Лекция 6

Семантический анализ

§33. Постановка задачи семантического анализа

Определение. *Символ* (symbol) – это именованная сущность в программе, определяемая парой $\langle \text{name}, \text{info} \rangle$, где name – идентификатор сущности, info – описание сущности.

Примеры сущностей: переменные, типы данных, функции, модули и т.п.

Две и более сущности могут иметь одно идентификатор. При этом им соответствуют разные символы: $\langle \text{name}, \text{info1} \rangle$, $\langle \text{name}, \text{info2} \rangle$ и т.д.

Реже бывает, что два и более идентификатора обозначают одну сущность (тогда говорят, что сущность имеет псевдонимы). При этом каждому идентификатору соответствует свой символ: $\langle \text{name1}, \text{info} \rangle$, $\langle \text{name2}, \text{info} \rangle$ и т.д.

Семантика языка программирования определяет, в каких узлах дерева синтаксического разбора программы может использоваться идентификатор того или иного символа.

Если идентификатор символа может использоваться в некотором узле, говорят, что символ *виден* в этом узле.

В большинстве языков программирования видимость символа – это статическая характеристика символа, то есть она определяется во время компиляции.

Категории символов, для которых происходит выделение памяти во время выполнения программы (например, переменные), в дополнение к видимости имеют ещё одну характеристику – время жизни.

Определение. *Время жизни* (lifetime, extent) символа – это часть времени выполнения программы от момента выделения памяти до момента освобождения памяти.

Время жизни – это динамическая характеристика, то есть оно определяется во время выполнения программы.

Следует понимать, что во время выполнения программы символ может присутствовать в памяти в нескольких экземплярах (например, локальные переменные в рекурсивных функциях).

Определение. *Таблица символов* (symbol table) для узла дерева синтаксического разбора – это отображение идентификаторов символов, видимых в этом узле, в описания соответствующих этим символам сущностей.

Таблицы символов в компиляторах реализованы в виде хеш-таблиц. Идентификаторы являются ключами этих хеш-таблиц.

Лексический анализатор связывает с каждым идентификатором уникальное целое число – код идентификатора. Это существенно ускоряет вычисление хеш-функции.

Иногда семантика языка программирования допускает, что множества идентификаторов разных категорий сущностей, видимых в одном и том же узле дерева, могут пересекаться.

Пример. В C теги структур могут совпадать с именами переменных:

```
struct A
{
    int x, y;
};

void f()
{
    struct A A;
    A.x = 5;
}
```

В таком случае нужно иметь по отдельной таблице символов для каждой категории сущностей.

Семантический анализ программы решает две задачи:

1. построение таблицы символов для каждой области в дереве синтаксического разбора программы и генерация ошибок для узлов дерева, в которых используются идентификаторы, не являющиеся ключами соответствующих таблиц символов;

2. проверка допустимости применения операций к сущностям программы (для языков со статической типизацией может потребоваться вычисление типов для всех узлов, соответствующих выражениям в программе).

§34. Области и локальные таблицы символов

Определение. *Область* (scope) – это множество узлов дерева синтаксического разбора программы, в которых по семантике языка программирования видимы одни и те же символы.

Использование областей увеличивает эффективность семантического анализатора благодаря тому, что для всех узлов, принадлежащих области, строится одна таблица символов.

Определение. *Вложенная область* (nested scope) – это область, узлы которой являются дочерними по отношению к узлу другой области.

Вложенная область наследует таблицу символов родительского узла, дополняя её собственными символами. При этом собственные символы могут экранировать часть унаследованных символов.

Определение. Область A является *открытой* для области B, если область B непосредственно вложена в область A, или если специальная конструкция языка открывает область A для области B.

Пример. Операция доступа к полю структуры открывает область, содержащую символы, описывающие поля. В языке Pascal поля структуры открываются оператором with:

```
type
  R = record
        a, b: integer
      end;
var
  x: R;
begin
  with x do
  begin
    a := 5;
    b := 10;
  end;
end.
```

Определение. *Эффективные символы* области – это все символы, видимые в узлах области.

Определение. *Собственные символы* области – это символы, объявленные внутри области.

Определение. *Унаследованные символы* области – это эффективные символы областей, открытых для этой области.

Определение. *Экранированные символы* области – это унаследованные символы, идентификаторы которых совпадают с идентификаторами собственных символов.

Таким образом, множество E эффективных символов области вычисляется по формуле
$E = P \cup I \setminus S$,
где P – множество собственных символов;
I – множество унаследованных символов;
S – множество экранированных символов.

Пример. Вложенные области в программе на языке Pascal.

```
program P;
var x, y: integer;

  procedure F;
  var x, a, b: real;

    procedure G;
    var a, c: char;
    begin
      (* [a->char, c->char] +
         унаследованные [x->real, a->real, b->real] +
         унаследованные [x->integer, y->integer] *)
    end;

  begin
    (* [x->real, a->real, b->real] +
       унаследованные [x->integer, y->integer] *)
  end;

begin
  (* [x->integer, y->integer] *)
end.
```

Определение. *Локальная таблица символов* (local symbol table) – это таблица символов, в которой перечисляются только собственные символы области и имеются ссылки на локальные таблицы областей, открытых для данной области.

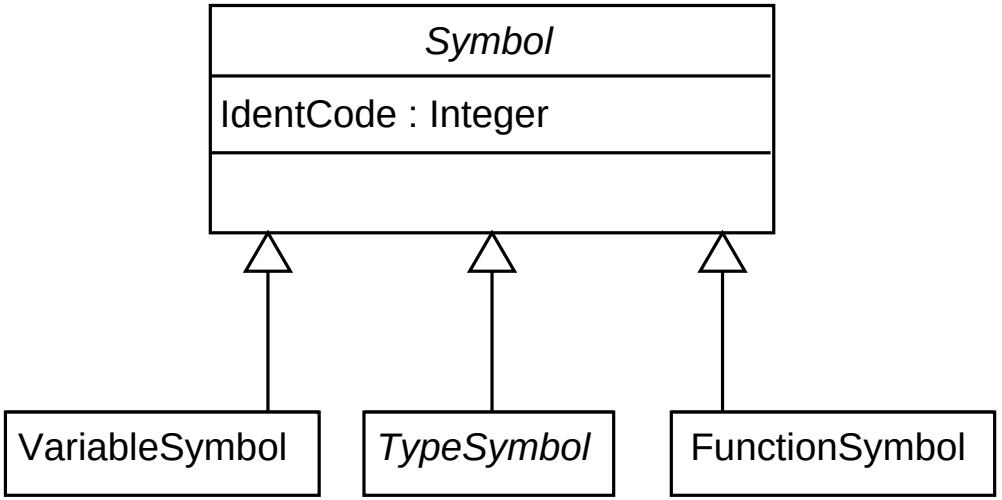
Локальные таблицы позволяют экономить память за счёт того, что каждый символ входит только в одну таблицу.

На выходе семантического анализатора, строящего локальные таблицы, получается ациклический направленный граф локальных таблиц. Корнями этого графа являются локальные таблицы самых внешних областей.

Поиск символа для узла дерева синтаксического разбора начинается с обращения к локальной таблице области X, в которую входит узел, и если символ в ней не найден, то поиск рекурсивно запускается для локальных таблиц областей, открытых для X.

§35. Проектирование объектно-ориентированных таблиц символов

Символ представляется абстрактным классом Symbol, от которого наследуются классы основных категорий символов.



Поле IdentCode содержит код, присвоенный идентификатору лексическим анализатором.

Класс SymbolTable представляет локальную таблицу символов.

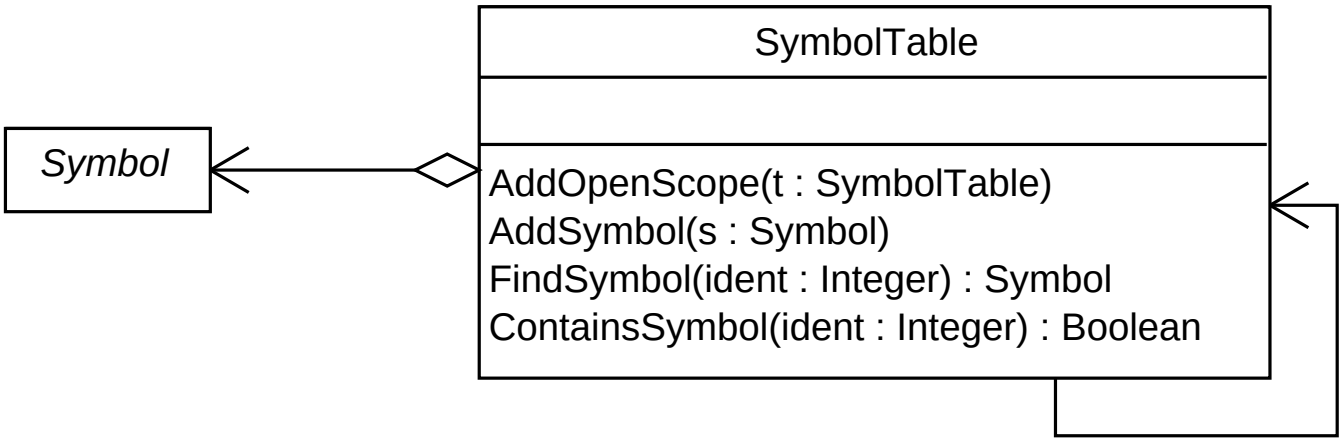
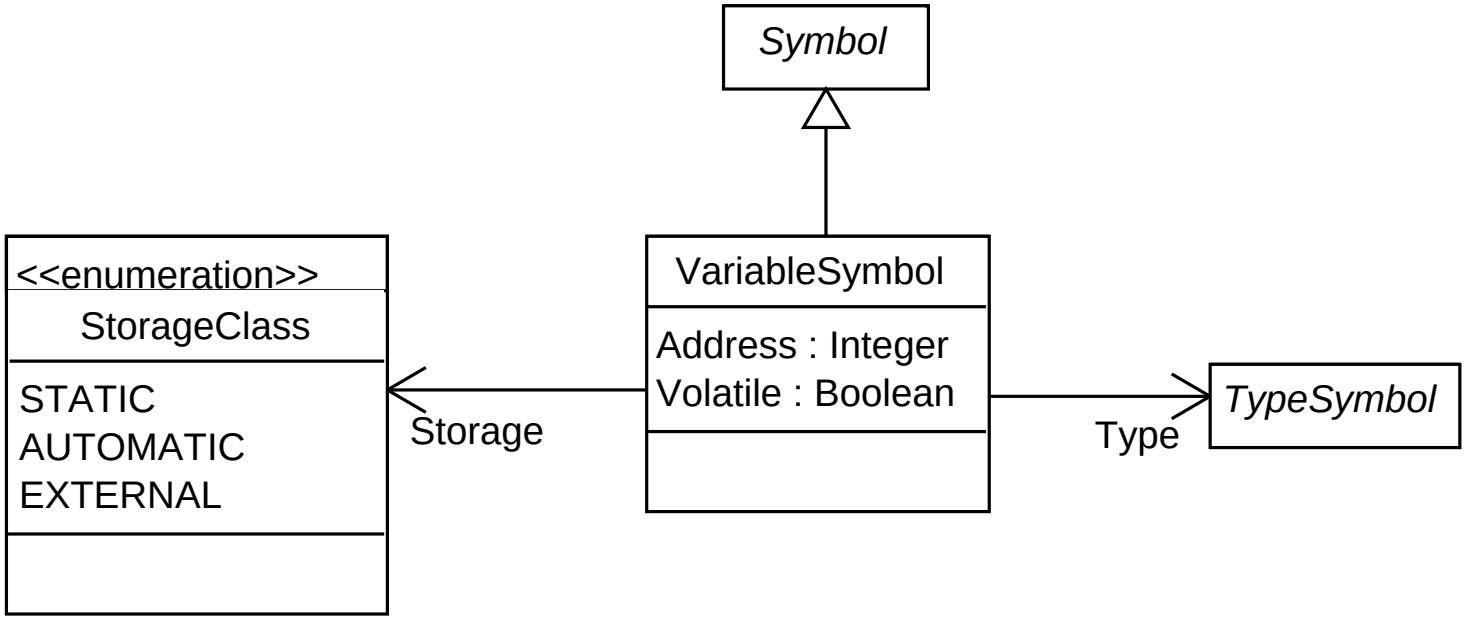


Таблица символов содержит хеш-таблицу символов и ссылки на таблицы символов открытых областей.

Метод AddOpenScope добавляет открытую область, метод AddSymbol добавляет новый символ, метод FindSymbol выполняет поиск эффективного символа по идентификатору, а метод ContainsSymbol проверяет наличие собственного символа по идентификатору.

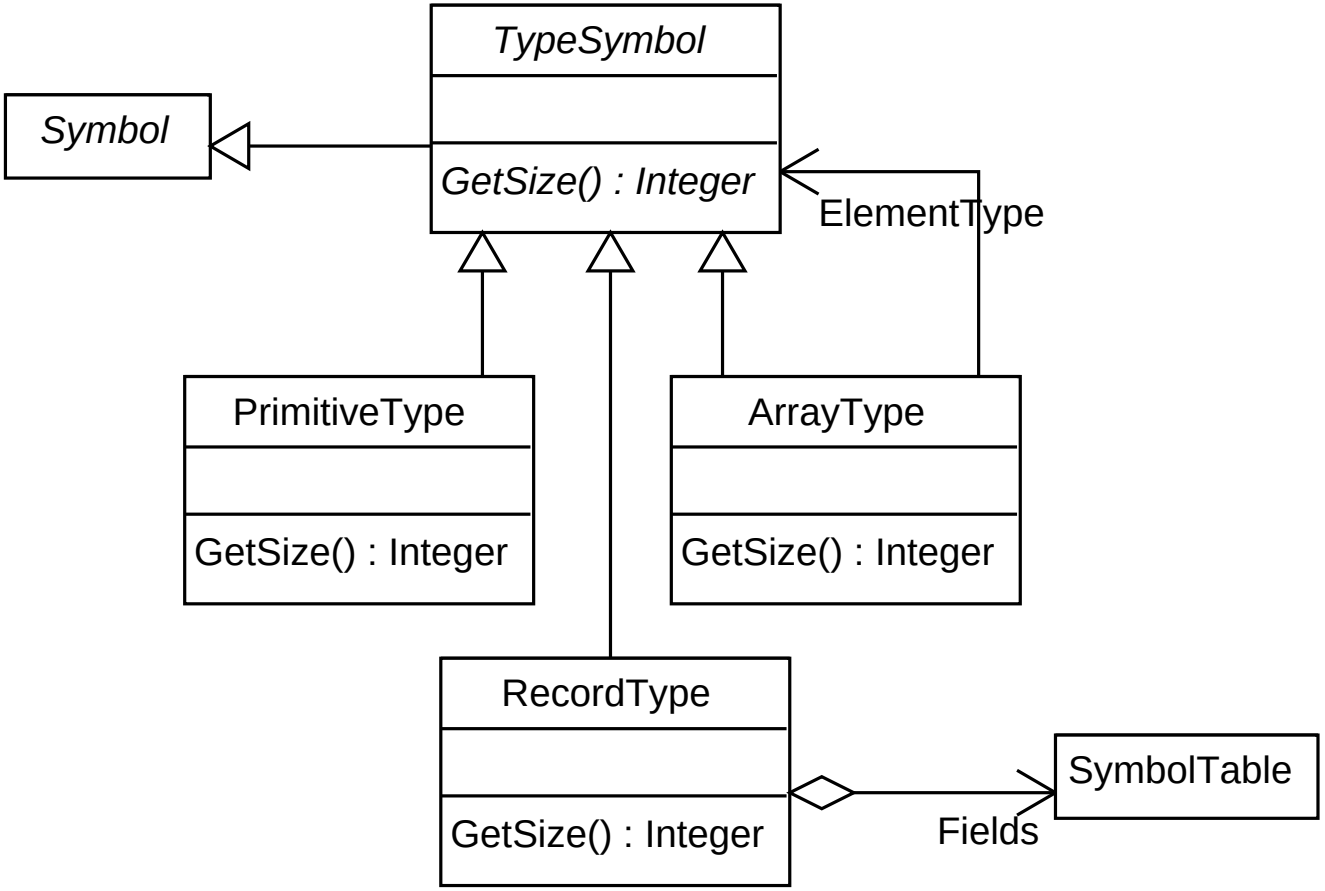
Класс VariableSymbol служит для представления переменных.



Перечисление StorageClass содержит варианты классов памяти для переменной.

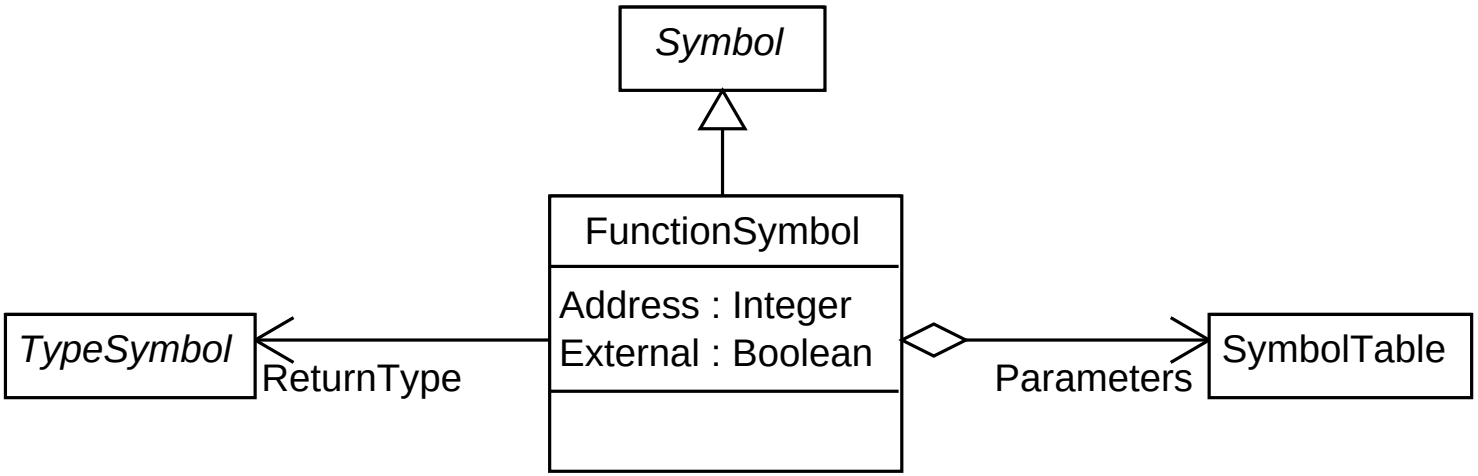
Свойство Address зарезервировано для адреса переменной и заполняется во время работы фазы распределения памяти.

От абстрактного класса TypeSymbol наследуют классы, представляющие примитивные и составные типы.



Предполагается, что у безымянных типов поле IdentCode содержит какое-нибудь специальное значение (например, -1).

Класс FunctionSymbol служит для представления функций.



Свойство Address зарезервировано для адреса функции и заполняется во время работы фазы генерации кода на целевом языке.

Сводная диаграмма классов: