Лекция №1 Системное программное обеспечение для компиляции программного кода.

Место компилятора в программном обеспечении

Компиляторы составляют одну из важнейших частей системного программного обеспечения. Это связано с тем, что языки высокого уровня стали основным средством разработки программ. Только очень незначительная часть программного обеспечения, требует особой эффективности, программируется с помощью ассемблера. В настоящее время распространены довольно много языков программирования. С другой стороны, постоянно растущая потребность в новых компиляторах связана с бурным развитием архитектуры ЭВМ. Это развитие идет по разным направлениям. Совершенствуются старые архитектуры как в концептуальном отношении, так и по отдельным, конкретным линиям. Это можно проиллюстрировать на примере процессора Intel-80X86. Последовательные версии этого процессора 8086, 80186, 80286, 80386, 80486, 80586, Pentium и т.д. отличаются не только техническими характеристиками, но и, что более важно, новыми возможностями и, получается, изменением (расширением) системы команд. Естественно, это требует новых компиляторов (или модификации старых). .компиляторы ля многих языков программирования. Здесь необходимо также отметить, что новые архитектуры требуют разработки совершенно новых подходов к созданию компиляторов, так что наряду с собственно разработкой компиляторов ведется и большая научная работа по созданию новых методов трансляции.

структура компилятора

На фазе лексического анализа (ЛА) входная программа, которая представляет собой поток символов, разбивается на лексемы - слова в соответствии с определениями языка. Основным формализмом, что лежит в основе реализации лексических анализаторов, являются конечные автоматы и регулярные выражения. Лексический анализатор может работать в двух основных режимах: либо как подпрограмма, вызываемая синтаксическим анализатором за очередной лексемой, или как полный проход, результатом которого является файл лексем. В процессе выделения лексем ЛА может как самостоятельно строить таблицы имен и констант, так и выдавать значение для каждой лексемы при

очередном обращении к нему. В этом случае таблица имен строится в последующих фазах (например, в процессе синтаксического анализа.

На этапе ЛА обнаруживаются некоторые (самые простые) ошибки (недопустимые символы, неправильная запись чисел, идентификаторов и др.). Основная задача синтаксического анализа - разбор структуры программы. Как правило, под структурой понимается дерево, соответствующее разбору в контекстно-свободной грамматике языка. В настоящее время чаще всего используется либо LL (1) -анализ (и его вариант - рекурсивный спуск), или LR (1) -анализ и его варианты (LR (0), SLR (1), LALR (1) и другие). Рекурсивный спуск чаще используется при ручном программировании синтаксического анализатора, LR (1) - при использовании систем автоматизации построения синтаксических анализаторов. результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицу имен. В процессе синтаксического анализа также обнаруживаются ошибки, связанные с структурой программы. На этапе контекстного анализа выявляются зависимости между частями программы, которые не могут быть описаны контекстно свободным синтаксисом. Это в основном связи "опис- использования", в частности анализ типов объектов, анализ областей видимости, соответствие параметров, метки и другие. В процессе контекстного анализа строится таблица символов, которую можно рассматривать как таблицу имен, пополнения информацией о описания (свойствах) объектов. Основным формализмом, который используется при контекстном анализе, является атрибутные грамматики. Результатом работы фазы контекстного анализа есть атрибутированное дерево программы. Информация об объектах может быть как рассредоточена в самом дереве, так и сосредоточена в отдельных таблицах символов. В процессе контекстного анализа также могут быть обнаружены ошибки, связанные с неправильным

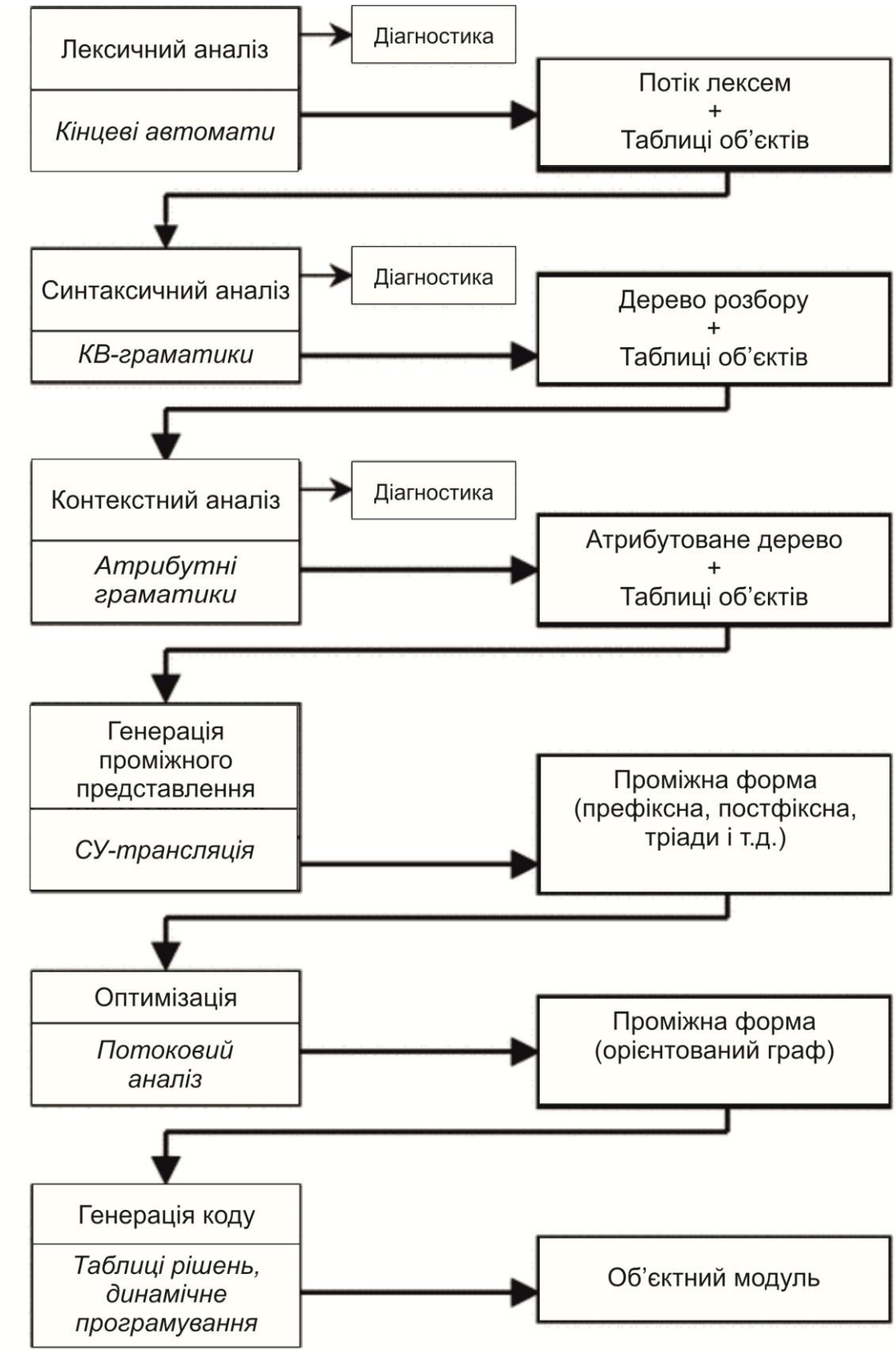

Рисунок 1.1 - Структура компилятора

использованием объектов. Затем программа может быть переведена в внутреннее представление. Это делается для целей оптимизации и / или удобства генерации кода. Еще одной целью преобразования программы во внутреннюю

представление есть желание иметь переносимый компилятор. Тогда только последняя фаза (Генерация кода) является машинно-зависимой. В качестве внутреннего представления может использоваться префиксний или постфиксний запись, ориентированный граф, тройки, четверки и другие. Фаз оптимизации может быть несколько. оптимизации конечно разделяют на машинно-зависимые и машинно-независимые, локальные и глобальные. Часть машинно-зависимой оптимизации выполняется на фазе генерации кода. Глобальная оптимизация пытается принять во внимание структуру всей программы, локальная - только небольших ее фрагментов. Глобальная оптимизация основывается на глобальном потоковый анализе, выполняется на графе программы и представляет собственно говоря преобразования этого графа. При этом могут учитываться такие свойства программы, как межпроцедурный анализ, межмодульный анализ, анализ областей жизни переменных и т.д. Наконец, генерация кода - последняя фаза трансляции. Результатом ее является или ассемблерный модуль, или объектный (или загрузочный) модуль. В процессе генерации кода могут выполняться некоторые локальные оптимизации, такие как распределение регистров, выбор длинных или коротких переходов, учет стоимости команд при выборе конкретной последовательности команд. Для генерации кода разработаны различные методы, такие как таблицы решений, сопоставление образцов, включая динамическое программирование, различные синтаксические методы. Конечно, те или иные фазы транслятора могут быть либо отсутствуют вовсе, либо сочетаться. В простейшем случае однопроходной транслятора форуме явной фазы генерации промежуточного представления и оптимизации, другие фазы объединены в одну, причем нет и явно построенного синтаксического дерева.