Семантический анализ

Особенности семантики и семантического анализа

Следующий шаг анализа текста программы – семантический, существенно отличается от двух предыдущих – лексического и синтаксического. И дело не столько в том, что фаза семантического анализа реализуется не формальными, а **содержательными** методами (т.е. на данный момент нет универсальных математических моделей и формальных средств описания «смысла» программы). Лексический и синтаксический анализ имеют дело со **структурными,** т.е. внешними, текстовыми конструкциями языка. Семантика же, ориентированная на содержательную интерпретацию, имеет дело с внутренним представлением «смысла» объектов, описанных в программе. Для любого, имеющего опыт практического программирования, ясно, что формальные конструкции языка дают описание свойств и действий над **внутренними объектами**, с которыми имеет дело программа. Для начала перечислим все, что их касается и лежит на поверхности:

- ·большинство объектов являются **именованными.** Имя объекта позволяет его идентифицировать, существуют различные области действия имен, соглашения об именах, различные умолчания и т.п.. Все это относится к семантике;
- виды, сложность и набор характеристик объектов различаются в разных языках программирования и сильно зависят от области приложения языка (в этом смысле семантика языков программирования более разнообразна, нежели синтаксис и лексика). Например, классический Си, ориентированный на максимальное приближение к архитектуре компьютера, работает с такими объектами, как **типы данных, переменные, функции**. Все они имеют различные свойства и характеристики. Например, переменная характеризуется именем, типом данных, размерностью, областью действия, временем жизни, текущим значением;
- объекты связаны между собой (ссылаются друг на друга). В том же Си переменная ссылается на описание того типа данных, к которому она относится, далее производный тип данных ссылается на базовый и т.п.. Можно сказать, что семантика программы во внутреннем представлении выглядит как система взаимосвязанных объектов;
- внутреннее представление семантики программы не совсем удачно называется **семантическими таблицами.** На самом деле структура данных, соответствующая представлению семантики, может быть любой. Термин **«таблицы»** говорит о том, что имеются множества объектов различных типов, для каждого из которых заведена отдельная таблица, но нельзя забывать, что элементы различных таблиц связаны между собой. Наиболее близкий термин для описания подобной системы – **база данных.**

Семантика программы – внутренняя модель (база данных) множества именованных объектов, с которыми работает программа, с описанием их свойств, характеристик и связей.

Теперь, когда у нас есть представление о синтаксической фазе, можно оценить ее центральную роль в организации процесса трансляции. Только на уровне синтаксиса текст программы представляет собой **единое структурное целое –** любое предложение языка сводится к единственному начальному нетерминалу **Z.** Лексические единицы, как известно, вообще независимы друг от друга и являются терминальными символами синтаксиса. Семантика программы тоже не обладает структурной целостностью и представлена фрагментарно, но при этом связана с синтаксисом следующим образом:

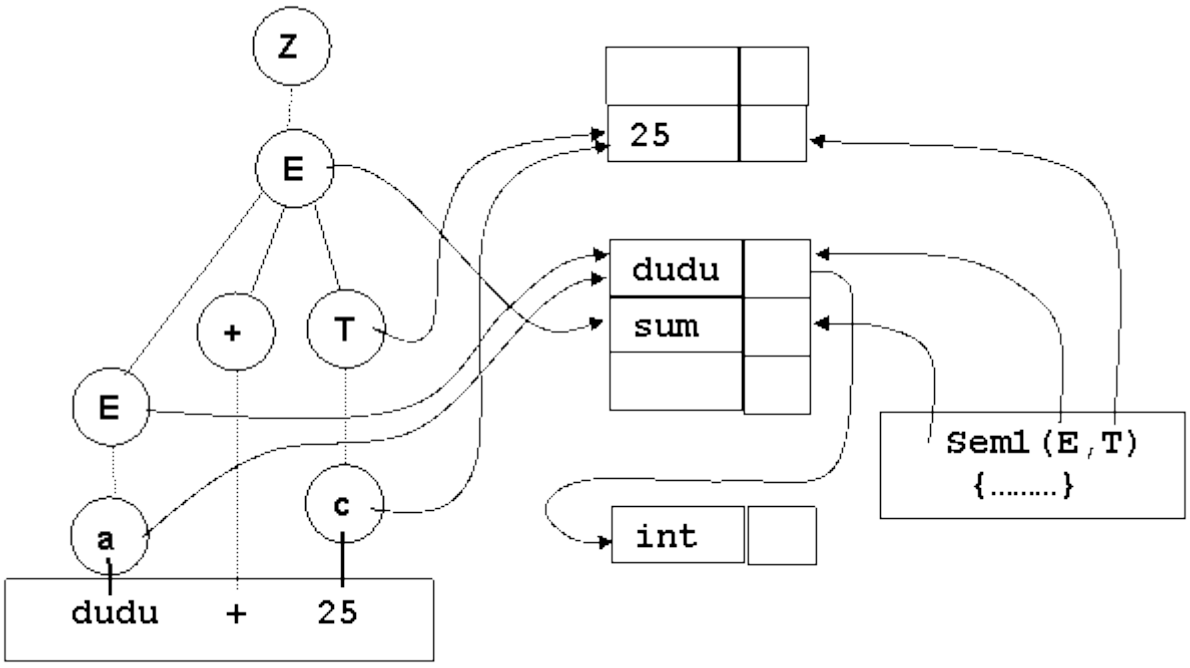
- один и тот же семантический объект (например, переменная) может встречаться в различных, синтаксически несвязанных частях программы;
- синтаксические конструкции *описаний, определений и объявлений* являются **источником семантики** объектов программы, они «заявляют» о существовании объектов и задают их свойства;
- синтаксические конструкции, связанные с *действиями, выполняемыми над объектами*, являются **потребителями семантики,** их интерпретация, корректность, «смысл» зависят от семантических свойств объекта. Забегая вперед,

можно заметить, что заключительная фаза трансляции (генерация кода, интерпретация) может рассматриваться как особые семантические действия, производимые над объектами;

- первичным источником семантики является лексический анализ. Некоторые из лексем (например, идентификаторы и константы) наряду с **классом лексемы** (обозначение выходной единицы лексического анализа), т.е. **символом** (обозначение той же единицы на входе синтаксического анализатора) имеют **значение.** Значением лексемы является сама распознанная цепочка литер, она и представляет **семантическую составляющую лексемы,** которая попадает в семантические таблицы;

- лексемы, или то же самое, что терминальные символы входной строки (в терминах синтаксического анализа), ссылаются в семантические таблицах на свою семантику. В дальнейшем каждый промежуточный нетерминал также ссылается на собственную семантику. При этом любое правило преобразует семантику терминалов и нетерминалов правой части в семантику нетерминала левой части при помощи назначенной правилу **семантической процедуры.** Таким образом**,** формирование семантической составляющей связано с движением **снизу вверх** по синтаксическому дереву, от вершин – потомков к предкам.

- семантическая процедура, получая ссылки на семантику терминальных и нетерминальных символов правой части, формирует семантику результата и размещает ее в семантических таблицах, связывая ее через ссылку с нетерминалом левой части. Таким образом, семантическая составляющая транслятора тоже является фрагментарной (набор семантических процедур, соответствующих правилам грамматики) и объединяется в единое целое только в рамках синтаксического дерева.



Задача семантического анализа, т.е. «описания смысла» фразы относится скорее к области искусственного интеллекта. Ее неформализуемость означает, что она **не имеет формальных средств описания,** например, языков**.** Следовательно, семантическая модель языка разрабатывается в каждом случае уникально, здесь отсутствует общий подход, а имеет место набор частных решений и рекомендаций. Отсюда и уникальность семантики языка.

Замечание. Исторически сложилось, что в описание семантической фазы трансляции часто попадают разделы, связанные со структурами данных и алгоритмами их обработки, например, такие как хеширование (размещение и поиск вычислением адреса), двоичный поиск в таблицах. На самом деле они не имеют никакого отношения к основной идее: множества объектов внутреннего представления программы могут быть реализованы и в виде списков, деревьев и т.п..

О взаимосвязи синтаксиса и семантики

Как известно, один и тот же синтаксис может быть представлен различными грамматиками. Они эквивалентны между собой, поскольку распознают одно и то же множество цепочек (Как известно, проблема эквивалентности формальных грамматик **алгоритмически неразрешима**, т.е. невозможно разработать алгоритм, устанавливающий эквивалентность двух произвольных грамматик). Мы уже видели, что требования к грамматикам восходящего и нисходящего разбора также различны. Различны и технологически подходы к реализации одних и тех же элементов синтаксиса (повторений, необязательных элементов синтаксиса и т.п.). Поэтому получается, что в различных грамматиках при разборе одного и того же предложении будут строиться различные деревья, но результат трансляции (например, последовательность выполняемых операций в выражении) должен быть если не одинаковым, то эквивалентным.

Поэтому задача семантического анализа состоит в том, чтобы извлечь из формально построенного синтаксического дерева содержательную информацию о структуре программы. И делается это всегда уникальным способом, отражающим **смысл** каждого правила формальной грамматики, примененного при построении данного дерева.

Кроме того, существует семантическая связь между элементами синтаксиса, реализуемыми различными правилами. Например, описание типа переменной в Си вида **int a,b=5,c[10]** синтаксически реализуется группой правил, задающих цикл описания отдельных переменных, а общий тип и полученный список определяется отдельным правилом. Но между ними имеется связь, состоящая в том, что семантика типа **int** распространяется на все элементы списка. Отсюда имеем различные варианты реализации семантических процедур:

- если синтаксическая единица реализуется в одном правиле, то ее семантическая обработка., а также генерация кода или интерпретация могут быть выполнены независимой семантической процедурой. Взаимодействие с другими процедурами происходит по указанной выше схеме: процедура использует ссылки на семантику символов правой части (потомков поддерева для этого правила), формирует и возвращает ссылку на семантику левой части (корневой вершины поддерева). То же самое можно сказать и способе передачи сгенерированного кода или результата интерпретации;
- если синтаксическая единица реализуется несколькими правилами, то такая стройная картина нарушается. Например, может потребоваться более сложное взаимодействие семантических процедур (не только вверх-вниз по дереву, но и между смежными вершинами и т.п.). Для передачи результатов между процедурами могут использоваться не только ссылки на записи в семантических таблицах, но и более сложные структуры данных.

В приведенном выше примере правила, соединяющее общий тип описания (**int**) со списком переменных, имеет вид **X::=TL** и получает от первого нетерминала ссылку в семантической таблице типов, а от второго нетерминала множество ссылок (массив, список) на записи в семантической таблице переменных. Семантическая процедура правила должна дополнить все цепочки описаний типа данных в этом множестве записей указанным общим типов (причем по семантике определения типа данных в Си он должен быть дописан **в конец цепочки**).

Семантические таблицы для Си-подобного компилятора

Попробуем в первом приближении представить себе, как может выглядеть внутреннее описание семантики объектов «классического» Си. В нем имеет место одноуровневая система объектов различных видов: **типов данных, переменных, функций.** Каждому такому виду объектов должна соответствовать своя **семантическая таблица (таблица имен).** На самом деле речь идет не об одной, а о нескольких таблицах.

Поскольку имеется одна глобальная область, а у каждой функции – своя локальная область переменных, то транслятор должен иметь множество таблиц переменных – по одной на каждую область действия. Очевидно, каждая функция должна ссылать на свою таблицу переменных. Кроме того, при трансляции любой функции имеют место **контекст локальных** и **глобальных переменных:** анализ имен переменных производится в соответствии с глобальной таблицей имен и таблицей имен текущей транслируемой функции. Очевидно, что семантическая процедура, которая связана с синтаксисом заголовка функции, должна создавать такую таблицу и устанавливать ее в качестве локального контекста трансляции. Кстати, формальные параметры

функции также являются составной частью такого контекста, их семантика также должна быть внесена в эту таблицу.

Для Си++ в условиях объектно-ориентированного представления транслируемой программы возникает еще один вид составных семантических объектов – **классы.** Появляется еще один текущий контекст – **контекст текущего класса.** Каждый класс ссылается на собственную таблицу **данных класса (свойств)** и **функций класса (методов).** Иллюстрацией этого принципа является возможность идентификации объекта программы как по полному, составному имени (например, **A::F** – функция **F** в классе **A**), так и по краткому в соответствующем контексте. Наличие механизма наследования, а также возможность включения описания класса внутри класса еще более усложняют структуру семантических таблиц и их взаимосвязи.

Рассмотрим, как можно представить семантику переменных и типов данных (**ТД**) для «классического» Си. Прежде всего, условимся, что все типы данных, явно или неявно определяемые в программе, будут размещаться в таблице (массиве) **TYPES**. Элементом этого массива является структура, которая содержит описание **ТД** (**d_type**). Компонентами этой структуры являются:

- **name** - имя ТД. Если этот ТД является базовым, то его имя инициализировано в таблице. Если это ТД определяется в описателе typedef, то имеющийся в определении идентификатор становится именем ТД (это и составляет семантику описателя typedef: имя переменной, содержащейся в контекстном определении, следующим за typedef, и является именем вводимого ТД). Кроме того, в контексте часто определяются ТД для переменных, а также абстрактные ТД, которые не имеют имени – для них имя содержит пустую строку;
- **size** - размерность памяти под ТД в байтах. Каждый ТД в Си имеет фиксированную размерность, которая используется для создания переменных такого типа;
- **TYPE** – идентификатор текущего ТД. Если ТД является базовым, то он идентифицируется значением BTD. Если это производный ТД, то он обычно представляет собой цепочку (или дерево) вложенных друг в друга ТД, Текущий ТД может быть указателем (PTR), массивом (ARR), структурой (STRU) или объединением (UNI) (функции здесь не рассматриваются).
- **child** – указатель на описание вложенного (составляющего) ТД. Для всех ТД, кроме структуры и объединения, имеется единственный составляющий ТД, на который ссылается указатель. Для типов struct и union указатель ссылается на массив описателей составляющих ТД;
- **dim** – количество элементов в составляющем ТД или его описании. Если текущий ТД – массив, то это количество его элементов, а child указывает на единственный описатель вложенного ТД. Если это структура или объединение, то dim определяет количество элементов структуры, а child указывает на массив описателей этих элементов.

В следующем примере семантическая сеть для различных ТД задана с помощью инициализации, чтобы по ней можно было показать, каким образом определения различных **ТД** сохраняются в семантических таблицах. Реально же инициализируются только описания базовых **ТД**, остальные строятся динамически в процессе семантического анализа явных и контекстных определений типов.

```
typedef char *PSTR;
struct man {
        char name[20];
        char *addr;
        int hight;
        } A[10];
int B[20],C;
PSTR pp;

#define         BTD     0
#define         PTR     1
#define         STRU    2
#define         UNI     3
#define         ARR     4
struct d_type {
```

```
        char     name[20];        // Имя ТД
        int     size;             // Размерность памяти ТД в байтах
        int     dim;              // Количество элементов вложенного ТД
        int     TYPE;             // Идентификатор типа
        d_type *parent;           // Составляющий ТД (один или несколько)
        };
extern    d_type TYPES[100];

// Определение полей структурированного типа man
d_type XXX[]={
        {“name”,     20,  20,  ARR, &TYPES[4] },          // char name[20];
        {“addr”,      4,   1,  PTR,  &TYPES[3] },          // char *addr;
        {“hight”,     4,   1,  BTD,  &TYPES[1] }};         // int class;

// Основная таблица типов данных
d_type TYPES[100] ={
// Определение БТД
        {“char”,    1,     0,     BTD,    NULL},            //  [0]
        {“int”,     4,     0,     BTD,    NULL},            //  [1]
        {‘long’,    8,     0,     BTD,    NULL},            //  [2]
// Неявное определение ТД или абстрактный ТД char*
        {“”,        4,     1,     PTR,    &TYPES[0]},       //  [3]
// Неявное определение ТД или абстрактный ТД char[]
        {“”,        4,     1,     PTR,    &TYPES[0]},       //  [4]
// Явное определение ТД typedef char *PSTR
        {“PSTR”,   4,     1,     PTR     &TYPES[0]},       //  [5]
// Неявное определение ТД или абстрактный ТД int [20];
        {“”,        80,    20,    ARR,    &TYPES[1]},       //  [6]
// Определение структурированного типа man
        {“man”,     28,    3,     STRU,   &XXX},            //  [7]
// Определение ТД – массив структур man[10]
        {“”,        280,   1,     ARR,    &TYPES[7]}};      //  [8]
```

Структура семантической таблицы для переменных естественным образом вытекает из ее основных свойств в языке и должно содержать:

- имя переменной;
- указатель на описание типа в таблице типов;
- смещение (адрес), который получает эта переменная при трансляции в том сегменте данных, где она размещается компилятором;
- указатель на область памяти, где размещаются ее значение – для интерпретатора.

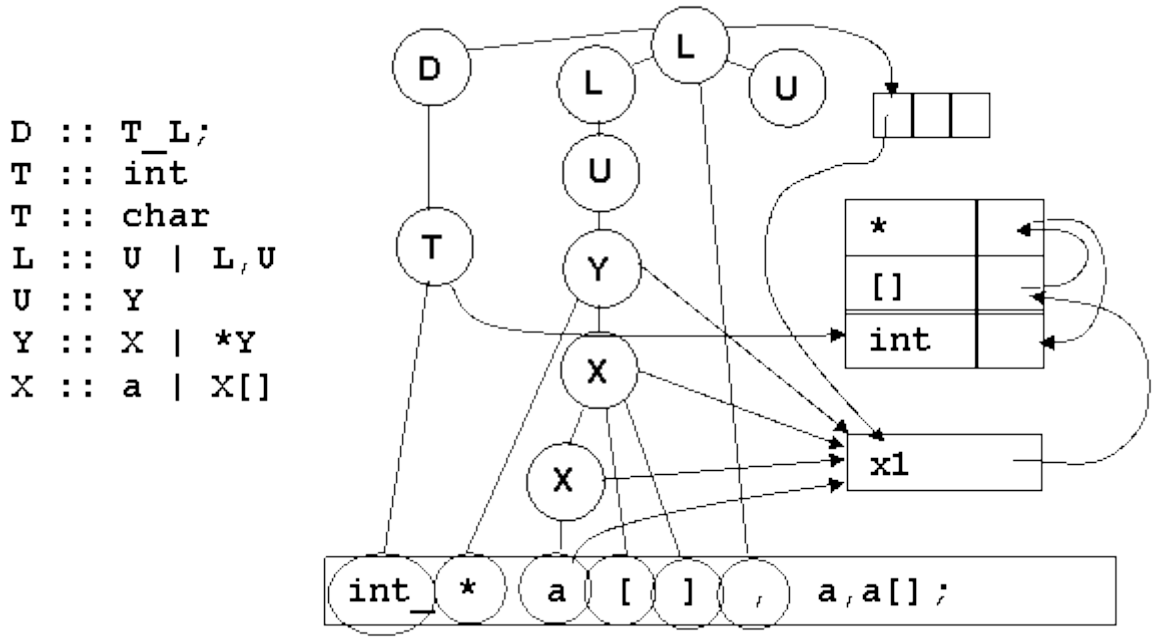
```
struct var {
        char name[20];          // имя переменной
        d_type  *ptype;         // указатель на описатель ТД переменной
        int     offset;         // смещение (адрес) в сегменте данных
        void    *addr;          // адрес переменной в памяти (для интепретатора)
        };
```

Каждая функция в программе должна иметь собственный контекст – семантическую таблицу локальных переменных и формальных параметров. Кроме того, существует семантическая таблица глобальных переменных.

```
var GLOBAL[100]={
        {“A”,     &TYPES[8], 0,     NULL},
        {“B”,     &TYPES[6], 280, NULL},
        {“C”,     &TYPES[1], 360, NULL},
        {“pp”,    &TYPES[5], 364, NULL},
        {NULL}                                  // конец таблицы
        };
```

Понятно, что программы, работающие даже с такой семантической сетью, будут достаточно сложны. В качестве примера приведем программу, которая подсчитывает для произвольного ТД его размерность памяти в байтах с учетом всех вложенных в него ТД. Поскольку структурированный ТД предполагает ветвление семантической сети, то такая программа будет в добавок ко всему и рекурсивной.

```
//---------------------------------------- подсчет размерности производного ТД
int     GetSize(d_type *p){
switch     (p->TYPE)
        {
        // Размерность БТД фиксирована
case BTD:       return p->size;
        // Размерность указателя постоянна
case PTR:       return 4;
        // Размерность массива – произведение числа элементов
        // на размерность вложенного ТД
case ARR:       return p->dim * GetSize(p->child);
// Размерность структуры – сумма размерностей элементов
case STRU:
                int s,i;
                 for (s=0,i=0; i<p->dim; i++)
                        s+=GetSize(&(p->child[i]));
                 return s;
// Размерность объединения – максимальная размерность элемента
case STRU:
                int s,i,k;
                 for (s=0,i=0; i<p->dim; i++)
                        { k=GetSize(&(p->child[i]));  if (k>s) s=k; }
                 return s;
        }}
```



Различные элементы синтаксиса языка программирования по-разному взаимодействуют с семантикой.

При синтаксическом анализе описаний, т.е. при обработке правил, составляющих определение или объявление переменных, типов данных, заголовков функции, семантические процедуры **заполняют** описанные выше таблицы данными о семантике объектов программы, устанавливают взаимные ссылки**.**

Проследим, как можно сформировать требуемые структуру синтаксических таблиц, используя восходящий метод «свертка-перенос». Приведенная грамматика обеспечивает порядок построения синтаксического дерева, соответствующий приоритетам операций ***** и **[]** в описании. Все остальное выполняется семантическими процедурами в такой последовательности:

- с терминалом **a,** соответствующим идентификатору, связана первичная семантика – значение **x1.** Оно становится записью в таблице переменных, первоначально ссылка на описание типа данных отсутствует **(NULL);**
- последовательность сверток передает указатель на запись для **x1** в таблице переменных. Некоторые правила предназначены исключительно для установления приоритетов (например, **Y::X** определяет приоритет **[]** над *****). Они не меняют семантики передаваемого объекта. Другие правила, выполняющие свертку операций ***** и **[],** должны менять тип переменной. Для этого они добавляют новую запись в таблицу типов и помещают указатель нее в конец цепочки типов, начинающейся с записи **x1.** То есть цепочка типов формируется по принципу очереди: **переменная x1 – массив - указатель**;
- когда начинается свертка по правилам, соответствующим списку описателей, для нетерминала **L** заводится массив указателей на записи в таблице переменных, каждый новый нетерминал **U** добавляет к нему еще один указатель;
- нетерминал **T** содержит ссылку на общий базовый тип данных, предназначенный для всего списка. При окончательной свертке указатель на него добавляется в конец **всех** сформированных цепочек типов данных.

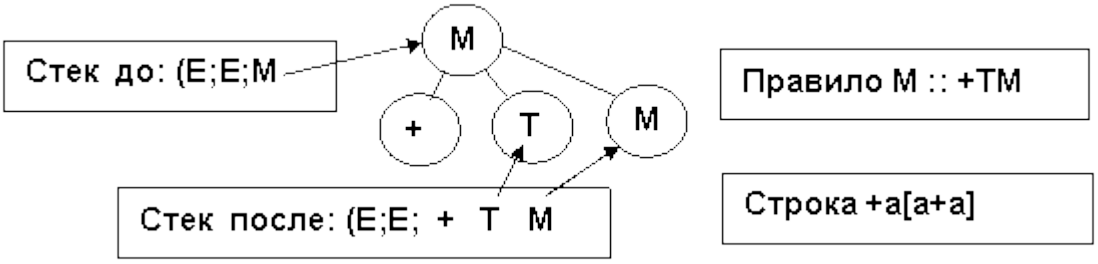
При синтаксическом анализе выражений, в которых заданы действия, выполняемые над объектами программы, семантические таблицы используются как **источник данных.** Например, при выполнении операций свертки, соответствующих операциям над переменной **x1,** будет происходить аналогичная передача указателя на текущую семантику выражения, связанного с нетерминалом. При этом будет проверяться семантическая правильность выполняемых операций. Например, выражение **x1[i]++** будет семантически корректно, поскольку свертка выражения **x1[i]** аналогичным образом пройдет по цепочке **x1 – массив** до записи – указатель **int*,** а для этого ТД операция ++ соответствует в Си++ операциям адресной арифметики. Аналогично, при выполнении сверток действия транслятора по генерации кода или интерпретации будут использовать данные семантических таблиц (например, размерность указуемой переменной типа **int** при генерации кода для операции **++**).

Семантический анализ в нисходящих синтаксических анализаторах

Поскольку семантическая составляющая передается «вверх по дереву», а нисходящий анализ строит его наоборот, сверху вниз, то включение семантической компоненты в синтаксический анализ потребует «хитрых» решений. Рассмотрим некоторые из них:

- при программировании распознавателя методом рекурсивного спуска «обратный ход» семантики соответствует передаче результата рекурсивной функции, каковой является каждый распознаватель. Результатом работы каждого распознавателя является ссылка (указатель, индекс) на описание семантики распознаваемого нетерминала в семантических таблицах. Текущий распознаватель, получая в таком виде семантику нетерминалов правой части правила, формирует из них собственную результирующую семантику (возможно, создавая новые записи в семантических таблицах) и возвращает ее в качестве собственного результата;
- при использовании автоматного распознавателя (магазинного автомата), который не строит дерево, а обходит его сверху вниз, можно в качестве промежуточного уровня построить структуру данных, соответствующую синтаксическому дереву. После чего дерево в терминальные вершины помещаются ссылки на первичную семантику, выполнятся рекурсивный обход дерева, в каждой нетерминальной вершине вызывается семантическая процедура, соответствующая нетерминалу этой вершины. Способ формирования семантики «снизу вверх» аналогичен применяемому в рекурсивном спуске.

Построение синтаксического дерева в явном виде в магазинном автомате нуждается в пояснении. Для этого:

- каждый нетерминал, содержащийся в стеке, имеет ссылку (указатель) на свою вершину – корень недостроенного поддерева;

- при замене в стеке левой части правила на правую для нетерминалов (и терминалов тоже) правой части создаются вершины дерева, ссылки на них помещаются в нетерминалы стека и вершину, связанную с нетерминалом левой части;
- таким образом, нисходящему применению любого правила соответствует достраивание поддерева, определяемого этим правилом.