# **Приветствие**

# **Введение**

Черная коробка – пожалуй так можно описать то, что представляет большинство при вопросе о работе компилятора. Компилятор — это программа, которая переводит текст, написанный на языке программирования, в машинные коды. С помощью компиляторов компьютеры могут понимать разные языки программирования

# **Этапы компиляции**

Компиляция— сборка программы, включающая:

1. Трансляцию всех модулей программы, написанных на одном или нескольких исходных языках программирования, в эквивалентные программные модули на низкоуровневом языке, машинному коду или непосредственно на машинном языке;
2. последующую сборку исполняемой машинной программы, в том числе вставка в программу кода всех функций, импортируемых из статических библиотек и/или генерация кода запроса к ОС на загрузку динамических библиотек, из которых функции будут вызываться программой.

# **Трансляция программы**

* Лексический анализ. На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем;
* Синтаксический (грамматический) анализ. Последовательность лексем преобразуется в древо разбора;
* Семантический анализ. На этой фазе древо разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их объявлениям, типам данных, проверка совместимости, определение типов выражений и т. д. Результат обычно называется «промежуточным представлением/кодом», и может быть дополненным древом разбора, новым деревом, абстрактным набором команд или чем-то ещё, удобным для дальнейшей обработки;
* Оптимизация. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах — например, над промежуточным кодом или над конечным машинным кодом;
* Генерация кода. Из промежуточного представления порождается код на целевом машинно-ориентированном языке.

# **Лексический анализ**

Компилятор считывает исходный код и разбивает его на элементарные части, называемые токенами. Это слова и символы кода. Посмотрим, как это выглядит на деле! Для этого воспользуемся инструментом лексического анализа – flex. Все примеры демонстрируются на операционной системе linux.

# **Flex**

Сейчас вы видите код простейшего лексического анализатора на flex. Фактически, при помощи регулярных выражений мы определяем шаблоны токенов и задаем имена для результатов поиска.

# **Результат работы**

Запустив этот анализатор, подадим на вход две строки. Как видите, на выходе получаем разбиение строк на токены. Цель работы лексического анализатора состоит в том, чтобы подготовить входную последовательность для синтаксического анализа.

# **Синтаксический анализ**

Синтаксический анализ – процесс сопоставления линейной последовательности лексем естественного или формального языка с его грамматикой. Результатом обычно является дерево разбора или, по-другому, синтаксическое дерево. Синтаксический анализатор – это программа или ее часть, которая выполняет синтаксический анализ. Давайте сделаем небольшой калькулятор при помощи лексического и синтаксического анализаторов. В качестве синтаксического анализатора воспользуемся инструментом bison.

# **БНФ грамматика**

Прежде, чем перейдем к коду, хотелось бы обратить ваше внимание на формальную систему описания синтаксиса или «Форму Бэкуса-Наура». БНФ используется для описания контекстно свободных формальных грамматик, БНФ конструкции имеют вид:

*<определяемый символ> ::= <посл.1> | <посл.2> | . . . | <посл.n>*

Такая конструкция определяет правила замены символа на последовательность букв и символов.

# **Bison**

На слайде слева вы можете увидеть код лексического анализатора, а справа код синтаксического анализатора с БНФ-конструкциями. Внутри синтаксического анализатора уже определены токены, которые он будет получать от лексического, а также описано как интерпретировать связи между ними.

# **Результат работы**

Вводя строки, лексический анализатор разбивает их на токены, а синтаксический сопоставляет их с нашими правилами, созданными при помощи БНФ-грамматики. Как итог, мы получили простой калькулятор.

# **Сематический анализ**

На этом этапе компилятор использует абстрактное синтаксическое дерево для обнаружения любых семантических ошибок, например:

* присвоение переменной неправильного типа
* объявление переменных с одинаковым именем в одной области видимости
* использование необъявленной переменной
* использование ключевого слова языка в качестве имени переменной

Семантический анализ можно разделить на три этапа:

* Проверка типов – проверяет соответствие типов в операторах присваивания, арифметических операциях, функциях и вызовах методов.
* Проверка управления потоком – проверяет, правильно ли используются структуры управления потоком и осуществляется ли правильный доступ к классам и объектам.
* Проверка меток – проверяет использование меток и идентификаторов.

# **Оптимизация**

Оптимизация - в компиляторе используются различные методы получения более оптимального программного кода при сохранении его функциональных возможностей. Наиболее распространённые цели оптимизации: сокращение времени выполнения программы, повышение производительности, компактификация программного кода, экономия памяти, минимизация энергозатрат, уменьшение количества операций ввода-вывода.

# **Генерация кода**

Генерация кода — это последний и заключительный этап работы компилятора. Он получает входные данные от этапов оптимизации кода и в результате создает код страницы или объектный код. Целью этого этапа является выделение памяти и создание перемещаемого машинного кода.

Он также выделяет ячейки памяти для переменной. Инструкции промежуточного кода преобразуются в машинные инструкции. На этом этапе оптимизируемый или промежуточный код преобразуется в машинный код.

Таким образом, на этом этапе также выбираются и распределяются все ячейки памяти и регистры. Код, сгенерированный на этом этапе, выполняется для приема входных данных и генерации ожидаемых выходных данных.

# **Статический анализ и clang**

Если вам интересно углубиться в тему статического анализа, но нет желания изучать flex и бизон, то вы можете воспользоваться удобным транслятором clang. Стоит помнить, что он работает только с С-подобными языками:

*C, C++, Objective-C, Objective-C++*

# **Clang**

Пример получения токенов при помощи программы написанной на Python.

# **Результат**

Слева оригинальный исходный код, слева перечисление токенов. Можете самостоятельно сопоставить их между собой, чтобы убедиться в точности и удобстве.

# **Спасибо за внимание**

Все исходные коды можно посмотреть/получить по ссылке на гит. Если есть какие-то вопросы, то вы можете связаться со мной по указанной электронной почте.

# **Полезная литература**

# **Глоссарий**