Семантический анализ в трансляторах

Лекция 3: От структуры к смыслу

Курс Б1.В.12

ИГУ, Кафедра информационных технологий

2025

Синтаксис vs Семантика

Синтаксис

Как написано?

- Правила построения
- Грамматика
- Структура
- ”Форма”

Семантика

Что означает?

- Смысл конструкций
- Типы данных
- Области видимости
- ”Содержание”

Примеры

- 2 + 3 * 4 ☒
- 2 + * 3 ☒
- if x then y ☒

Примеры

- "2" + 3 ☒
- x = y (если y не объявлена) ☒
- 1 / 0 ☒

Архитектура семантического анализа

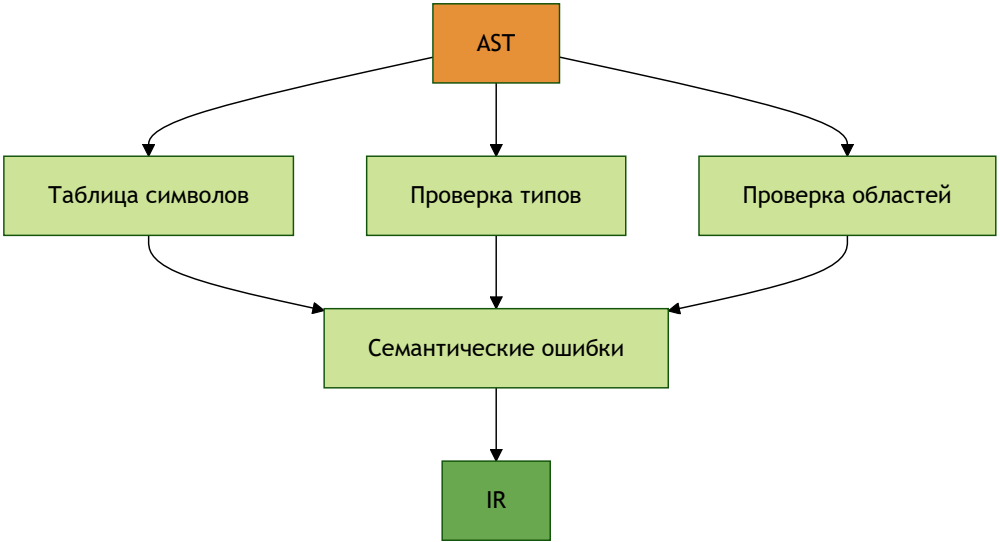


Таблица символов: ”паспортный стол” программы

Что хранится?

- Имена переменных
- Типы данных
- Области видимости
- Дополнительная информация

Педагогическая аналогия

- **Переменные** - как имена учеников
- **Типы** - как предметы (математика, физика)
- **Области** - как классы (5А, 6Б)

Пример

Имя	Тип	Область
x	int	main
y	string	main
pi	float	global

Области видимости: ”этажи” программы

Пример программы

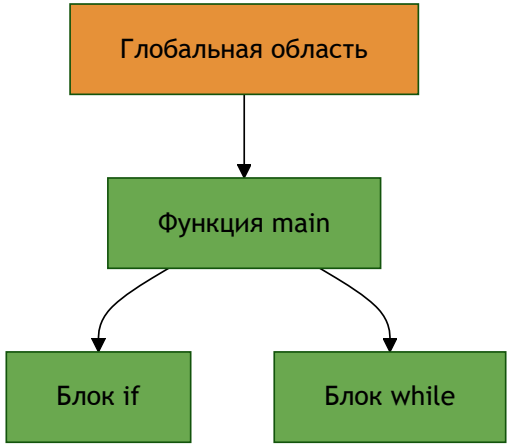
```
x = 10          # Глобальная область

def func():
    y = 20      # Локальная область
    print(x)    # Видит глобальную
    print(y)    # Видит локальную

print(z)        # Ошибка! z не видна
```

Иерархия областей

- **Глобальная** - видна везде
- **Локальная** - видна только внутри функции
- **Блочная** - видна только в блоке (if, for)



Типы данных: ”специализация” значений

Базовые типы

- **Целые** (int): 1, -5, 100
- **Вещественные** (float): 3.14, -2.5
- **Строки** (string): ”текст”, ’a’
- **Логические** (bool): true, false

Педагогическая аналогия

- **Типы** - как единицы измерения
- **5 метров** + **3 кг** = ошибка!
- **2 яблока** × **3** = 6 яблок
- **10 см** + **20 см** = 30 см

Зачем нужны типы?

- Предотвращение ошибок
- Оптимизация вычислений
- Ясность кода
- Автодокументирование

Проверка типов: ”совместимость” операций

Корректные операции

- 5 + 3 ⟶ 8 (int + int)
- "a" + "b" ⟶ "ab" (string + string)
- 3.14 * 2 ⟶ 6.28 (float * int)
- true and false ⟶ false (bool and bool)

Некорректные операции

- 5 + "текст" ⟶ Ошибка!
- true * 3 ⟶ Ошибка!
- "a" - "b" ⟶ Ошибка!
- 3.14 and 5 ⟶ Ошибка!

Правила совместимости

- Операнды должны иметь совместимые типы
- Результат операции имеет определенный тип
- Возможно автоматическое приведение типов

Семантические ошибки в учебных задачах

Математика

- площадь = длина + ширина ☒
- объем = сторона^3 ☒
- скорость = путь × время ☒

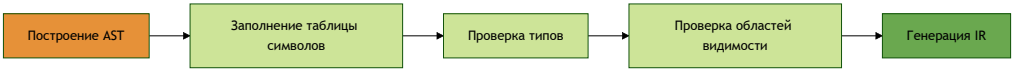
Информатика

- имя = 12345 ☒
- возраст = "пять" ☒
- счетчик = 0 ☒

Общие ошибки

- **Необъявленная переменная**: z = x + y
- **Несовместимые типы**: result = "ответ: " + 5
- **Повторное объявление**: x = 1; x = "текст"
- **Выход за границы**: arr[10] (при размере 5)

Процесс семантического анализа



Промежуточное представление (IR): ”универсальный язык”

Зачем нужно IR?

- **Абстракция** от конкретного языка
- **Оптимизация** независимо от платформы
- **Переносимость** между системами
- **Упрощение** генерации кода

Преимущества IR

- Единое представление для разных языков
- Легко анализировать и оптимизировать
- Упрощает поддержку новых платформ
- Позволяет делать кросс-компиляцию

Аналогия для педагогов

- **Исходный код** - как задача на русском
- **IR** - как математическая формула
- **Машинный код** - как решение на калькуляторе

Примеры промежуточных представлений

AST (Abstract Syntax Tree)

- Дерево разбора
- Сохраняет структуру
- Удобно для анализа

Трехадресный код

- t1 = 2 * 3
- t2 = x + t1
- y = t2

Простота и однозначность

LLVM IR

```
define i32 @main() {
  %x = alloca i32
  store i32 5, i32* %x
  %y = load i32, i32* %x
  %result = add i32 %y, 3
  ret i32 %result
}
```

Промышленный стандарт

От AST к трехадресному коду

Исходное выражение

y = x + 2 * 3

AST

- Присваивание
  - Лево: y
  - Право: +
    - Лево: x
    - Право: *
    - - Лево: 2
    - - Право: 3

Трехадресный код

```
t1 = 2 * 3     # Умножение
t2 = x + t1    # Сложение
y = t2         # Присваивание
```

Преимущества

- Линейная структура
- Простота генерации
- Легкость оптимизации

LLVM IR: промышленное промежуточное представление

Особенности LLVM IR

- Статически типизированный
- В виде SSA (Static Single Assignment)
- Независимый от платформы
- Поддерживает оптимизации

Педагогическое значение

- Показывает ”универсальный язык” компиляторов
- Демонстрирует принципы оптимизации
- Объясняет этапы трансляции

Пример IR

```
define i32 @calculate(i32 %a, i32 %b) {
  %1 = add i32 %a, %b
  %2 = mul i32 %1, 2
  ret i32 %2
}
```

Использование в курсе

- Генерация простого IR
- Визуализация процесса
- Понимание архитектуры

Оптимизации на уровне IR

Постоянная свертка

```
# До оптимизации
t1 = 2 * 3
t2 = x + t1

# После оптимизации
t2 = x + 6
```

Удаление мертвого кода

```
# До оптимизации
x = 5
y = 10
z = x + y  # Не используется

# После оптимизации
x = 5
y = 10
```

Распространение констант

```
# До оптимизации
x = 5
y = x + 3

# После оптимизации
x = 5
y = 8
```

Педагогический аспект

- Аналогично упрощению выражений в математике
- Показывает ”умные” преобразования
- Демонстрирует эффективность компиляторов

Практический пример: калькулятор выражений

Входные данные

```
# Выражение
expr = "2 * (3 + x)"

# Переменные
variables = {"x": 5}
```

Этапы обработки

1. Лексический анализ
2. Синтаксический анализ ⟶ AST
3. Семантический анализ
4. Генерация IR
5. Вычисление

Семантические проверки

- Проверка типов переменных
- Проверка областей видимости
- Проверка корректности операций
- Проверка инициализации

Результат

2 * (3 + 5) = 16

Пример: система проверки математических задач

Задача ученика

”Вычислить площадь прямоугольника со сторонами 5 и 3”

Решение ученика

```
длина = 5
ширина = "3"  # Ошибка типа!
площадь = длина * ширина
```

Семантический анализ

- **длина**: int ☒
- **ширина**: string ☒
- **умножение**: int × string ☒
- **Ошибка**: Несовместимые типы

Правильное решение

```
длина = 5
ширина = 3
площадь = длина * ширина  # 15
```

Инструменты для семантического анализа

ANTLR4 + Listeners/Visitors

- Автоматическое построение AST
- Обход дерева разбора
- Сбор информации о символах
- Проверка семантических правил

Собственные реализации

- Таблицы символов на хеш-таблицах
- Система типов на классах
- Проверки на условиях

Готовые

- **LLVM** - для серьезных проектов
- **Rust** - со встроенной системой типов
- **TypeScript** - для веб-разработки

Рекомендация для курса

- ANTLR4 для прототипирования
- Python для быстрой разработки
- Простые структуры данных

Структуры данных в семантическом анализе

Зачем нужны структуры данных?

- **Эффективное хранение** информации
- **Быстрый поиск** символов и типов
- **Управление сложностью** анализа
- **Масштабируемость** проектов

Педагогическая аналогия

- **Хеш-таблица** - как классный журнал
- **Дерево** - как структура школы
- **Граф** - как связи между предметами
- **Стек** - как стопка тетрадей

Основные структуры

- Хеш-таблицы для символов
- Деревья для областей видимости
- Графы для системы типов
- Стеки для анализа выражений

Хеш-таблицы: быстрый поиск символов

Принцип работы

- **Ключ**: имя переменной
- **Значение**: информация о переменной
- **Хеш-функция**: преобразует имя в индекс
- **Разрешение коллизий**: цепочки или открытая адресация

Пример в Python

```
symbol_table = {}
symbol_table["x"] = {"type": "int", "value": 5}
symbol_table["y"] = {"type": "string", "value": "text"}
```

Преимущества

- **Быстрый поиск**: O(1) в среднем случае
- **Простота реализации**
- **Гибкость** хранения

Применение в трансляторах

- Таблицы символов
- Кэширование вычислений
- Хранение констант

Представление типов данных

Базовые типы

- **Примитивные**: int, float, bool, string
- **Составные**: массивы, структуры
- **Специальные**: void, null, any

Структура типа

```
class Type:
    def __init__(self, name, size=0):
        self.name = name      # "int", "float"
        self.size = size      # размер в байтах
        self.fields = {}      # для структур
```

Иерархия типов

- **Числовые**: int ⟶ float
- **Строковые**: char ⟶ string
- **Логические**: bool
- **Пользовательские**: struct, enum

Проверка совместимости

- **Прямое совпадение**: int ⟶ int
- **Приведение**: int ⟶ float
- **Несовместимость**: int ⟶ string

Деревья областей видимости

Иерархическая структура

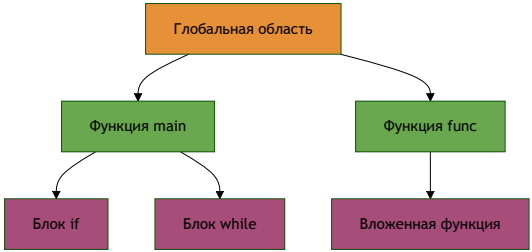
- **Корень**: глобальная область
- **Узлы**: функции, блоки
- **Листья**: локальные переменные
- **Связи**: родитель-потомок

Поиск символов

- Начинаем с текущей области
- Поднимаемся к родителям
- Останавливаемся при нахождении
- Ошибка если не найдено

Пример программы

```
x = 1          # Глобальная
def func():
    y = 2      # Локальная в func
    if True:
        z = 3  # Локальная в if
```



Графы для системы типов

Отношения между типами

- **Наследование**: классы
- **Совместимость**: приведение
- **Эквивалентность**: псевдонимы
- **Состав**: агрегация

Применение

- Проверка совместимости типов
- Автоматическое приведение
- Разрешение перегрузки
- Вывод типов

Пример графа типов

```
number
 |-- int
 |-- float
 +-- complex
object
 |-- string
 +-- array
```

Реализация

```
type_graph = {
   "int": ["float", "complex"],
   "float": ["complex"],
   "string": ["object"]
}
```

Стеки для анализа выражений

Принцип LIFO

Last-In-First-Out

- **Push**: добавление элемента
- **Pop**: извлечение элемента
- **Peek**: просмотр верхнего

Применение

- Проверка типов в выражениях
- Анализ вложенных вызовов
- Управление областью видимости
- Вычисление констант

Анализ выражений

Выражение: 2 + 3 * 4
Стек типов: [int, int, int]
Стек операций: [*, +]

Реализация в Python

```
type_stack = []
type_stack.append("int")    # push
top_type = type_stack[-1]  # peek
last_type = type_stack.pop() # pop
```

Реализация таблицы символов

Структура символа

```
class Symbol:
    def __init__(self, name, type, scope):
        self.name = name      # имя переменной
        self.type = type      # тип данных
        self.scope = scope    # область видимости
        self.value = None     # значение (опционально)
        self.line = 0         # строка объявления
```

Таблица символов

```
class SymbolTable:
    def __init__(self):
        self.symbols = {}     # хеш-таблица
        self.scopes = []      # стек областей
```

Методы таблицы

```
def add_symbol(self, symbol):
    key = f"{symbol.scope}.{symbol.name}"
    self.symbols[key] = symbol

def find_symbol(self, name, scope):
    # Поиск в текущей и родительских областях
    while scope:
        key = f"{scope}.{name}"
        if key in self.symbols:
            return self.symbols[key]
        scope = self.get_parent_scope(scope)
    return None
```

Система типов: представление и проверка

Базовые типы

```
class TypeSystem:
    def __init__(self):
        self.types = {
            "int": BaseType("int", 4),
            "float": BaseType("float", 8),
            "string": BaseType("string", 16),
            "bool": BaseType("bool", 1)
        }
        self.compatibility = {
            "int": ["float"],    # int → float
            "float": []          # float → ?
        }
```

Проверка совместимости

```
def is_compatible(self, from_type, to_type):
    if from_type == to_type:
        return True
    if from_type in self.compatibility:
        return to_type in self.compatibility[from_type]
    return False
```

Пример использования

```
# int → float: разрешено
system.is_compatible("int", "float")  # True
# float → int: запрещено
system.is_compatible("float", "int")  # False
```

Оптимизации структур данных

Кэширование поиска

- **Запоминание** результатов поиска
- **Инвалидация** при изменении
- **Экономия** времени вычислений

Пулы строк

- **Интернирование** строк
- **Сравнение** по ссылкам
- **Экономия** памяти

Ленивые вычисления

```
class CachedSymbolTable:
    def __init__(self):
        self.symbols = {}
        self.cache = {}  # кэш поиска

    def find_cached(self, name, scope):
        key = f"{scope}.{name}"
        if key not in self.cache:
            self.cache[key] = self.find_symbol(name, scope)
        return self.cache[key]
```

Блочные аллокаторы

- **Групповое выделение** памяти
- **Уменьшение фрагментации**
- **Быстрое освобождение**

Практическое задание: реализация семантического анализатора

Задача

Реализовать систему типов и таблицу символов для простого языка

Требования

- Поддержка базовых типов
- Проверка совместимости
- Области видимости
- Сообщения об ошибках

Структуры данных

- Хеш-таблица для символов
- Дерево областей видимости
- Граф совместимости типов
- Стек для анализа выражений

Пример теста

```
x = 5          # int
y = "text"     # string
z = x + y      # Ошибка: несовместимые типы
```

Лабораторная 3: Семантический анализатор

Цель

Добавить семантический анализ к парсеру из Лаб 2

Задачи

1. Реализовать таблицу символов
2. Добавить проверку типов
3. Обрабатывать семантические ошибки
4. Генерировать промежуточное представление

Требования

- Поддержка базовых типов
- Проверка областей видимости
- Сообщения об ошибках
- Генерация простого IR

Пример вывода

```
Ошибка: переменная 'y' не объявлена
Ошибка: несовместимые типы в операции '+'
```

Критерии оценки лабораторной 3

Таблица символов (30%)

- Корректное хранение информации
- Поддержка областей видимости
- Эффективный поиск
- Обработка конфликтов

Генерация IR (25%)

- Корректность представления
- Простота и ясность
- Соответствие исходному коду
- Возможности оптимизации

Проверка типов (30%)

- Обнаружение несовместимостей
- Поддержка базовых типов
- Правила приведения
- Сообщения об ошибках

Качество кода (15%)

- Структура проекта
- Тестирование
- Документация
- Читаемость

Педагогические применения семантического анализа

Для учителей математики

- Проверка типов в выражениях
- Валидация единиц измерения
- Контроль размерностей
- Проверка формул

Для учителей информатики

- Обучение системе типов
- Понимание областей видимости
- Отладка программ
- Анализ ошибок

Пример

- площадь = 5 м × 3 м = 15 м² ☒
- скорость = 10 м / 2 с = 5 м/с ☒
- масса = 5 м + 3 кг ☒

Образовательная ценность

- Развитие логического мышления
- Понимание абстракций
- Навыки анализа и проверки
- Подготовка к программированию

Что дальше? Генерация кода

Лекция 4

Генерация кода и оптимизации

- От IR к машинному коду
- Распределение регистров
- Оптимизации времени выполнения
- Генерация исполняемых файлов

Лабораторная 4

Генератор учебных материалов

- Генерация вариантов задач
- Создание тестов
- Экспорт в разные форматы

Итоговый проект

Сквозной транслятор для учебных задач

- Ввод: условие задачи
- Обработка: анализ и проверка
- Вывод: решение + объяснения

Практическое применение

Готовый инструмент для использования в школе

Ключевые выводы

Семантический анализ

- Проверяет смысловую корректность
- Работает с таблицей символов
- Контролирует типы данных
- Обнаруживает скрытые ошибки

Педагогическая ценность

- Понимание системы типов
- Навыки анализа и проверки
- Развитие абстрактного мышления
- Подготовка к программированию

Промежуточное представление

- Универсальный язык трансляции
- Основа для оптимизаций
- Абстракция от платформы
- Упрощает генерацию кода

Практический результат

Студенты создают работающие системы проверки для школьных задач

Вопросы и обсуждение

Для размышления

- Какие семантические ошибки чаще всего делают ваши ученики?
- Как система типов помогает в обучении математике?
- Что сложнее: синтаксический или семантический анализ?

Практическое задание

Придумайте 3 примера семантических ошибок в вашем предмете и объясните, как их обнаружить

Следующие шаги

- Изучить документацию ANTLR4
- Начать работу над Лаб 3
- Подготовить вопросы по семантическому анализу