Титульный лист материалов по дисциплине

ДИСЦИПЛИНА *Теория формальных языков*

ИНСТИТУТ *Информационных технологий*

КАФЕДРА *Вычислительной техники*

ВИД УЧЕБНОГО МАТЕРИАЛА *Лекция*

ПРЕПОДАВАТЕЛЬ *Унгер Антон Юрьевич*

СЕМЕСТР *3 семестр*

5. ПРОЕКТИРОВАНИЕ ТРАНСЛЯТОРА

Трансляция представляет собой перевод исходного текста программы, написанной на некотором языке программирования, во внутреннее представление, понятное машине. В общем случае, к трансляции относят компиляцию, при которой внутренним представлением является последовательность машинных команд, и интерпретацию, при которой программа выполняется на некоторой виртуальной машине инструкция за инструкцией. Мы не будет разделять эти понятия и будем пользоваться одним термином *трансляция*. Транслятор обычно выполняет свою работу в два этапа:

1. Этап *анализа* исходного текста.
2. Этап *синтеза*, в результате которого генерируется машинно-ориентированное представление.

Процесс трансляции схематично показан на рис. 5.

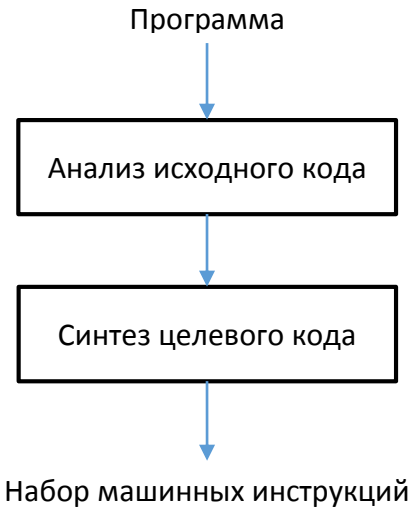

Рисунок 5. Процесс трансляции

Важно отметить, что первый этап трансляции – этап анализа исходного текста – успешно поддается автоматизации. Другими словами, существуют программные средства – лексические и синтаксические анализаторы – которые умеют генерировать компиляторы. Среди подобных систем наибольшую известность имеют *Lex* и *Yacc*, которые называют компиляторами компиляторов [11].

Они компилируют описания сканера и *парсера*, записанные на специальном языке, в программу на языке *C*, которая способна по заданному описанию языка выполнять лексический и синтаксический анализ исходных текстов.

Подход с разделением процесса трансляции на две фазы, хотя и кажется простым на первый взгляд, оказывает большое влияние на структуру транслятора.

Подход позволяет определить *интерфейс* между анализатором и генератором кода, который становится своего рода языком, объединяющим исходный и целевой код. Этот интерфейс, будучи разработанным правильно, позволяет сделать фазу анализа совершенно не зависящей от целевого кода, а фазу синтеза – на зависящей от исходного кода. Это, теоретически, позволит разработчику транслятора не вносить никаких изменений в анализатор исходного кода для добавления поддержки новой архитектуры. Аналогично, если целевая архитектура остается неизменной, а необходимо модифицировать сам язык, этап синтеза затронут не будет.

Транслятор языка программирования представляет собой компьютерную программу, которая сама может быть написана на языке программирования высокого уровня. Хорошо спроектированный компилятор имеет модульную структуру (рис. 6).

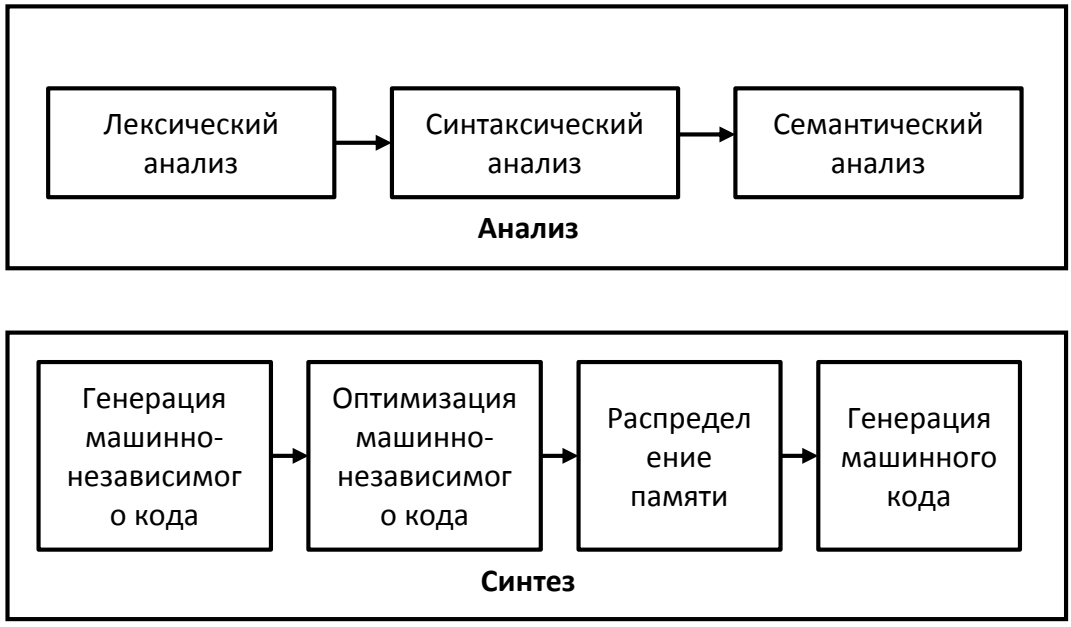

Рисунок 6. Этапы трансляции

Перечислим основные этапы трансляции:

1. Лексический анализ – оперирует с исходным текстом программы.

2. Синтаксический анализ – соединен с выходом лексического анализатора и входом семантического анализатора.

3. Семантический анализ – генерирует абстрактное синтаксическое дерево разбора.

4. Генерация машинно-независимого кода – переводит абстрактное дерево разбора в последовательность инструкция, не привязанную к конкретной архитектуре.

5. Оптимизация машинно-независимого кода – служит для исключения из кода бесполезных инструкций.

6. Распределение памяти – выделяет для каждой переменной в программе область памяти и закрепляет за ней определенный адрес.

7. Генерация машинного кода – выдает последовательность инструкций, пригодную для исполнения на ЭВМ.

Остановимся подробнее на фазе анализа.

Лексический анализ является наиболее простой фазой, в ходе которой вырабатываются так называемые *лексемы* языка или *лексические токены* (от англ. *token* – опознавательный знак). Например, ключевые слова

```
for do while,
```

или пользовательские идентификаторы

```
name sprintf,
```

или знаки операций

```
+ = * - /.
```

Эти цепочки удобнее представлять в виде одного символа, принадлежащего некоторому типу, и имеющего некоторое *ассоциированное значение*.

Задача лексического анализа – перевести исходное текстовое представление из последовательности знаков в последовательность лексем. Эта последовательность символов затем передается на вход синтаксического анализатора.

Рассмотрим пример программы на языке С:

```
sum = 0;
for (i = 0; i <= 99; ++i) {
    sum += a[i]; /* просуммируем массив */
}
```

Данный код на этапе лексического анализа превратится в последовательность токенов вида:

```
(идентификатор sum)
(оператор присвоения =)
(константа 0)
(разделитель ;)
```

```
(ключевое слово for)
(разделитель ()
(идентификатор i)
(оператор присвоения =)
(константа 0)
(разделитель ;)
(идентификатор i)
(оператор <=)
(константа 99)
(разделитель ;)
(оператор ++)
(идентификатор i)
(разделитель ))
(разделитель {)
(идентификатор sum)
(оператор комбинированного присвоения +=)
(идентификатор a)
(разделитель [)
(идентификатор i)
(разделитель ])
(разделитель ;)
(разделитель })
```

Можно видеть, как в процессе лексического разбора исключаются пробельные символы и комментарии.

Важно отметить, что лексический анализатор никак не воспринимает переданную ему на вход последовательность знаков. Например, он не примет никакого решения относительно корректности или некорректности следующей программы на языке С:

```
; number int return do==++.
```

Полная независимость от контекста программы (его синтактической нагрузки) делает лексический анализ относительно простой фазой трансляции. Вместе с тем, как правило лексический анализ занимает продолжительное время, так именно на этом этапе осуществляется посимвольное чтение исходной программы.

Синтаксический анализ (*парсинг*, от англ *parse* – разбирать) производит построение общей структуры программы. Здесь имеет значение, какой символ следует за текущим, а какой ему предшествует. Результат работы

синтаксического анализатора имеет структура дерева – синтаксического дерева разбора.

Например, алгебраическое выражение

```
((1-(2*3))+4),
```

имеет следующую синтаксическую структура (рис. 7).

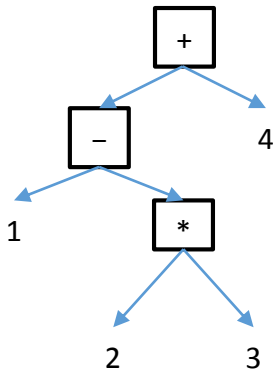

Рисунок 7. Дерево разбора арифметического выражения

Здесь отсутствуют скобки, поскольку сама структура дерева содержит их в неявном виде. Подобным же образом в виде *абстрактного синтаксического дерева* можно представить всю программу. Фаза синтаксического анализа является ключевой фазой анализа исходного текста программы.

Некоторые характеристики реализуемого языка не могут быть проверены в ходе синтаксического анализа. В частности, информация о типах переменных не может быть получена простым сканирование символов исходного кода слева направо. Для этого необходимо иметь возможность перемещаться по тексту программы в границах всего исходного кода.

Семантический анализ необходим для устранения особенностей языка, не поддающихся описанию средствами формальной грамматики. В частности, проверка типов и область видимости переменных происходит на семантической фазе анализа.

На данном этапе синтаксическое дерево разбора уже построено и его допускается модифицировать. Семантические проверки заключаются в обходе дерева разбора и вставке в каждый узел необходимых дополнительных условий.

Строго типизированные языки, в частности, требуют, чтобы с каждым узлом дерева, который отвечает идентификатору, был поставлен в соответствие его тип. К проверке на соответствие значений левой и правой частей выражения можно отнести, например, допускается ли присвоить переменной типа *integer* значение типа *float*?

Поскольку семантический анализ является заключительной фазой анализа, его результатом обычно является некоторое *промежуточное представление*, которое получается «спрямлением» дерева разбора. Результат синтаксического разбора в виде дерева позволяет легко его анализировать: обходить, добавлять и удалять узлы, однако, оно не подходит для хранения и исполнения, например, на некотором виртуальном вычислителе. Будучи по своей природе рекурсивной структурой, дерево на является тем представлением, которое можно эффективно выполнить на виртуальной машине. Выход семантического анализатора, таким образом, зачастую представляет собой последовательность инструкций, которая в функциональном смысле эквивалентна машинным инструкциям для некоей виртуальной машины.

Оптимизация машинно-независимого кода. Промежуточное представление на выходе семантического анализатора может быть оптимизировано. На этом этапе еще нет привязки к конкретной архитектуре, однако код, полученный без машинно-независимой оптимизации, отличается значительно более низким качеством, чем оптимизированный. Данный этап является опциональным, т.е. транслятор будет работать и без него, однако он может оказать сильное влияние на качество целевого кода.

В качестве примера машинно-независимой оптимизации можно привести встраивание (*inlining*) функций, размотка (*unrolling*) циклов, удаление избыточного кода. Получающееся при этом очередное представление кода не имеет ничего общего с представлением, полученным после семантического анализа.

Генерация кода. Данный этап является ключевым для синтеза, поскольку в результате получается набор машинных инструкций, которые машина может понять. На практике, как уже говорилось, на данном этапе необходимо учитывать множество нюансов целевой архитектуры, поэтому он с трудом поддается автоматизации.

В процессе генерации кода находятся ответы на следующие вопросы:

- каков набор инструкций процессора, который необходимо поддерживать?
- как используются регистры общего назначения?
- как выделяется память под переменные?
- как программа взаимодействует с внешними библиотеками и операционной системой?

Оптимизация машинного кода. Данный этап также, как и оптимизация машинно-независимого кода относится к опциональным. Однако, практически разработчики трансляторов под целевую архитектуру придают ему самое большое внимание. Набор поддерживаемых процессорным ядром инструкций (если мы говорим о CISC архитектуре, от англ. *Complex Instruction Set Computing*) может быть очень большим. Создавать для каждой инструкции отдельное грамматическое правило, которое бы поддерживало эту инструкцию, было бы нерационально, поскольку это привело бы к сильному усложнению исходного языка. Кроме того, инструкция может зависеть от контекста, который становится известен только на этапе выполнения.

Говоря об оптимизации, всегда необходимо соблюдать сдержанность. Не существует такого понятия, как «лучший код». Оптимизация служит для того, чтобы сделать код *лучше*. Под словом «лучше» может подразумеваться «быстрее» или «компактнее». Часто, эти понятия являются взаимоисключающими.

Результатом генерации кода является код, который может быть запущен на целевой машине. Это может быть объектный модуль, библиотека или исполняемый файл. В любом случае процессор может выполнить код, который в них содержится.

Распределение памяти занимается выделением для каждой переменной или константы некоторого места для хранения своего значения. При этом память может быть следующего типа:

1. Статическая память, в которой хранятся переменные, время жизни которых равно времени жизни программы.

2. Динамическая память, хранящая переменные некоторой функции, по завершение которой их можно удалить.

3. Глобальная память используется в том случае, если время жизни переменной можно определить только на этапе выполнения программы (*runtime*).

Стоит отметить, что в отличие от анализа, синтез трудно поддается автоматизации.

Несколько слов следует сказать о средствах автоматизированной разработки анализаторов.

1. Генераторах лексических анализаторов.

2. Генераторах синтаксических анализаторов.

На вход генератора лексических анализаторов поступает информация о *лексической структуре* языка (каким образом из знаков составляются

лексемы). Выход представляет собой программу (например, на языке *C*), которая представляет собой лексический анализатор для данного языка.

На вход генератора синтаксических анализаторов подается информация о *синтаксический структуре* языке (используемой им грамматике). Результатом работы является синтаксический анализатор в виде программы (например, на том же языке C).

Вход и выход обеих программ соединяют для получения окончательного результата – дерева синтаксического анализа. Процесс схематически изображен на рис. 8.

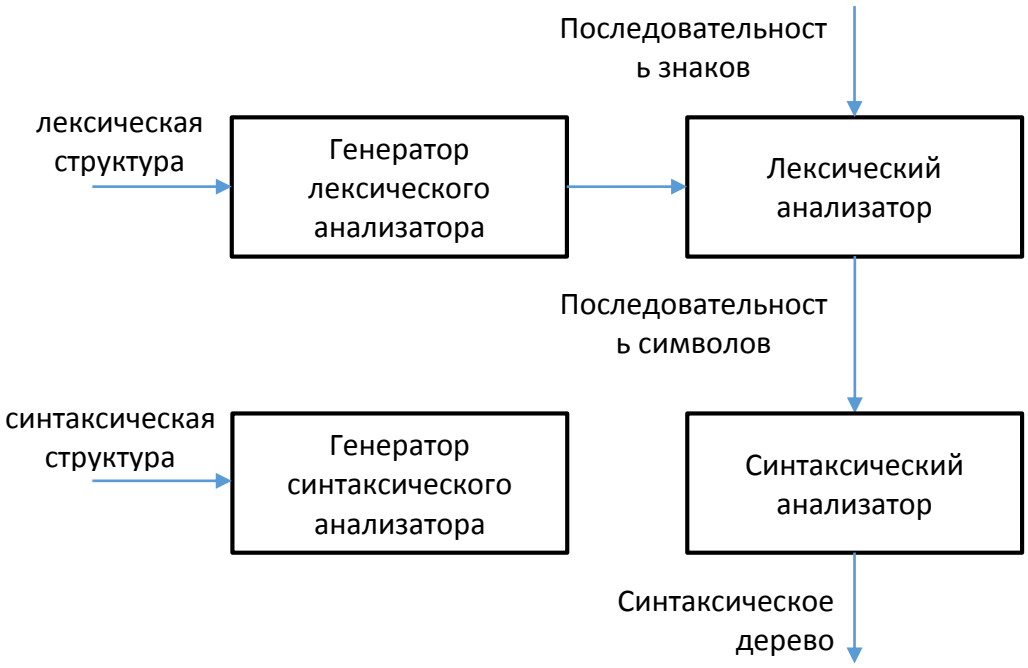

Рисунок 8. Автоматизация процесса анализа исходного кода

5.1. Регулярные выражение

Первой фазой анализа исходного текста программы является лексический анализ. Эта фаза достаточно проста и для нее достаточно мощности, предлагаемой регулярными грамматиками. Для описания регулярных языков существует другое, очень удобное представление, а именно, *регулярные выражения*. Для того, чтобы математически ввести это новое понятие, определим следующие операции.

Определение 24. *Объединением* двух языков L_1 и L_2 является язык $L_1 \cup L_2$, включающий все строки языка L_1 и все строки языка L_2.

Определение 25. *Конкатенацией* языков L_1 и L_2 является новый язык L_1L_2, все строки которого получены соединением каждой строки языка L_1 с каждой строкой языка L_2.

Определение 26. *Итерацией* языка L является новый язык, полученный в результате возведения каждой строки языка L в степень $n = 0,1, \dots \infty$.

Для записи перечисленных операций существуют удобные обозначения. Так, например, для обозначения операции итерации служит символ «*» (*звездочка Клини*). Запись x^* читается так «во входной строке символ *x* может встретиться ноль и более раз».

Символ «+» в выражении x^+y^+ читается, как «*x* может встретиться один и более раз, за которым следует *y*, который также может встретиться один или более раз». Строго говоря этот оператор необязателен, так как может быть заменен xx^*yy^*.

Определение 27. *Регулярное выражение*, определенное для алфавита *V*, представляет собой:

1. Пустую строку – ε;
2. Любой символ из алфавита *V*.

Если P и Q являются регулярными выражениями, то регулярными также являются выражения:

3. PQ;
4. $P|Q$;
5. P^*,

т.е. выражения полученные с помощью *конкатенации*, *объединения* и *итерации*, соответственно.

Пример 21. Регулярное выражение $(aab|ab)^*$ допускает строки вида:

$$\varepsilon$$
$$aabaabab$$
$$ababab$$
$$aababaabab$$
и т. д.

Оператор итерации имеет самый высокий приоритет, а оператор объединения самый низкий.

Далеко не каждый язык программирования можно определить с помощью регулярных выражений. Более того, большинство существующих языков программирования не поддаются описанию с помощью регулярных выражений. Однако механизм регулярных выражений часто используется на фазе лексического анализа для определения символа языка через составляющие его знаки. Например, регулярное выражение

$$I(I|d)^*, \qquad (3.1)$$

определяет идентификатор для многих языков программирования. В самом деле, если считать, что I – любая буква, а d – любая цифра, то данное регулярное можно расшифровать так: одна буква за которой следует произвольное количество букв или цифр.

Регулярные выражения широко используются для целей разбора текстов не только в задачах трансляции, но и в задачах обработки графики, аудио и видео файлов. Существуют расширенные описания регулярных выражений, которые включают операции, не перечисленные выше. Эти описания входят в состав стандартных библиотек практически всех языков программирования и операционных систем. К наиболее популярным расширениям относятся *POSIX* и *PCRE*.

5.1.1. Отличия регулярных и контекстно-свободных грамматик

При разработке компиляторов наиболее часто используются языки 2-го и 3-го типов, т.е. контекстно-свободные (КС) и регулярные. При этом, как уже говорилось выше, КС-языки включают в себя все регулярные языки в том смысле, что грамматика 3-го типа одновременно является грамматикой 2-го типа (но не наоборот). КС-языки более сложны для анализа, чем регулярные языки. Следовательно, где только возможно следует использовать регулярные языки. Для этого необходимо уметь определять, является ли язык регулярным.

Введем понятие *рекурсии* в языке. Следующие продукции:

$$A \to Aa; \tag{3.2}$$

$$B \to cB; \tag{3.3}$$

$$C \to dCf, \tag{3.4}$$

содержат рекурсию, так как нетерминал, стоящий в левой части, присутствует также и в правой. Различают *левую* (3.2), *правую* (3.2) и *среднюю* (3.4) рекурсии, в зависимости от того, где располагается терминал в правой части продукции.

Необходимо отметить, что грамматики практически всех языков содержат рекурсию, так как это единственный способ описать язык, допускающий произвольно большие предложения.

Также отметим, что существует еще и *непрямая* рекурсия, например,

$$A \to Bc;$$
$$B \to Cd;$$
$$C \to Ae.$$

Первый шаг в определении того, является ли язык регулярным, состоит в том, содержит ли грамматика рекурсию. Если – нет, следовательно, язык содержит конечное множество предложений, и является регулярным.

Утверждение 4. Если язык не содержит среднюю рекурсию, то он является регулярным.

Мы примем это утверждение без доказательства. Таким образом, язык, описываемый следующими грамматическими правилами, является регулярным

$$S \to XY|x|y|\varepsilon;$$
$$X \to xX|x;$$
$$Y \to yY|y.$$

С этим языком мы уже встречались $L = \{x^n y^m | n, m \geq 0\}$.

С другой стороны, язык, описываемый следующими продукциями, не является регулярным, поскольку содержит среднюю рекурсию

$$S \to xSy|xy.$$

С этим языком мы также уже встречались $L = \{x^n y^n | n > 0\}$.

Поговорим о связи только что введенных понятий с принципами построения компиляторов языка программирования. Лексика большинства языков программирования, к которой относятся имена переменных, литералы, константы, многознаковые операнды (например, ++), практически всегда можно описать языком регулярных выражений.

С другой стороны, выражения (например, арифметические выражения), которые составляют синтаксис языка, описать с помощью регулярных выражений невозможно. Для примера рассмотрим встречающуюся во многих языках возможность вкладывать одно выражение в другое с помощью скобок. Грамматически это можно описать так:

$$S \to (S);$$
$$S \to SS;$$
$$S \to \varepsilon.$$

Здесь присутствует средняя рекурсия, что делает грамматику нерегулярной.

5.1.2. Ограничения КС-грамматик

Как уже говорилось основой для построения трансляторов большинства современных языков программирования являются грамматики 2-го типа или *контекстно-свободные* грамматики. Существуют языки, достаточно простые, но для которых не существует КС-грамматики.

Пример 22. $\{a^m | m - $ положительная степень двойки$\}$.

Пример 23. $\{a^n b^n c^n | n \geq 0\}$.

Учитывая это, естественно возникает вопрос – насколько КС-грамматики подходят для генерации языков программирования?

Утверждение 5. В общем случае невозможно создать КС-грамматику, которая учитывала бы все возможные конструкции языка.

Это связано с тем, что многие выражения таких языков требуют явного знания *контекста применения*.

Пример 24. Ниже приведен листинг фрагмента программы на языке *C*

Листинг 1

```
1) int f(int, int);
2) int main() { return f(4,5,6); }
```

Здесь функция f вызывается в тремя параметрами вместо двух. Это пример того, как информация, которая требуется для правильной интерпретации кода в строке 2, содержится в строке 1.

Таким образом, использование КС-грамматики требует введения дополнительных ограничений на вход. Трансляция КС-языков программирования производится в два этапа:

1. Проверка синтаксиса на соответствие контекстно-свободной грамматике.

2. Синтаксически корректные выражения проверяются на соответствие дополнительным ограничениям.