функции начинается с того, что вводятся

локальные переменные v_i, перечисленные в ее первом аргументе, и всем им в качестве начального значения присваивается пустой список `nil`. После этого функция последовательно вычисляет остальные свои аргументы – формы e_i. Вычислив последнюю из них, функция prog заканчивает работу со значением этой формы, уничтожив перед этим все свои локальные переменные v_i .

Вычисленные значения всех форм e_i, кроме последней, нигде не запоминаются, поэтому имеет смысл использовать в качестве них только функции с побочным эффектом. Некоторые из этих функций перечислены ниже.

В качестве одной из форм e_i может быть записан атом-идентификатор, в этом случае он не вычисляется, а трактуется как метка, на которую будет производиться переход внутри этого блока (функцией go).

(**return** *e*)

Это функция досрочного выхода из блока. Она может использоваться только внутри блочной функции prog, поскольку завершает вычисление ближайшей объемлющей блочной функции, устанавливая ее значение равным значению аргумента *e*.

(**go** *e*)

Функция реализует переход по метке. Аргумент ее не вычисляется, в качестве ее аргумента должен быть задан идентификатор – одна из меток ближайшей объемлющей блочной функции. Функция go полностью завершает вычисление той формы этой блочной функции, в которую она входит (на любом уровне), и осуществляет переход на вычисление формы, непосредственно следующей за указанной меткой.

(**setq** *v e*)

Это аналог оператора присваивания. В качестве аргумента *v* (он не вычисляется) должно быть задано имя переменной, существующей в данный момент. Функция присваивает этой переменной новое значение – вычисленное значение формы *e*. Это же значение является значением и самой функции setq, однако оно, как правило, не используется.

Следующие две особые функции используются для упрощения записи часто используемых конструкций (setq V (cdr V)) и (setq V (cons (e V)).

(**pop** *v*)

Аргументом этой функции (он не вычисляется) должно быть имя переменной, существующей в данный момент и имеющей своим значением непустой список. Хвост этого непустого списка присваивается в качестве нового значения указанной переменной, а также выступает в качестве значения самой функции pop.

(**push** *e v*)

В качестве второго аргумента этой функции (он не вычисляется) должно быть задано имя переменной, в качестве первого – произвольная форма. Функция вычисляет эту форму и строит новый список, первый элемент которого – вычисленное значение, а хвост – список, являющийся значением переменной *v*. Результирующий список становится новым значением переменной *v* и значением самой функции push.

Например, если переменная X имеет значение (d (e) g), а переменная U – значение (1 2), то значение формы (pop X) равно ((e) g), а значение (push U X) равно ((1 2) d (e) g).

Основным средством реализации циклических программ в Лиспе является рекурсия.
Рассмотрим примеры простейших рекурсивных программ на Лиспе:

(listp *l*) – функция-предикат, проверяющая является ли значение ее аргумента списком (на верхнем уровне). Если да, то значение функции равно T, иначе – `()`.

```
(defun  listp  (lambda (x)
        (cond  ((null  x)  T)
               ((atom  x)  ( ))
               ( T  (listp  (cdr  x))) )))
```

(memb *a l*) - функция ищет атом, являющийся значением первого ее аргумента, в списке (на верхнем его уровне), являющемся значением второго аргумента. В случае успеха поиска значение функции равно T, иначе – `()`.

```
(defun  memb  (lambda  (a  l)
        (cond  ((null  l)  nil)
               ((eq  a (car  l))  T)
               (T  (memb  a  (cdr l)) ) ))
```

(out *a l*) - функция удаляет из списка, являющегося значением ее второго аргумента, все вхождения (на верхнем) атома, являющегося значением первого аргумента.

```
(defun out (lambda (a l)
        (cond ((null l) nil)
              ((eq a (car l)) (out a (cdr l)))
              (T (cons (car l) (out a (cdr l))))) ))
```

(equal e_1 e_2) – функция, сравнивающая два произвольных S-выражения – значения своих аргументов. Если они равны (имеют одинаковую структуру и состоят из одинаковых атомов), то значение функции равно T, иначе – ().

```
(defun equal (lambda (x y)
              (cond ((atom x) (eq x y))
                    ((atom y) ( ))
                    ((equal (car x) (car y)) (equal (cdr x) (cdr y)))
                    ( T ( ) ) )))
```