

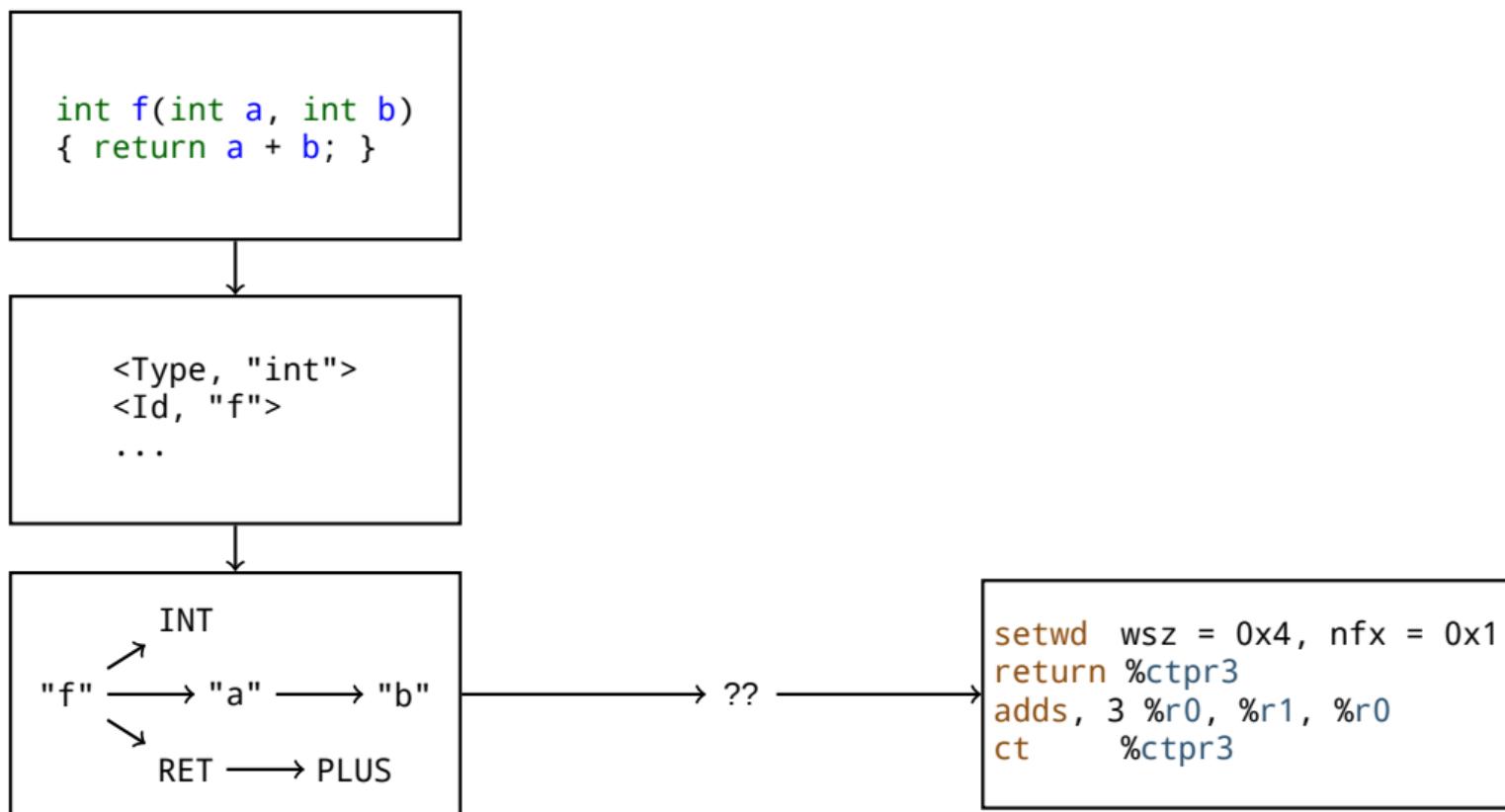
Проектирование компиляторов

Лекция 4.
Семантический анализ.

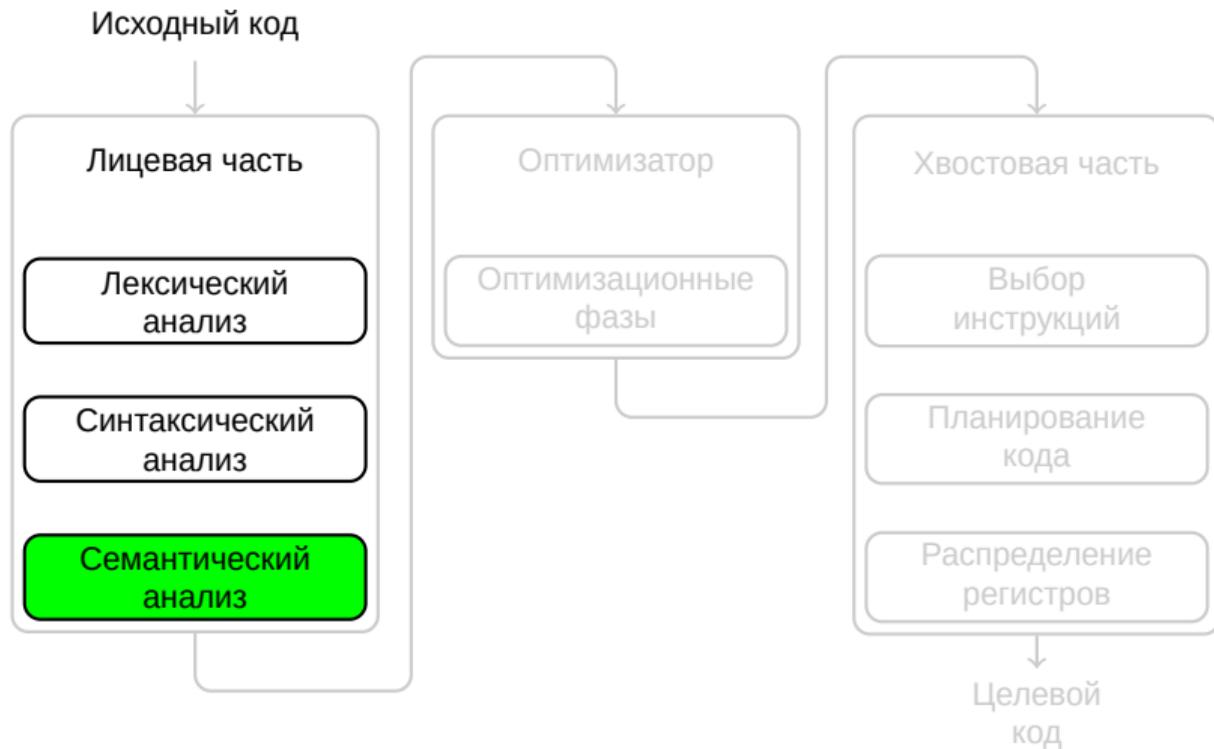
Маркин А. Л.
master@lab-pp.ru

2023

Компиляция программы



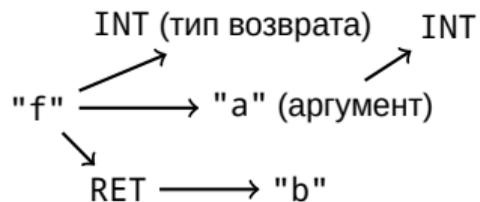
Компиляция программы



Проблема

Что не так с этим кодом?

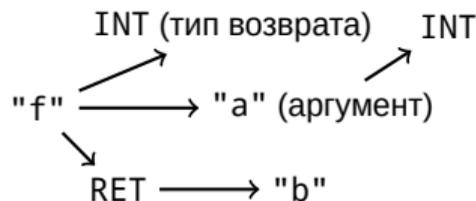
```
int f(int * a)
{
    return b;
}
```



Проблема

Что не так с этим кодом?

```
int f(int * a)
{
    return b;
}
```



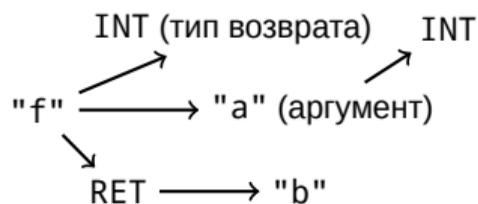
С точки зрения синтаксиса программа правильна, однако с точки зрения стандарта Си она содержит ошибку.

Контекстно-свободные грамматики не умеют отвергать программы, содержащие «смысловые» ошибки.

Проблема

Что не так с этим кодом?

```
int f(int * a)
{
    return b;
}
```



Для выявления обращений к несуществующим идентификаторам следует запоминать все **объявленные** идентификаторы, а в узлах обращения к ним следует проверять наличие этих идентификаторов.

Таблица символов

```
int f(int * a)
{
    return b;
}
```

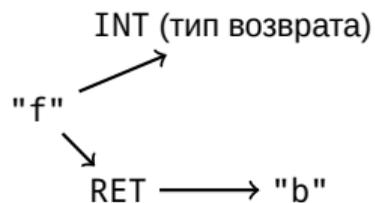


Таблица символов
"a"

Одним из способов проверки существования объектов является поиск по **таблице символов**.

Таблица символов

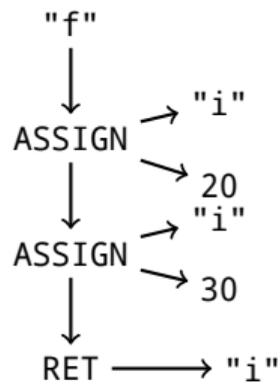
Таблица символов связывает идентификатор с его типом и местоположением. Работа с таблицей символов производится по следующему принципу:

- ▶ При обработке узла дерева разбора с объявлением переменной, идентификатор заносится в таблицу. Сам узел перестаёт быть нужным и может быть удалён.
- ▶ При обработке узла, содержащего идентификатор производится поиск этого идентификатора, узел начинает ссылаться на найденную запись.
- ▶ Если запись не найдена, то это означает использование объекта без его определения, что считается ошибкой.

Проблема

Что не так с этим кодом?

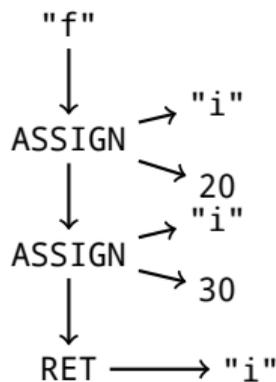
```
int f(int i)
{
  {
    int i;
    i = 20;
  }
  {
    int i;
    i = 30;
    return i;
  }
}
```



Проблема

Что не так с этим кодом?

```
int f(int i)
{
  {
    int i;
    i = 20;
  }
  {
    int i;
    i = 30;
    return i;
  }
}
```



"f"
"i"
??? "i"???
??? "i"???

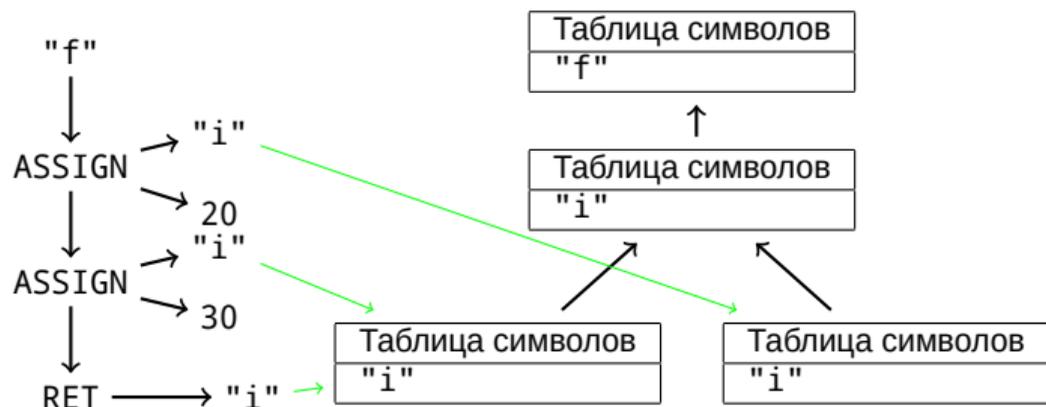
С точки зрения языка Си программа правильная. Но в таблице не может находиться несколько записей с одним ключом.

Область видимости

Область видимости (scope) — область программы, в которой идентификатор связан с конкретной записью в таблице символов.

Каждая область видимости будет обладать собственной таблицей символов.

```
int f(int i)
{
  {
    int i;
    i = 20;
  }
  {
    int i;
    i = 30;
    return i;
  }
}
```



Область видимости

Обычно таблица символов представляет из себя структуру «вывернутого дерева», т.к. дуги в ней идут от потомков к родителям.

Каждый узел таблицы символов привязан к области видимости программы. Для определения соответствия идентификатора и записи в таблице применяется следующий алгоритм:

1. Ищем идентификатор в текущей таблице символов.
2. Если идентификатор найден — привязываем запись таблицы к узлу дерева разбора.
3. Если идентификатора в таблице нет — переходим к родительской таблице и повторяем п. 1.
4. Если родительская таблица отсутствует, то объекта с таким идентификатором в данном участке программы не существует.

Область видимости

Области видимости можно поделить на:

- ▶ **Статическая** (lexical scope) — идентификатор привязывается к записи на основе кода программы. Используется почти всеми языками.
- ▶ **Динамическая** (dynamic scope) — идентификатор привязывается к записи на основе контекста исполнения. Характерно для языка `bash`, некоторых диалектов `lisp`.

Основное отличие динамической области видимости заключается в том что на этапе компиляции (а обычно такие языки являются интерпретируемыми) невозможно сказать с какой записью будет ассоциирован данный идентификатор.

Область видимости

Возможные привязки идентификаторов:

- ▶ Выражение
- ▶ Лексический блок
- ▶ Функция
- ▶ Файл
- ▶ Модуль
- ▶ Глобальная

Область видимости

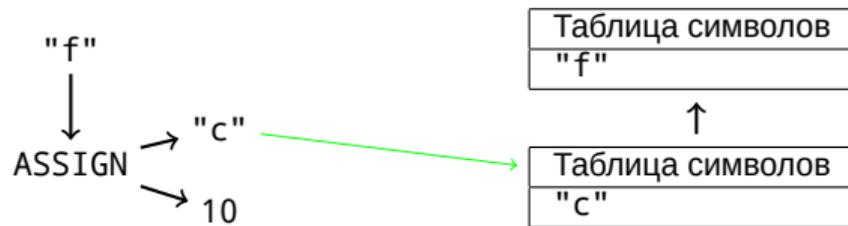
Разные языки программирования обладают разными правилами привязки идентификаторов и записей:

- ▶ **Fortran**: обладает глобальным пространством имён и локальным для каждой функции. Локальные объявления имеют приоритет над глобальными.
- ▶ **C**: Обладает глобальным пространством и локальным для каждой функции. Локальные объявления имеют приоритет над глобальными. При этом функции имеют структуру вложенных лексических блоков, которые являются границей области видимости.
- ▶ **Scheme**: Практически все идентификаторы находятся в едином глобальном пространстве. При этом можно объявлять локальные идентификаторы, которые видны только во время исполнения конкретного участка кода.

Проблема

Что не так с этим кодом?

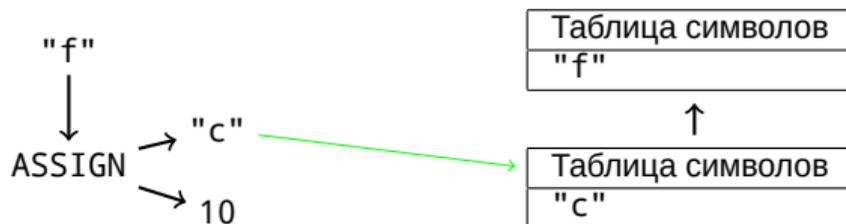
```
void f(char * c)
{
    c = 10;
}
```



Проблема

Что не так с этим кодом?

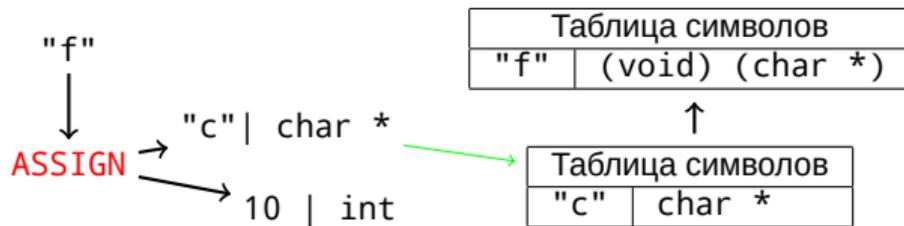
```
void f(char * c)
{
    c = 10;
}
```



Ещё один случай синтаксически правильной программы, содержащей ошибку. Здесь происходит попытка присвоить переменной типа `char *` значение типа `int`.

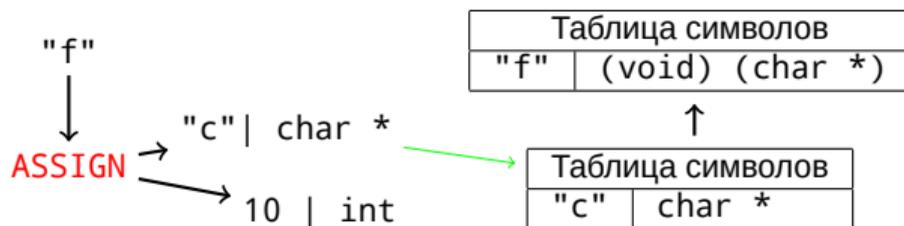
Типы в компиляторе

```
void f(char * c)
{
  c = 10;
}
```



Типы в компиляторе

```
void f(char * c)
{
    c = 10;
}
```



Ещё один случай синтаксически правильной программы, содержащей ошибку. Здесь происходит попытка присвоить переменной типа `char *` значение типа `int`.

Типы в компиляторе

Таблица символов служит не только для определения факта наличия связи идентификатора с записью, но и для хранения информации о записи, в том числе о её типе.

Одной из задач семантического анализа является простановка узлам дерева разбора их типов, также поиск ошибок, связанных с типами.

Дерево разбора с полной информацией о типах и построенной таблицей символов называется **деревом абстрактного синтаксиса**.

Типы в компиляторе

Тип — множество возможных значений и операций над ними.

Система типов языка описывает какие операции допустимы для данных типов.

Проверка типов языка определяет корректность программы с точки зрения системы типов.

Типы в компиляторе

Типы бывают:

- ▶ Базовые (интегральные): простые типы, изначально известные компилятору: `int`, `float`, `bool` и т.п.
- ▶ Пользовательские: программист сам создаёт тип: структуры, объединения.
- ▶ Агрегатные: массивы с элементами заданного типа.

В современные компиляторы могут быть встроены типы, не являющиеся базовыми, например векторные типы, представляющие из себя структуры.

Типы в компиляторе

Операции могут совершаться только с объектами эквивалентных типов. Сама по себе эквивалентность понимается в каждом языке по-разному.

- ▶ Каждый тип эквивалентен сам себе.
- ▶ Типы являются структурно-одинаковыми (например в языке Питон. В языке Си это утверждение неверно).

Часто возникает ситуация когда аргументами выражения являются объекты разных типов: `int a = 1 + 0.3;`. В таких случаях в зависимости от правил языка должно произойти либо приведение одного из аргументов к нужному типу, либо выведено сообщение об ошибке.

Типы в компиляторе

Способы проверки типов:

- ▶ **Статические** — почти все проверки типов выполняются во время компиляции (C, C++, Java).
- ▶ **Динамические** — почти все проверки типов выполняются во время исполнения (Python, Scheme).

Также бывают языки без проверки типов, например ассемблер и машинный код. Тем не менее, разработчики аппаратуры стремятся устранить это упущение, например: «защищённый режим» в процессорах Эльбрус, MTE — расширение системы команд ARM, и другие.

Типы в компиляторе

- ▶ Статические проверки типов:
 - ▶ Позволяют выявлять ошибки во время компиляции
 - ▶ Позволяют не тратить время на проверку типов во время исполнения
 - ▶ Позволяют проводить некоторые оптимизации кода

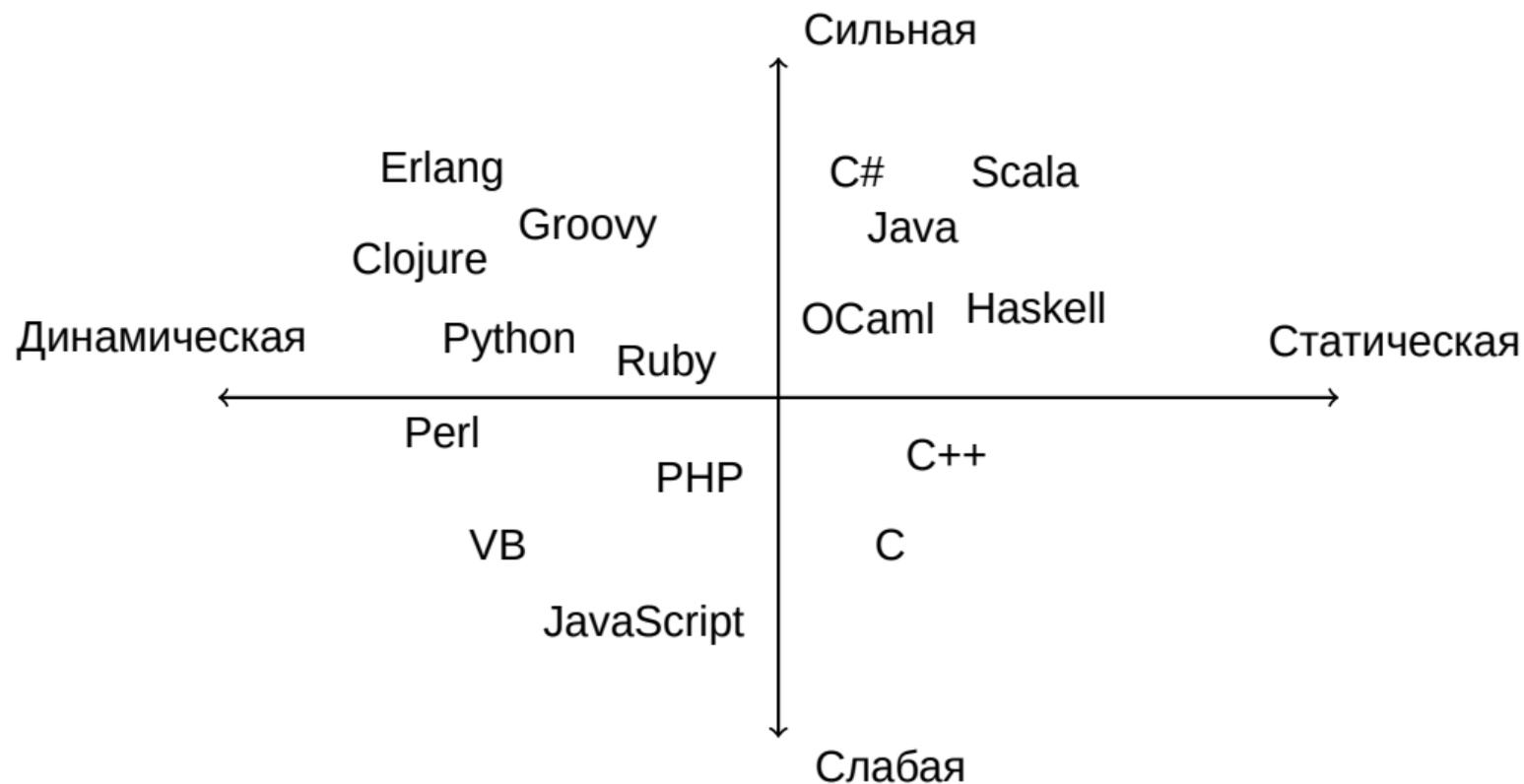
- ▶ Динамические проверки типов:
 - ▶ Упрощают и ускоряют прототипирование на данном языке

Типы в компиляторе

Ещё одной характеристикой системы типов в компиляторе является строгость её проверок:

- ▶ **Сильная** система типов позволяет производить операции только для объектов эквивалентных типов. Попытки произвести операции над не эквивалентными типами приводят к ошибке.
- ▶ **Слабая** система типов позволяет неявно для программиста строить операции приведения объекта одного типа к другому.

Система типов в языках



Ошибки проектирования языков

В языке Си существует конструкция, которую невозможно правильно распознать без семантической информации:

(A)*B

Ошибки проектирования языков

В языке Си существует конструкция, которую невозможно правильно распознать без семантической информации:

$(A)*B$

Данная конструкция может быть:

- ▶ Умножением двух переменных: $A * B$
- ▶ Приведением к типу A разыменованной переменной B : $(A)(*B)$

Одним из способов решения проблемы (классическим) является заполнение и использование таблицы символов на этапе лексического анализа. Такой способ называется **lexer hack**.

Ошибки проектирования языков

Крупной ошибкой проектирования является язык JavaScript:

```
> [] + []           > true == 1
< ""                < true

> [] + {}           > true === 1
< "[object object]" < false

> {} + []           > (!+[]+[]+![]).length
< 0                 < 9

> true+true+true === 3 - true > 9 + "1"
< true                < "91"

> true - true       > 91 - "1"
< 0                  < 90
```

Заключение

Работа семантического анализатора состоит из двух частей:

1. Построение дерева абстрактного синтаксиса:
 - ▶ Заполнение таблицы символов
 - ▶ Дополнение информацией узлов дерева разбора, устранение лишних узлов
2. Выявление смысловых ошибок в программе:
 - ▶ Ошибки объявления идентификаторов
 - ▶ Неправильное с точки зрения типов взаимодействие объектов узлов
 - ▶ Неправильное количество аргументов в вызываемых функциях и т.п.
 - ▶ Прочие специфичные для конкретного языка ошибки

Заключение

Обычно работа лексического, синтаксического и семантического анализаторов не имеет такой чёткой границы как в описывается теории. Это может быть связано как с архитектурой языков, так и с соображениями времени компиляции.

Дерево абстрактного синтаксиса хорошо подходит для описания программы приближённого к языку программирования, но использовать его для дальнейшей компиляции не так удобно.

Для продолжения работы с компилятором, лицевая часть должна перевести дерево абстрактного синтаксиса в промежуточное представление этого компилятора.

Литература

- ▶ *A. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман* Компиляторы: принципы, технологии и инструменты.
- ▶ *K. D. Cooper, L. Torczon* Engineering a compiler.
- ▶ *A. Aiken* CS143 Compilers. Stanford lectures.
- ▶ *Р. Хантер* Основные концепции компиляторов