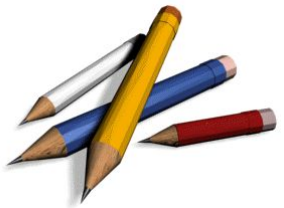
Семантический анализ

Семантический анализ

С целью упрощения синтаксических анализаторов большинство из них основано на контекстно-свободных грамматиках (тип 2). Однако большинство языков программирования содержат те или иные контекстно-зависимые элементы (тип 1).

Например, во многих языках идентификаторы не могут применяться, если они не описаны, также имеются ограничения в отношении способов употребления в программе значений различных типов.



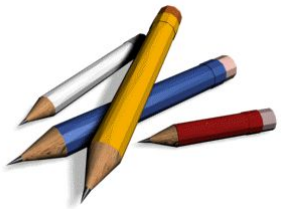
Семантический анализ

Алгоритмы анализа, основанные на контекстно-зависимых грамматиках, имеют низкую эффективность (как по времени работы, так и по используемой памяти). Поэтому на практике они не используются.

Все реально существующие трансляторы выполняют анализ исходной программы в два этапа:

1) синтаксический анализ на основе распознавателя для одного из классов КС-грамматик;

2) семантический анализ.

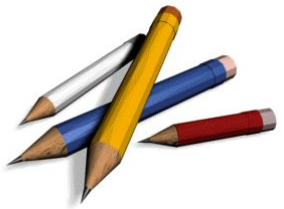


Семантический анализ

Основная задача семантического анализа – определить, соответствует ли программа семантическим соглашениям языка.

На этапе семантического анализа выполняется несколько типов проверок.

Конкретный состав проверяемых соглашений зависит от семантики языка.



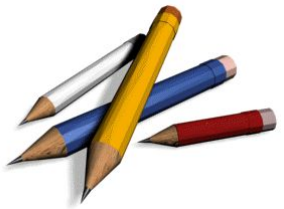
Проверка соответствия типов

Необходимо убедиться в том, что:

– арифметические операторы применяются к операндам допустимого типа;

Пример.

```
double a,b,c;
a=6.25;
b=2.25;
c=a%b;
```



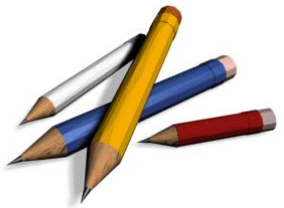
Семантический анализ

– при присваивании тип переменной соответствует типу арифметического выражения;

Пример.

```
int a;
double b,c;
a=b+c;
```

На этапе проверке соответствия типов также выявляется необходимость выполнения неявных преобразований.



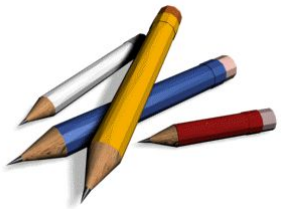
Семантический анализ

– тип фактического параметра функции соответствует типу формального параметра;

– тип возвращаемого значения соответствует типу, указанному в заголовке функции;

и т.д.

Транслятор должен сообщать об ошибке, если оператор применяется к несовместимому с ним операнду (например при сложении переменных, представляющих собой массив и функцию).



Проверка правильности употребления имен

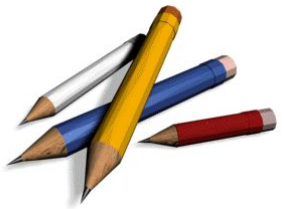
Необходимо убедиться в том, что:

– используемый идентификатор был ранее объявлен и находится в области видимости;

– отсутствует повторное объявление идентификаторов (например, в операторах case, enum);

Пример.

```
int a;
for(a=1;a<=10;a++) {
printf("a=%d\n",a);
double a=3.1415;
printf("a=%f\n",a);
}
```

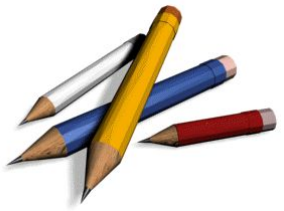


Проверка правильности передачи управления

Необходимо убедиться, что осуществляется допустимая передача управления.

Пример.

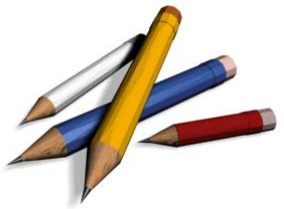
В С-подобных языках программирования оператор *break* передает управление за пределы наиболее вложенной инструкции *while*, *for* или *switch*; если же таковые отсутствуют, то выводится сообщение об ошибке.



Проверка элементарных смысловых норм, не связанных со входным языком

– каждая объявленная переменная или константа должна хотя бы один раз использоваться в программе;

– каждая переменная должна быть определена до ее первого использования при любом ходе выполнения программы (т.е. первому использованию переменной должно всегда предшествовать присвоение ей какого-либо значения);



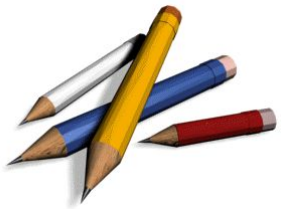
Семантический анализ

– результат функции должен быть определен при любом ходе ее выполнения;

– операторы ветвления и выбора должны предусматривать возможность хода выполнения программы по каждой из своих ветвей;

– операторы цикла должны предусматривать возможность завершения цикла;

– каждый оператор в исходной программе должен иметь возможность хотя бы один раз выполниться.



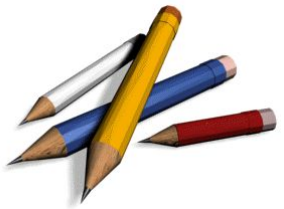
Способы организации семантического анализа

Входными данными для семантического анализа служат:

– таблица идентификаторов;

– результаты разбора синтаксических конструкций входного языка (синтаксическое дерево программы).

Результаты семантического анализа фиксируются в какой-либо внутренней форме представления программы, чаще всего в виде *атрибутированного синтаксического дерева.*



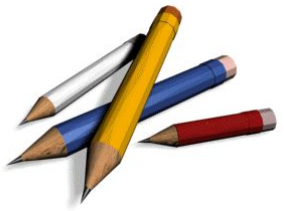
Реализация проверки правильности употребления имен

Определяющей (описывающей) реализацией идентификатора называет синтаксическая конструкция, содержащая объявление идентификатора, например:

int a;

Все остальные конструкции, содержащие идентификатор, являются *прикладными (использующими) реализациями*, например:

a=4 или **a+b** или **read(a)**



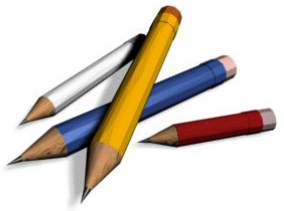
Семантический анализ

Многие современные языки высокого уровня обладают следующими свойствами:

– определяющая реализация идентификатора появляется (текстуально) раньше любой прикладной реализации;

– при наличии прикладной реализации идентификатора соответствующая определяющая реализация разыскивается в наименьшем включающем блоке, в котором содержится этот идентификатор;

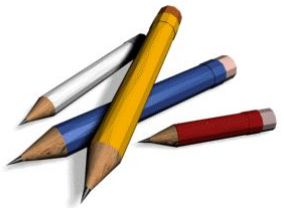
– в одном и том же блоке идентификатор не может описываться более одного раза



Семантический анализ

При анализе определяющей реализации идентификатора, семантический анализатор должен поместить характеристики объекта в *таблицу идентификаторов*.

При анализе прикладной реализации в таблице идентификаторов осуществляется поиск элемента, соответствующего определяющей реализации объекта, для определения его типа и (возможно) других признаков, требующихся во время анализа.



Структура таблицы идентификаторов должна соответствовать блочной структуре программы

```
{
  int a,b;

  {
    char c,d;
  }

  {
    float e,f;
  }

}
```

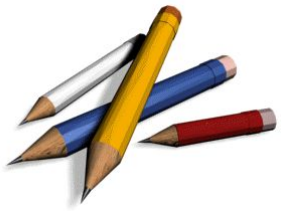
a	int
b	int

a	int
b	int

c	char
d	char

a	int
b	int

e	float
f	float



Семантический анализ

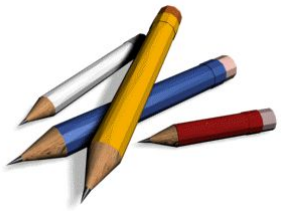
Во многих языках программирования один и тот же идентификатор может использоваться в различных частях программы для представления различных объектов.

В таких случаях структура программы помогает различать эти объекты, например:

```
{
 int a;
 ...
}
...
{
 char a;
 ...
}
```

a	int

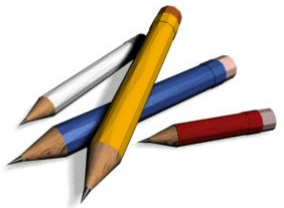
a	char



Семантический анализ

В качестве структуры данных для таблицы идентификаторов очень удобен стек, каждым элементом которого служит элемент этой таблицы.

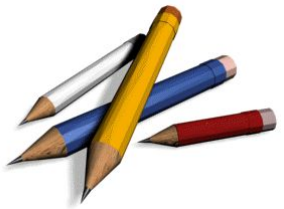
При встрече с описанием соответствующий элемент таблицы идентификаторов помещается в вершину стека. Перед занесением необходимо проверить, нет ли в последнем блоке (C/C++) или во всем стеке (Java) идентификатора с таким же именем.



Семантический анализ

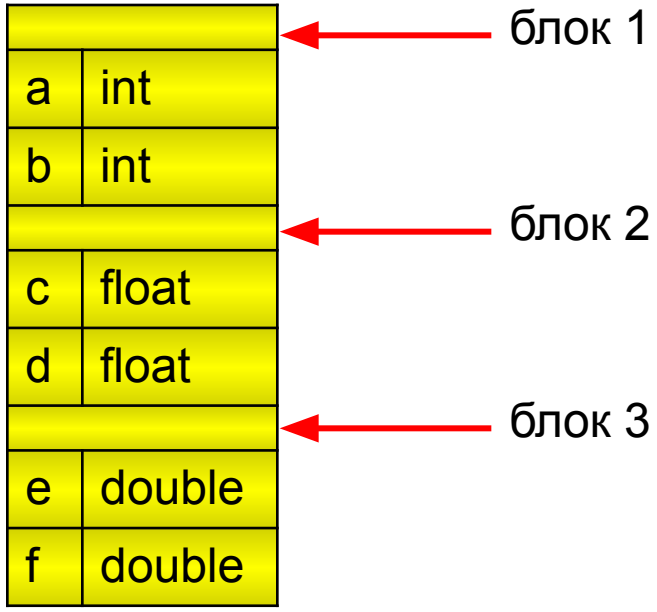
При выходе из блока все элементы таблицы идентификаторов, соответствующие описаниям в этом блоке, удаляются из стека. Указатель же стека возвращается в положение, которое он имел при вхождении в блок.

В результате в любой момент разбора элементы таблицы, соответствующие всем текущим идентификаторам, находятся в стеке, а связанные с ним прикладные и определяющие реализации идентификаторов требуют поиска в стеке в направлении от вершины к дну стека.

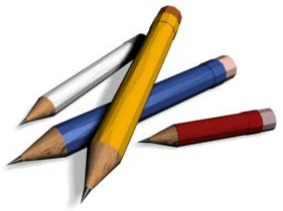


Семантический анализ

```
{ // блок 1
  int a,b;
  // ...
  {
    // блок 2
    float c,d;
    // ...
    {
      // блок 3
      double e,f;
      // ...
    }
  }
}
```



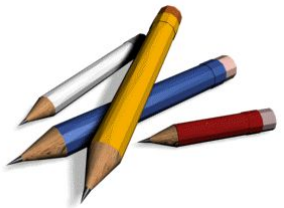
Для удобства организации поиска и обработки стек может содержать граничные элементы, соответствующие началу каждого блока



Другим примером реализации таблицы идентификаторов может служить списочная структура данных "список списков" ("стек стеков").

Каждому вложенному блоку в такой структуре будет соответствовать отдельный список, содержащий элементы таблицы идентификаторов.

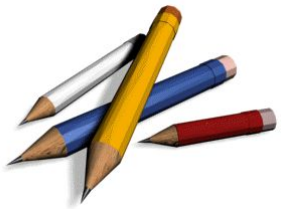
При появлении очередного блока создается новый список идентификаторов, который будет заполняться по мере анализа блока.



Семантический анализ

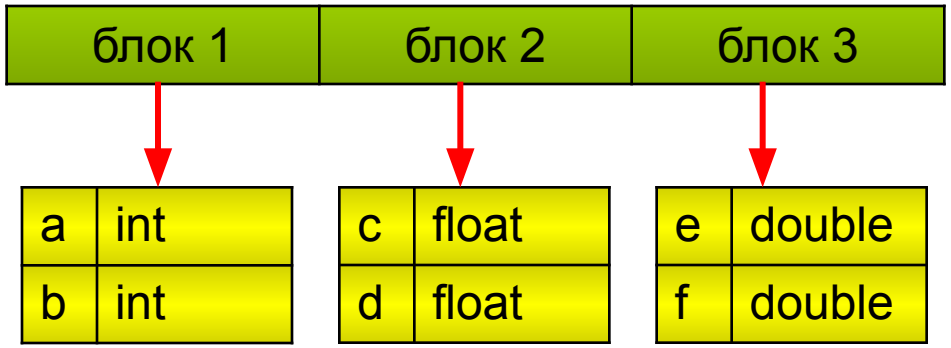
Перед занесением идентификатора необходимо проверить, нет ли в последнем списке (C/C++) или во всей структуре (Java) идентификатора с таким же именем.

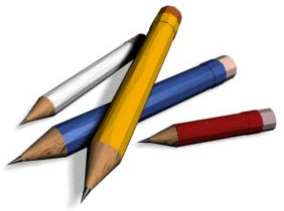
При выходе из блока удаляется список идентификаторов, соответствовавший данному блоку.



Семантический анализ

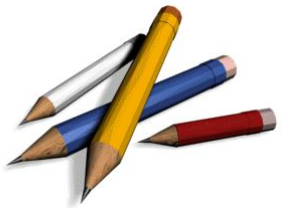
```
{ // блок 1
  int a,b;
  // ...
  {
    // блок 2
    float c,d;
    // ...
    {
      // блок 3
      double e,f;
      // ...
    }
  }
}
```





Реализация проверки совместимости типов в арифметическом выражении

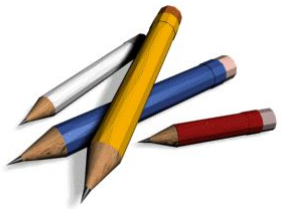
Исходными данными для анализа является синтаксическое дерево арифметического выражения, результатом – атрибутированное синтаксическое дерево.



Семантический анализ

Каждая вершина синтаксического дерева выражения должна помечаться соответствующим ей *типом*.

Для этого используется алгоритм *полного обхода графа в глубину*.

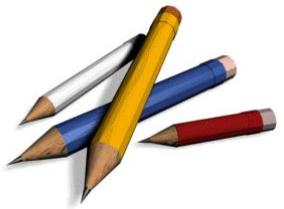


Семантический анализ

Тип каждой вершины определяется следующим образом:

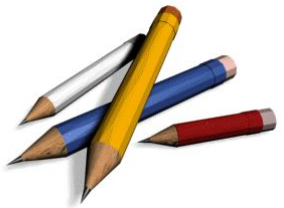
1. Для "листьев", соответствующих константам, тип определяется на основании результатов *лексического анализа*.

2. Для "листьев", соответствующих идентификаторам, тип определяется на основании *семантического анализа* (исходя из содержимого таблицы идентификаторов).

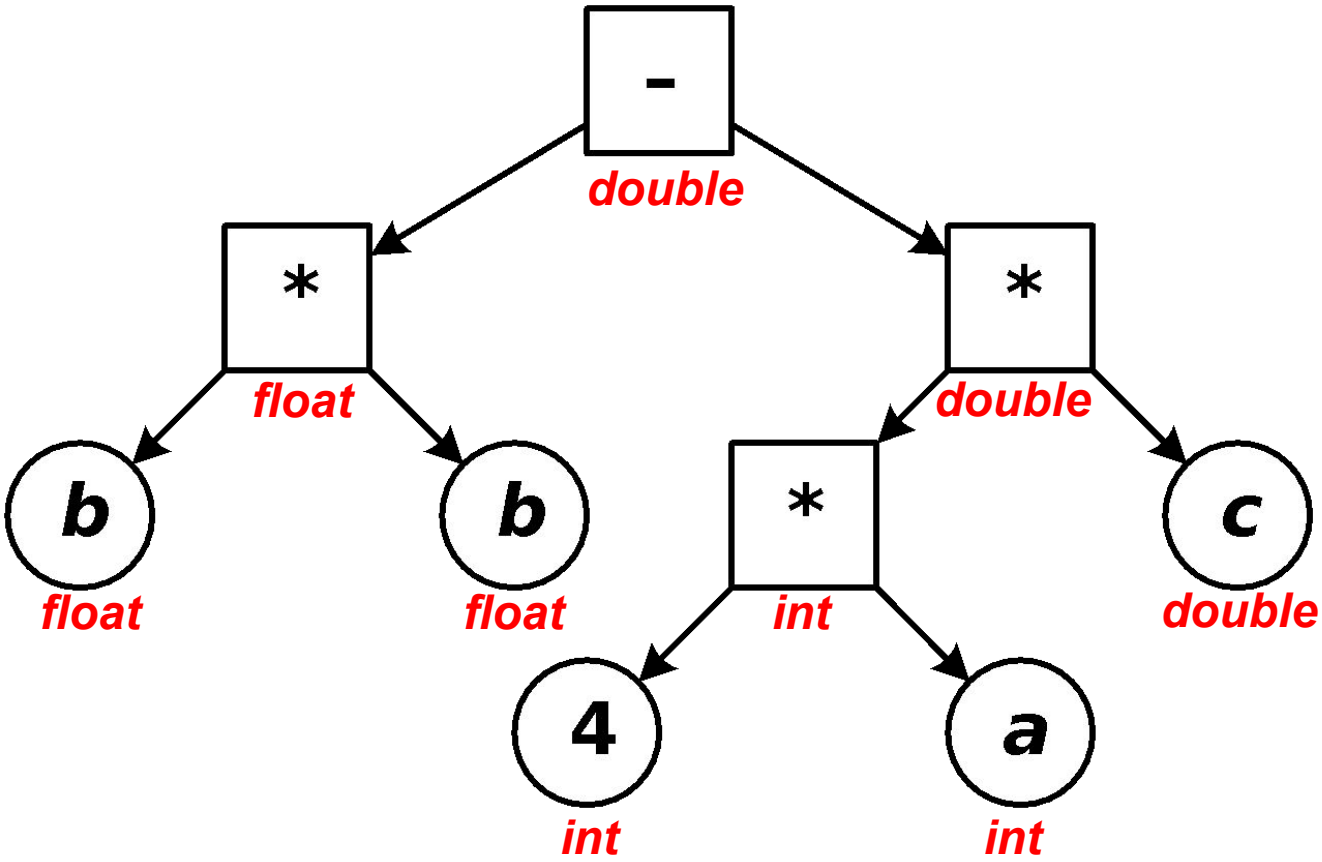


3. Для внутренних вершин, соответствующих арифметическим операциям, тип определяется в соответствии с правилами языка программирования для каждой конкретной операции отдельно.

Если комбинация типов операндов для анализируемой операции является недопустимой, то семантический анализатор должен завершить работу с ошибкой.



Семантический анализ



Фаза оптимизации

Фаза оптимизации

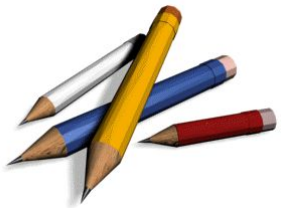
Необязательной фазой трансляции перед работой генератора кода может быть *фаза оптимизации*. На этой фазе делается попытка преобразовать промежуточный код в такой вид, по которому может быть получен более эффективный целевой код.

Основные *критерии оптимизации*:

– используемые ресурсы (память);

– быстродействие;

– объем кода.



Фаза оптимизации

Математически проблема генерации оптимального кода является неразрешимой.

На практике используются *эвристические технологии*, дающие хороший, но не обязательно оптимальный код.