Теория компиляторов

Лекция 1
Обзор предметной области

Соломатин Д.И., ПиИТ ФКН ВГУ,
solomatin@cs.vsu.ru

При подготовке данной лекции использовались материалы

- **Учебный курс «Разработка компиляторов на платформе .NET»**

  © Андрей А.Терехов, Наталья Вояковская, Дмитрий Булычев, Антон Москаль, 2001

Учебная литература

- ToDo

Понятие транслятора

- Основная задача *транслятора* – сделать программу на некотором языке программирования понятной компьютеру

- Виды трансляторов:
  - *Компиляторы*
  - *Интерпретаторы*
  - *Трансляторы (в узком смысле)*

Компилятор

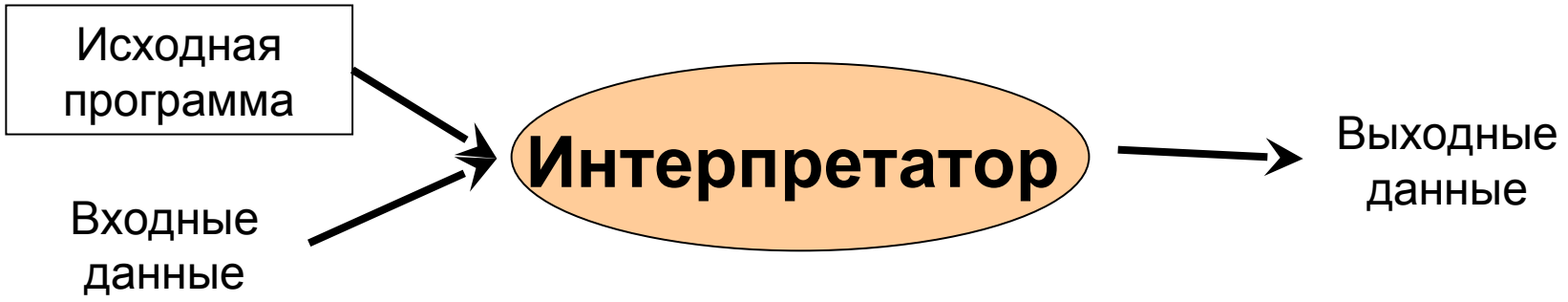
- Компилятор переводит *исходную программу* в эквивалентную ей *целевую программу*.
  - Исходная программа, как правило, написана на *языке высокого уровня*
  - *Целевая программа*, как правило, программа на *низкоуровневом языке*, понимаемом *целевой платформой* (машинный язык)

Формальное определение компилятора

- Пусть
  - L_1 – язык исходных программ
  - L_2 – язык целевых программ
- Тогда компилятор:
  - $K : L_1 \rightarrow L_2$
    – отображение множества исходных программа L_1 в множество целевых программ L_2
- Пусть
  - L_3 – язык, на котором написан компилятор (может быть несколько языков)
- Тогда компилятор:
  - $K_{L_3} : L_1 \rightarrow L_2$
- Для компилятора также можно говорить о языке
  - L_4 – язык, в который скомпилирован сам компилятор
  - Если L_4 != L_2, то компилятор является *кросс-компилятором*

Интерпретатор

- Интерпретатор выполняет программы на входном языке
  - Как правило, интерпретатор в процессе работы все же переводит входную программу в какое-либо промежуточное представление, удобное для выполнения (абстрактное синтаксическое дерево, промежуточный код и т.п.)
  - Более того, интерпретатор может транслировать входную программу в инструкции платформы, на которой выполняется, для увеличения быстродействия (т.е. выполнять внутреннюю компиляцию)



Условность терминологии

- Разница между компилятором и интерпретатором часто весьма условна
  - Яркий пример, интерпретатор языка Python, при первом запуске программы (файлы *.py) компилирует ее во внутреннее представление, которое сохраняет в файловой системе (файлы *.pyc) для ускорения последующих запусков

Транслятор (в узком смысле)

- Переводит одно текстовое представление программы или данных в другое, также непригодное для непосредственного исполнения:
  - Программы с языка C++ в язык C
  - Данные из формата JSON в формат XML
  - и т.п.
- Условность терминологии

К предметной области также относятся

- Дизассемблеры
- Декомпиляторы
  - Программы, скомпилированные под высокоуровневые целевые платформы (например, виртуальные машины Java и .NET) с отладочной информацией без оптимизации довольно успешно поддаются декомпиляции
- Обфускаторы
  - Используются для сокрытия логики программы (защиты) в качестве средства против декомпиляторов и «ручного» анализа

Целевая программа компилятора

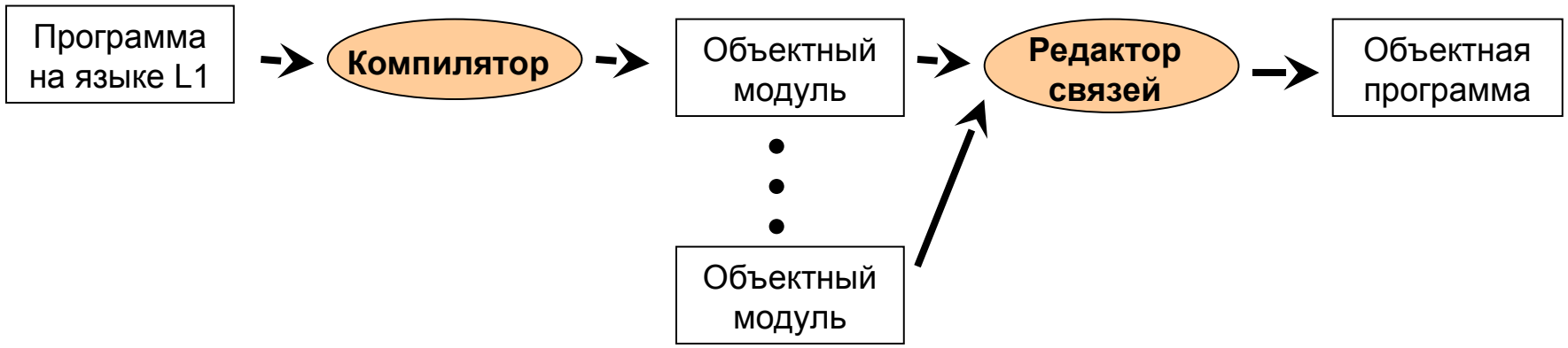
- Последовательность абсолютных команд машинных команд (готовая программа)
- Последовательность перемещаемых машинных команд (объектный модуль)
- Программа на языке ассемблера
- Программа на некотором другом языке

Объектный модуль

- Представляет собой набор машинных команд, адресация которых происходит относительно некоторого подвижного начала (места размещения в памяти)

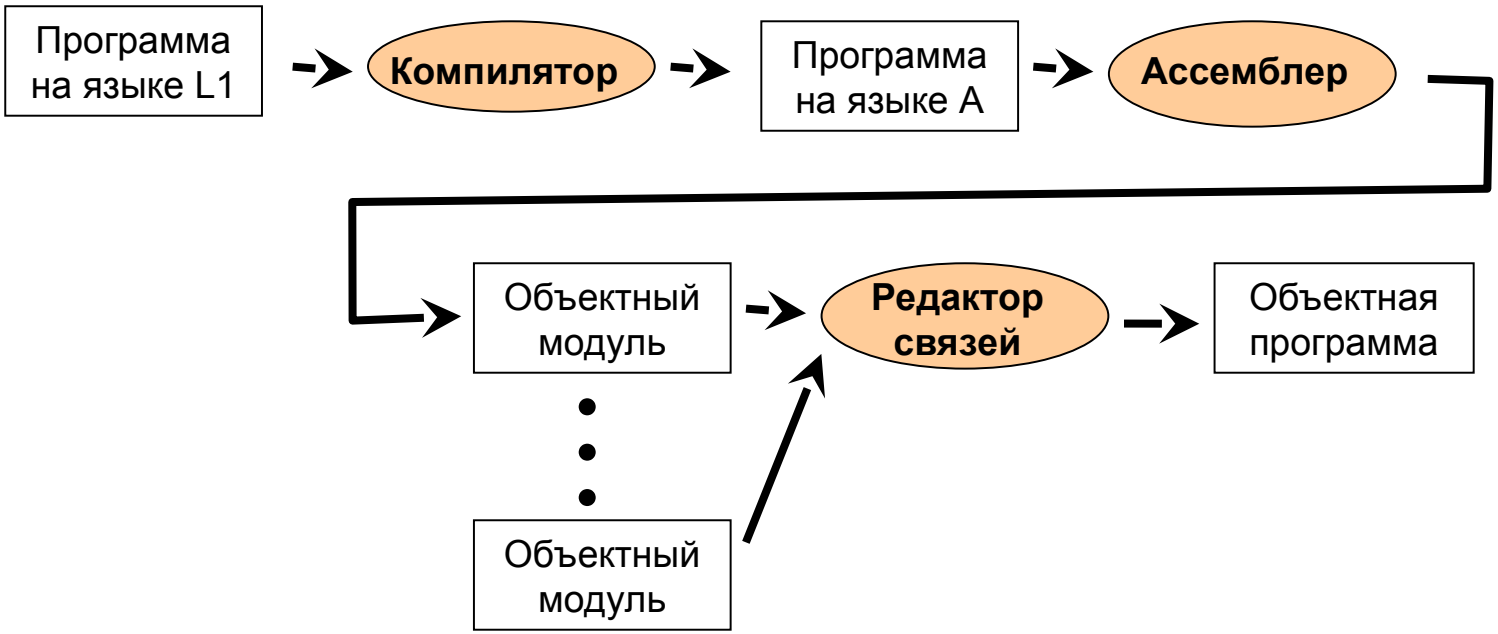
- Содержит метаинформацию об исходной программе (имена и адреса экспортируемых функций и переменных) для возможности последующего связывания с другими объектными модулями в процессе создания результирующей исполняемой программы

Схема построения результирующей исполняемой программы



- Схема иллюстрирует *статическое связывание* объектных модулей (на этапе компиляции код всех необходимых модулей помещается в объектную программу)
- *Динамическое связывание* предполагает загрузку и связывание объектных модулей в момент запуска программы *загрузчиком*
  - При этом размер результирующей программы будет меньше
  - Необходимо наличие библиотек модулей, установленных в системе
  - Исполняемый код загруженных библиотек может разделяться несколькими программами (экономия памяти)

Трансляция в ассемблер



- Трансляция в более высокоуровневый (относительно машинных команд) язык ассемблера упрощает создание компилятора
  - Технически проще (использование меток вместо адресов и т.п.)
  - Легче отлаживать (нагляднее и т.п.)

Методики создания компилятора

- Прямой
  - Разработка на языке ассемблера A компилятора языка L в язык ассемблера A
- *Раскрутка* (*bootstrapping)*
- Использование кросс-трансляторов
- Использование виртуальных машин

Раскрутка (bootstrapping)

- A – язык ассемблера целевой платформы
- Надо получить компилятор K_A: L → A
- Пусть уже есть компилятор K_A: P → A
  (P –язык уровнем выше A, но ниже L)

- Надо написать компилятор K_P: L → A
- Откомпилировав K_P : L → A компилятором K_A: P → A, получим:
  $K_A = K_A(K_P)$: L → A

- Схема применима для написания языка на нем самом
  - При этом язык P является подмножеством языка A
  - Количество шагов раскрутки может быть больше одного, т.е. последовательно получаются компиляторы языков P_1, P_2, …, L, где каждый последующий является расширением предыдущего и компилируется компилятором предыдущего
  - Первый компилятор языка P_1 может быть написан на языке A (ассемблере) или же на самом P_1 и оттранслирован в A вручную

  - Впервые схема была применена в 1960 году при написании компилятора языка Neliac
  - В 1971 году Вирт таким образом реализовал транслятор языка Pascal

Кросс-компилятор

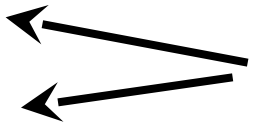
- Компилятор, который работает на одной платформе, а создает код для другой платформы
- Кросс-компиляторы могут создавать код под несколько платформ одновременно

Виртуальная машина

- Компилятор реализуется не под целевую платформу, а под виртуальную (виртуальная машина), которая проще для реализации компилятора
  - Виртом в 70-х годах была разработана виртуальная стековая машина (P-код) для компилятора Pascal-P
- Для каждой целевой платформы разрабатывается интерпретатор
- Как правило, современных виртуальные машины (Java, .NET) гораздо более высокого уровня, чем существующие процессоры

Фазы компиляции

- Анализ
  - Лексический анализ
  - Синтаксический анализ
  - Семантический анализ
    - Видозависимый анализ (также относится к синтезу)



«Парсинг»

- Синтез
  - Оптимизация
  - Генерация кода

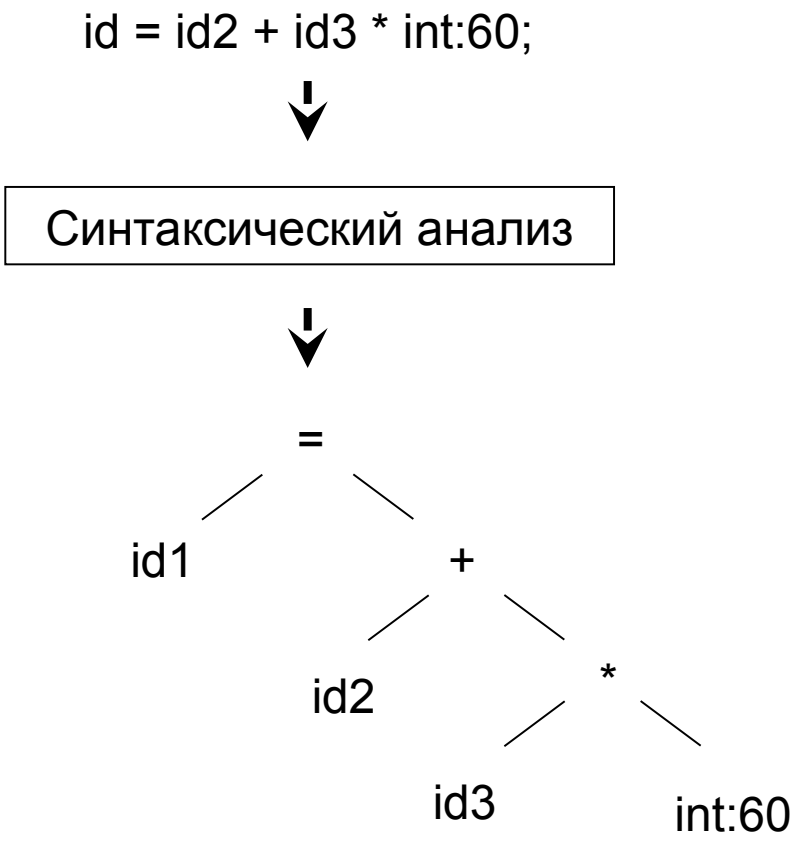
Лексический анализ

position = position + rate * 60;

Лексический анализ

id1 = id2 + id3 * int;

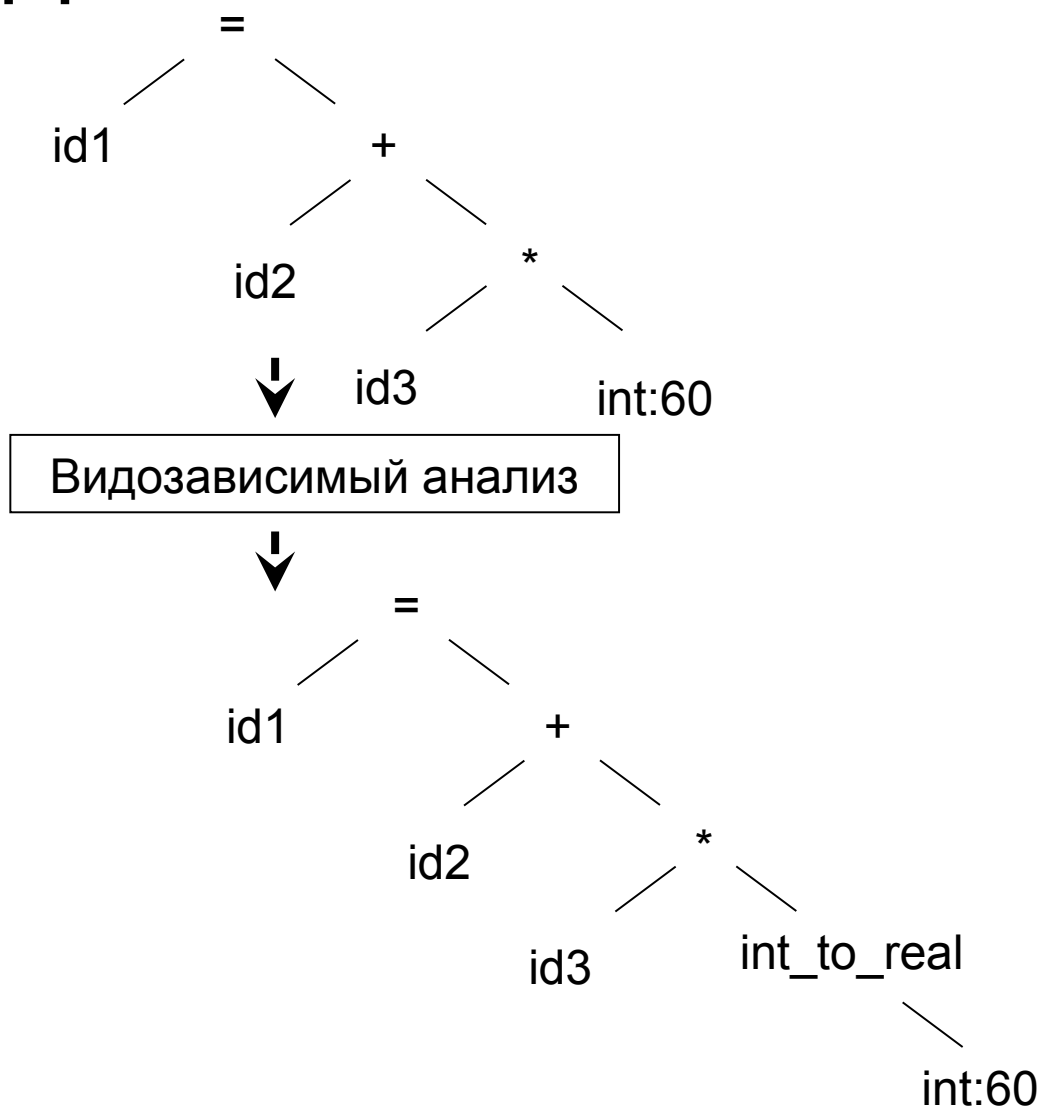
Синтаксический анализ



Семантический анализ

- Анализ наличия ошибок в синтаксически правильной программе:
  - Обращение к необъявленным функциям или переменным
  - Неправильное количество аргументов функций
  - Недопустимые типы данных аргументов функций и операторов
  - И т.п.

Видозависимый анализ



Оптимизация

```
temp1 = int_to_real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Оптимизация

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

Генерация кода

temp1 = id3 * 60.0
id1 = id2 + temp1

Генерация кода

```
ldsfld    id3
ldc.r8    60.
mul
stloc     temp1
ldsfld    id2
ldloc     temp1
add
stsfld    id1
```

Перейдем к практике

Неформальное введение в грамматики
Пример **MathExpr**

```
NUMBER   ->    <число>
group    ->    "("   add   ")"
             | NUMBER
mult     ->    group   ( ( "*" | "/" )   group )*
add      ->    mult    ( ( "+" | "-" )   mult   )*
result   ->    add
```

- Грамматика – описание синтаксиса языка, в виде алфавита языка и правил вывода/разбора выражений языка
- Грамматика – язык описания синтаксиса языков

Пример math_expr.py

- Разбор и вычисление значений правильных математических выражения
- Использован низходящий рекурсивный разбор
- Проект состоит из модулей:
  - MathExprSimpleParser – класс парсера

Позиция разбора

- Указатель, отделяющий разобранную часть входной строки от неразобранной
- Символ в позиции разбора

```
@property
def curr(self) -> str:
    if self.pos < len(self.expr):
        return self.expr[self.pos]
    else:
        return '$'
```

Правило NUMBER

NUMBER -> <число>

```
def NUMBER(self) -> float:
    tmp = ''
    while self.curr.isdigit() or self.curr == '.':
        tmp += self.curr
        self.pos += 1
    return float(tmp)
```

Правило group

```
group    ->    "("   add   ")"
             |  NUMBER
```

```
def group(self) -> float:
    if self.curr == '(':
        self.pos += 1
        res = self.add()
        self.pos += 1
    else:
        res = self.NUMBER()
    return res
```

Правило mult

```
mult    ->   group   ( ( "*" | "/" )   group )*
```

```
def mult(self) -> float:
    res = self.group()
    while self.curr in ('*', '/'):
        op = self.curr
        self.pos += 1
        tmp = self.group()
        res = res * tmp if op == '*' else res / tmp
    return res
```

Правило add

```
add     ->   mult   ( ( "+" | "-" )  mult  )*
```

```
def mult(self) -> float:
    res = self.group()
    while self.curr in ('*', '/'):
        op = self.curr
        self.pos += 1
        tmp = self.group()
        res = res * tmp if op == '*' else res / tmp
    return res
```

Правило result

result -> add

```
def result(self) -> float:
    res = self.add()
    if self.pos != len(self.expr):
        raise Exception(f'Лишний символ {self.curr} в позиции [{self.pos}]')
    return res
```

Пример разбираемой строки

- **Input**

```
expr = '3+2*(5+2-2*(1+1.05))+10*(5+5)'
p = MathExprSimpleParser(expr)
res = p.result()
print(res)
```

- **Output**

```
108.8
```