Тема 5. Семантический анализ

5.1. Основные функции семантического анализа

Фаза семантического анализа предназначена для проверки исходной транслируемой программы на соответствие семантическим соглашениям. Такая проверка называется *статической* (существует также *динамическая* проверка, которая выполняется в процессе выполнения скомпилированной целевой программы).

Типичный перечень семантических соглашений для многих языков программирования [3; 4]:

1. Никакой идентификатор объекта (переменная и т.п.) в программном блоке не должен быть объявлен более одного раза.

2. Определяющим вхождениям идентификатора должны соответствовать их использующие вхождения. Здесь под *определяющим* вхождением понимается вхождение идентификатора в конструкцию, объявляющую этот идентификатор, а *использующим* – остальные вхождения. Например, для любого использующего вхождения должно быть определяющее вхождение, причем идентификатор должен быть объявлен до его использования. Если для определяющего вхождения нет использующего вхождения (не является семантической ошибкой), то вполне естественно сформировать соответствующее предупреждение.

3. Соглашение о соответствии типов объектов, входящих в синтаксическую конструкцию языка программирования, например совместимости типов операндов операции, соответствии типов формальных и фактических параметров процедур.

4. Соглашения, определяющие различные количественные ограничения, например глубину вложенности блоков, ограниченность размерности массивов.

Первые три вида семантических соглашений связаны с *проверкой типов*.

Кроме проверки семантических соглашений в функции семантического анализа включают также *распределение памяти* для размещения объявленных объектов (простых переменных, массивов и других структур данных). По типу объекта можно определить объем памяти, необходимый для его хранения. Во время трансляции эти величины могут использоваться для назначения каждому объекту относительного адреса. Тип и относительный адрес заносятся в запись таблицы символов, соответствующую объекту. Данные переменной длины, такие как строки, или данные, размер которых невозможно определить до начала работы программы (например, динамические массивы), обрабатываются путем резервирования известного фиксированного объема памяти для указателя на данные. При размещении объектов в памяти следует учитывать также необходимость выравнивания адресов. Например, команда сложения целых чисел может требовать, чтобы числа были размещены в определенных позициях памяти, например по адресу, кратному 4.

5.2. Выражения типа

Типы имеют структуру, которую можно представить с использованием выражений типов. *Выражение типа* представляет собой либо фундаментальный (базовый) тип, либо образуется путем применения оператора, называющегося *конструктором типа*, к выражению типа. Множество фундаментальных типов и конструкторов зависит от конкретного языка. Воспользуемся следующим определением выражения типа [1]:

а) фундаментальный тип является выражением типа; типичные фундаментальные типы языка включают предопределенные типы (например, *integer*, *real*, *Boolean*, *char*) и, при необходимости, специальный тип *void*, означающий отсутствие типа;

б) имя типа является выражением типа;

в) выражение типа может быть образовано путем применения конструктора типа к выражению типа;

г) выражение типа может содержать переменные, значениями которых являются выражения типов.

Различные подтипы фундаментальных типов можно рассматривать также как фундаментальные, например тип диапазон вида 1..100 является ограниченным подтипом целого типа. Однако ничто не мешает использовать для этого специальный конструктор типа, отнеся его к создаваемым типам.

Конструктор типа предназначен для создания нового типа на основе фундаментального или другого создаваемого типа. Примеры конструкторов типа (T, T_1, T_2 – выражения типа):

а) *array* (I, T) – определяет массив элементов типа T и множество индексов I, обычно представляющих собой диапазон целых чисел;

б) *pointer* (T) – определяет указатель на объект типа T;

в) $T_1 \times T_2$ – декартово произведение типов T_1 и T_2 (считается левоассоциативным); обычно используются для представления списка типов, например параметров процедур и функций;

г) *record* (($f_1 \times T_1$) $\times$ ($f_2 \times T_2$)) – определяет запись из двух полей: поля с именем f_1 типа T_1 и поля с именем f_2 типа T_2. Описание полей и их типов можно хранить в отдельной таблице символов. Тогда этот конструктор будет иметь более простую форму: *record* (t), где t –таблица символов с информацией о полях этого типа записи (по сути это указатель на таблицу);

д) $T_1 \times T_2 \to T$ – определяет функцию с входными параметрами типов T_1 и T_2 (область определения), возвращающую значение типа T (область значений). По аналогии с конструктором записи информацию о входных параметрах функции можно хранить в отдельной таблице символов. Тогда областью определения будет эта таблица символов: $t \to T$.

Набор подобных конструкторов типа формируется для всех возможных типов языка программирования. Множество правил назначения выражений типов различным конструкциям языка программирования называется *системой типов* данного языка.

Одним из способов представления выражений типов является использование графов, в частности деревьев или ориентированных ациклических графов – дагов (Directed Acyclic Graph), в которых внутренним вершинам соответствуют конструкторы типа, а листьям – фундаментальные типы. Например, тип двумерного массива, представляющего квадратную матрицу вещественных чисел размером 10×10, определяется выражением *array*(1..10, *array*(1..10, *real*)). Дерево (*а*) и даг (*б*) для этого типа представлены на рис. 7.

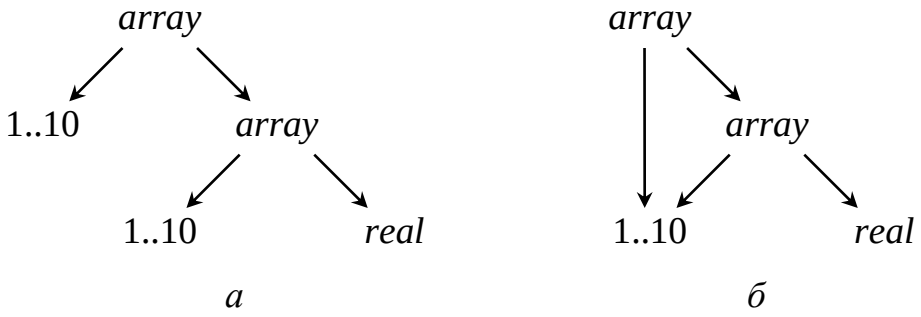

Рис. 7. Представления выражения типа *array* (1..10, *array* (1..10, *real*)):
а – дерево; *б* – даг

Использование дагов может оказаться более предпочтительным, поскольку в этом случае возможна экономия памяти (вместо самого типа в этом случае хранится ссылка на него).

Ряд структур данных, таких, как связанные списки, могут определяться рекурсивно. Графы, представляющие такие типы, могут содержать циклы.

В некоторых случаях для выражений типов можно найти более компактную по сравнению с графом запись. Например, информацию о выражении типа можно закодировать последовательностью двоичных разрядов или в виде некоторой строки символов. При этом следует продумать систему кодирования так, чтобы различные типы не могли иметь одинаковый код. Такую запись выражений типов можно назвать линейным представлением типов, которые можно хранить в специальной таблице типов.

Определив представление типов, можно в таблице символов в качестве значения поля типа идентификатора использовать указатель на дерево или даг, либо ссылку на строку таблицы типов, хранящую линейное представление типов.

5.3. Эквивалентность и преобразование типов

При проверке типов возникает необходимость определения эквивалентности типов. Выделяют два метода определения эквивалентности типов:

1. *Структурная эквивалентность*. Два типа структурно эквивалентны, если они имеют одинаковую структуру, т.е. они либо представляют собой один и тот же фундаментальный тип, либо образованы путем применения одного и того же конструктора к структурно эквивалентным типам.

2. *Именная эквивалентность*. Два типа эквивалентны, если они объявлены с помощью одного и того же имени типа. Именная эквивалентность является более ограничивающей формой типизации, чем структурная, и обычно используется в строго типизированных языках.

Рассмотрим следующее объявление типов в синтаксисе языка Паскаль:

```
type
   TSpeed = real;
   TSize = real;
var
   Speed : TSpeed;
   Size : TSize;
```

Переменные *Speed* и *Size* структурно эквивалентны, поскольку представляют один и тот же фундаментальный тип *real*, но не эквивалентны по критериям именной эквивалентности.

Если выражения типа содержат переменные, то при определении эквивалентности выражений возникает задача подстановки выражений типа вместо этих переменных. Такая задача известна как *унификация* выражений. Алгоритм унификации выражений типа и примеры его применения для проверки структурной эквивалентности типов подробно рассмотрены в работах [1; 2].

Фрагмент алгоритма проверки структурной эквивалентности может быть представлен следующим образом:

function *StrEquiv*(*s*, *t*): Boolean
begin
  if *s* и *t* один и тот же базовый тип **then return**(**true**)
  else if $s = array(i_1, t_1)$ **and** $t = array(i_2, t_2)$ **then**
    return $StrEquiv(i_1, i_2)$ **and** $StrEquiv(t_1, t_2)$
  else if $s_1 \times s_2$ **and** $t_1 \times t_2$ **then**
    return $StrEquiv(s_1, t_1)$ **and** $StrEquiv(s_2, t_2)$
  else if …
    return …
  else return(**false**)
end

Если надо игнорировать границы и типы индексов массивов, то можно соответствующие строки переписать очевидным образом:

  else if $s = array(i_1, t_1)$ **and** $t = array(i_2, t_2)$ **then**
    return $StrEquiv(t_1, t_2)$

Приведенный алгоритм рекурсивно сравнивает структуры выражений типов без проверки цикличности, так что он может быть применен к представлению в виде дерева или дага. Структурная эквивалентность типов с циклами может быть проверена и использованием алгоритма унификации [1; 2].

Для проверки именной эквивалентности все типы должны иметь имена, что приведет к тривиальной проверке на совпадение имен типов. Для этого компилятор вынужден будет генерировать временные имена типов. Рассмотрим объявления

```
var
    MyArr: array[1..20] of integer;
    MyPnt: ^Integer;
```

надо будет при трансляции реализовать создание неявных имен типов, чтобы эти объявления трактовалось следующим образом (T1 и T2 – неявные имена типов, создаваемые компилятором, чтобы тип задавался только именем):

```
type
    T1 = array[1..20] of integer;
    T2 = ^Integer;
var
    MyArr: T1;
    MyPnt: T2;
```

Когда в тех или иных операциях или операторах присутствуют данные, относящиеся к различным типам, возникает вопрос о совместимости и преобразовании типов. Совместимость типов означает, что для объектов разных типов возможно приведение типа объекта к другому типу. Преобразование одного типа в другой называется *неявным*, если оно должно выполняться компилятором автоматически. Для этого в семантических действиях схемы трансляции должны быть предусмотрены соответствующие операции по преобразованию типов. Преобразование называется *явным*, если в программе специально указывается необходимость преобразования. Для этого язык должен иметь специальные средства (обычно они выглядят как применение функции или процедуры) для реализации подобных преобразований. Явное преобразование типов выполняется в процессе выполнения программы, а не в процессе компиляции, поэтому, если язык предусматривает только явные преобразования (характерно для строго типизированных языков), в схемах трансляции отсутствуют действия, связанные с преобразованием типов.

Если при преобразовании типов нет потери информации, такое преобразование называется *расширяющим*, например преобразование целого типа в вещественный тип. Преобразование типов называется *сужающим*, если оно может привести к потере информации, например при преобразовании вещественного типа в целый тип.

5.4. СУО для проверки типов

Процесс построения *L*-атрибутного СУО для проверки типов рассмотрим отдельными фрагментами: сначала объявления типов, затем проверку типов в выражениях и операторах языка.

Объявления типов различных объектов (переменных, функций и т.п.) обычно сгруппированы в декларативной части транслируемой программы. В ряде языков программирования могут быть и другие способы объявления типов (например, объявления типов рассредоточены в пределах исходной программы), вплоть до определения типа объекта по умолчанию (т.е. тип объекта явно не объявляется).

Очевидная грамматика для объявления типа для некоторого списка переменных – это грамматика со следующими продукциями:

$D \to L$ **:** T **;**

$L \to$ **id** | L **,** **id**.

Несмотря на то что это *LR*-грамматика, на ее основе нельзя получить *L*-атрибутное СУО, поскольку идентификаторы порождаются нетерминалом *L*, но тип как синтезируемый атрибут нетерминала *T* в поддереве *L* дерева разбора еще не известен. Эта проблема решается соответствующим преобразованием грамматики:

$D \to$ **id** L **;**

$L \to$ **,** **id** L | **:** T.

Как получили из эту грамматику

$D \to L$ **:** T **;** $D \to$ **id** L **;**

$L \to$ **id** | L **,** **id** $L \to$ **,** **id** L | **:** T

	Доказательство, что $L \to$ **id** | L **,** **id** эквивалентны $L \to$ **id** | **id** **,** L
а) представим $L \to$ **id** | L **,** **id** как $L \to$ **id** | **id** **,** L б) замена вхождений L $D \to$ **id** **:** T **;** | **id** **,** L **:** T **;** в) факторизация $D \to$ **id** (**:** T | **,** L **:** T) **;** $D \to$ **id** X **;** $X \to$ **:** T | **,** L **:** T г) замена вхождений L $X \to$ **:** T | **,** **id** **:** T | **,** **id** **,** L **:** T д) факторизация $X \to$ **:** T | **,** **id** (**:** T | **,** L **:** T), т.е. $X \to$ **:** T | **,** **id** X Получили $D \to$ **id** X **;** $X \to$ **:** T | **,** **id** X Отличие только в обозначении X (вместо L)	а) устраняем левую рекурсию $L \to$ **id** | **id** X $X \to$ **,** **id** | **,** **id** X б) факторизация $X \to$ **,** (**id** | **id** X), т.е. $X \to$ **,** L в) замена вхождений X, т.е. получаем $L \to$ **id** | **id** **,** L что и требовалось доказать.

Теперь тип рассматривается как синтезируемый атрибут *type* нетерминалов *L* и *T*, который можно внести в таблицу символов каждого идентификатора, порождаемого *L*, как это показано в СУО в табл. 6.

Таблица 6

L-атрибутное СУО для объявления типа

Продукция	Семантические правила
$D \rightarrow$ **id** L **;**	*AddType* (**id.***pnt*, *L.type*)
$L \rightarrow$ **,** **id** L_1	*AddType* (**id.***pnt*, L_1*.type*); *L.type* := L_1*.type*
$L \rightarrow$ **:** T	*L.type* := *T.type*

В данном СУО процедура *AddType* (**id.***pnt*, *L.type*) сохраняет тип *L.type* для идентификатора **id** в записи таблицы символов, на которую указывает атрибут терминала **id.***pnt* (*pnt* – атрибут токена **id**, предоставляемый лексическим анализатором). СУО легко можно дополнить семантическими правилами для сохранения в таблице символов размера типа, диапазона значений, адреса выделенной памяти, числа измерений для массивов и тому подобного, дополнив символы грамматики соответствующими атрибутами.

Приведенный пример иллюстрирует то, что в процессе разработки СУО (даже если его основой является грамматика соответствующего класса) могут потребоваться дополнительные преобразования грамматики, чтобы СУО стало *L*-атрибутным.

В качестве другого примера рассмотрим продукции, используемые для объявления массива:

ArrType → **arr [** *LstInd* **] of** *Type*

LstInd → *SmpType* **|** *SmpType* **,** *LstInd*

SmpType → **id | num rng num**

Эти продукции не подходят для применения конструктора типа *array*(*I, T*), определяющего массив с элементами типа *T* и множеством индексов *I*, представляющим собой диапазон целых чисел, а также для вычисления размера массива (путем умножения размера элемента на количество элементов) и числа измерений. Проблема решается, если представить указанные продукции следующим образом:

ArrType → **arr [** *LstInd*

LstInd → *SmpType* **] of** *Type* **|** *SmpType* **,** *LstInd*

SmpType → **id | num rng num**

Теперь можно построить СУО, которое для массива определяет тип (*type*), размер типа (*width*), число измерений (*ndim*) как синтезируемые атрибуты нетерминалов:

Продукция	Семантические правила
ArrType → **arr [** *LstInd*	*ArrType*.*type* := *LstInd*.*type*; *ArrType*.*width* := *LstInd*.*width*; *ArrType*.*ndim* := *LstInd*.*ndim*
LstInd → *SmpType* **] of** *Type*	*m* := *SmpType*.*low*; *n* := *SmpType*.*high*; *LstInd*.*type* := *array*(*m*..*n*, *Type*.*type*); *LstInd*.*width* := (*n* – *m* + 1) * *Type*.*width*; *LstInd*.*ndim* := 1
LstInd → *SmpType* **,** $LstInd_1$	*m* := *SmpType*.*low*; *n* := *SmpType*.*high*; *LstInd*.*type* := *array*(*m*..*n*, $LstInd_1$.*type*); *LstInd*.*width* := (*n* – *m* + 1) * $LstInd_1$.*width*; *LstInd*.*ndim* := $LstInd_1$.*ndim* + 1
SmpType → **id**	**if** *GetType*(**id.***pnt*) ≠ диапазон **then** *type_error*; *SmpType*.*low* := *GetLow*(**id.***pnt*); *SmpType*.*high* := *GetHigh*(**id.***pnt*)
SmpType → $\mathbf{num}_1$ **rng** $\mathbf{num}_2$	**if** $\mathbf{num}_1$.*type* ≠ *integer* **or** $\mathbf{num}_2$.*type* ≠ *integer* **then** *type_error*; **if** $\mathbf{num}_1$.*val* > $\mathbf{num}_2$.*val* **then** *type_error*; *SmpType*.*low* := $\mathbf{num}_1$.*val*; *SmpType*.*high* := $\mathbf{num}_2$.*val*

В приведенном СУО предполагается, что выполнение действия *type_error* (ошибка типа) приводит к формированию соответствующего сообщения об ошибке и немедленному прекращению выполнения всех остальных семантических правил. Синтезируемые атрибуты *low* и *high* нетерминала *SmpType* предназначены для хранения соответственно нижней и верхней границ диапазона. Функции *GetType*, *GetLow* и *GetHigh* предоставляют соответственно значения типа, нижней и верхней границ диапазона из записи таблицы символов, на которую указывает **id.***pnt*. Терминал **num** имеет синтезируемые атрибуты *type* и *val*, представляющие собой соответственно тип и значение числовой константы (доступ к типу и значению константы обеспечивается по значению атрибута токена, указывающего на соответствующую запись в таблице символов).

Варианты СУО и СУТ, обеспечивающих проверку типов, распределение памяти и генерацию промежуточного кода, для основных типов данных и операторов языков программирования можно найти в [1].

СУО для проверки типов арифметических выражений будем рассматривать в предположении, что язык является строго типизированным, и нет никаких неявных преобразований типов, т.е. при выполнении арифметических операций совместимы только выражения, имеющие один и тот же тип. Тогда достаточно просто построить СУО, представленное в табл. 7.

Таблица 7

СУО для проверки типов арифметических выражений

Продукция	Семантические правила
$S \to E\perp$	
$E \to E_1 + T$	**if** E_1**.***type* = T**.***type* **then** E**.***type* := E_1**.***type* **else** *type_error*
$E \to T$	E**.***type* := T**.***type*
$T \to T_1 * F$	**if** T_1**.***type* = F**.***type* **then** T**.***type* := T_1**.***type* **else** *type_error*
$T \to T_1$ **mod** F	**if** T_1**.***type* = *integer* **and** F**.***type* = *integer* **then** T**.***type* := *integer* **else** *type_error*
$T \to F$	T**.***type* := F**.***type*
$F \to (E)$	F**.***type* := E**.***type*
$F \to$ **num**	F**.***type* := *GetType* (**num.***pnt*)
$F \to$ **id**	F**.***type* := *GetType* (**id.***pnt*)

Если для каких-либо операций допустимы только выражения определенных типов, то семантические правила необходимо дополнить проверкой этих ограничений, как это сделано, например, в продукции для операции **mod** (вычисление остатка от целочисленного деления).

Проверка типов в логических (булевых) выражениях зависит от места их использования (синтаксического контекста).

Если логическое выражение используется в правой части оператора присваивания, то вычисляется его значение. В этом случае вычисление логического выражения можно реализовать подобно вычислению арифметического выражения и СУО для проверки типов будет похоже на приведенное выше. Можно также вместо вычисления значения использовать команды переходов, когда значение выражения устанавливается в зависимости от достигнутой программой позиции.

Если же логическое выражение используется в качестве условия в управляющих операторах (**if**, **while** и т.п.) для управления ходом выполнения программы, то вместо вычисления его значения используются команды условных и безусловных переходов, которые в зависимости от значения логического выражения передают управление в ту или иную позицию кода. Это все реализуется в процессе генерации промежуточного кода и, следовательно, не требует специальной проверки типов для логических выражений, как это делается для арифметических выражений. Для определения синтаксического контекста использования логических выражений можно использовать разные приемы [1]: применение разных нетерминалов для обозначения логических и арифметических выражений, добавление специальных наследуемых атрибутов или установка соответствующего флага в процессе синтаксического анализа.

Проверка типов для операторов языков программирования затрагивает в основном только те операторы, в которых есть ограничения на типы используемых выражений, причем эти ограничения не реализованы синтаксически. Это чаще всего управляющие операторы, где обычно используемые в качестве условий выражения должны иметь тип *Boolean*. Оператор присваивания также требует совместимости или совпадения типов левой и правой частей. В качестве примера для таких операторов можно представить следующее СУО (при условии, что нет синтаксических ограничений на тип выражения *E*) (табл. 8).

Таблица 8

СУО для проверки типов управляющих операторов

Продукция	Семантические правила
$S \rightarrow$ **id** := E	**if** *GetType* (**id.***pnt*) ≠ *E.type* **then** *type_error*
$S \rightarrow$ **if** E **then** S	**if** *E.type* ≠ *Boolean* **then** *type_error*
$E \rightarrow$ **while** E **do** S	**if** *E.type* ≠ *Boolean* **then** *type_error*