ОГЛАВЛЕНИЕ

Место компилятора в программном обеспечении 3

Основные принципы работы синтаксического анализатора 6

Дерево разбора. Преобразование дерева разбора в дерево операций 8

Автоматизация построения синтаксических анализаторов (программа YACC) 10

Назначение семантического анализа 12

Этапы семантического анализа 13

Идентификация лексических единиц языков программирования 16

Список использованных источников 19

# Место компилятора в программном обеспечении

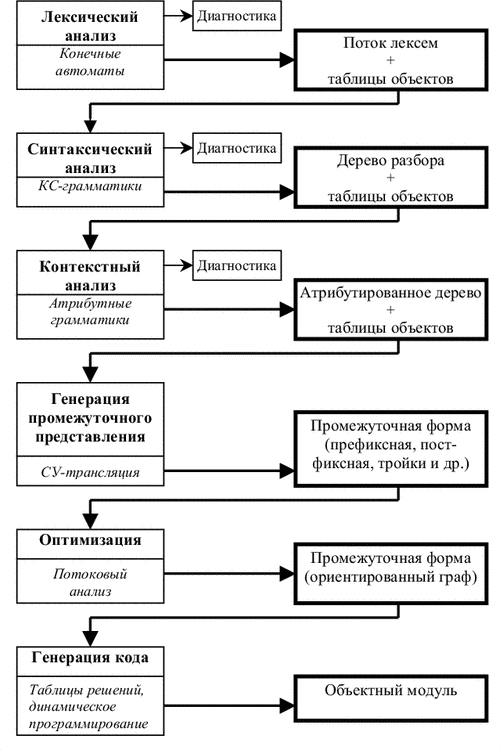
Компиляторы составляют существенную часть программного обеспечения ЭВМ. Это связано с тем, что языки высокого уровня стали основным средством разработки программ. Только очень незначительная часть программного обеспечения, требующая особой эффективности, программируется с помощью ассемблеров. В настоящее время распространено довольно много языков программирования. Наряду с традиционными языками, такими, как Фортран, широкое распространение получили так называемые «универсальные» языки (Паскаль, Си, Модула-2, Ада и другие), а также некоторые специализированные (например, язык обработки списочных структур Лисп). Кроме того, большое распространение получили языки, связанные с узкими предметными областями, такие, как входные языки пакетов прикладных программ.

Для некоторых языков имеется довольно много реализаций. Например, реализаций Паскаля, Модулы-2 или Си для ЭВМ типа IBM PC на рынке десятки.

С другой стороны, постоянно растущая потребность в новых компиляторах связана с бурным развитием архитектур ЭВМ. Это развитие идет по различным направлениям. Совершенствуются старые архитектуры как в концептуальном отношении, так и по отдельным, конкретным линиям. Это можно проиллюстрировать на примере микропроцессора Intel-80X86. Последовательные версии этого микропроцессора 8086, 80186, 80286, 80386, 80486, 80586 отличаются не только техническими характеристиками, но и, что более важно, новыми возможностями и, значит, изменением (расширением) системы команд. Естественно, это требует новых компиляторов (или модификации старых). То же можно сказать о микропроцессорах Motorola 68010, 68020, 68030, 68040.

В рамках традиционных последовательных машин возникает большое число различных направлений архитектур. Примерами могут служить архитектуры CISC, RISC. Такие ведущие фирмы, как Intel, Motorola, Sun, начинают переходить на выпуск машин с RISC-архитектурами. Естественно, для каждой новой системы команд требуется полный набор новых компиляторов с распространенных языков.

Наконец, бурно развиваются различные параллельные архитектуры. Среди них отметим векторные, многопроцессорные, с широким командным словом (вариантом которых являются суперскалярные ЭВМ). На рынке уже имеются десятки типов ЭВМ с параллельной архитектурой, начиная от супер-ЭВМ (Cray, CDC и другие), через рабочие станции (например, IBM RS/6000) и кончая персональными (например, на основе микропроцессора I-860). Естественно, для каждой из машин создаются новые компиляторы для 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.



На фазе лексического анализа входная программа, представляющая собой поток литер, разбивается на лексемы - слова в соответствии с определениями языка. Основными формализмами, лежащим 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 для получения очередной лексемы, либо как полный проход, результатом которого является файл лексем.

В процессе выделения лексем лексический анализатор может как самостоятельно строить таблицы объектов (идентификаторов, строк, чисел и т.д.), так и выдавать значения для каждой лексемы при очередном к нему обращении. В этом случае таблицы объектов строятся в последующих фазах (например, в процессе синтаксического анализа).

На этапе лексического анализа обнаруживаются некоторые (простейшие) 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), либо 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 автоматического построения синтаксических анализаторов.

Результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицы объектов. В процессе синтаксического анализа также обнаруживаются ошибки, связанные со структурой программы.

На этапе контекстного анализа выявляются зависимости между частями программы, которые не могут быть описаны контекстно-свободным синтаксисом. Это в основном связи «описание-использование», в частности, анализ типов объектов, анализ областей видимости, соответствие параметров, метки и другие. В процессе контекстного анализа таблицы объектов пополняются информацией об описаниях (свойствах) объектов.

Основным формализмом, использующимся при контекстном анализе, является аппарат атрибутных грамматик. Результатом контекстного анализа является атрибутированное дерево программы. Информация об объектах может быть как рассредоточена в самом дереве, так и сосредоточена в отдельных таблицах объектов. В процессе контекстного анализа также могут быть обнаружены ошибки, связанные с неправильным использованием объектов.

Затем программа может быть переведена во внутреннее представление. Это делается для целей оптимизации и/или удобства генерации кода. Еще одной целью преобразования программы во внутреннее представление является желание иметь переносимый компилятор. Тогда только последняя фаза (генерация кода) является машинно-зависимой. В качестве внутреннего представления может использоваться префиксная или постфиксная запись, ориентированный граф, тройки, четверки и другие.

Фаз оптимизации может быть несколько. Оптимизации обычно делят 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 потоковом анализе, который выполняется на графе программы и представляет по существу преобразование 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 либо ассемблерный модуль, либо 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, включающее динамическое программирование, различные синтаксические методы.

Конечно, те или иные фазы транслятора могут либо отсутствовать совсем, либо объединяться. В простейшем случае однопроходного транслятора нет явной фазы генерации промежуточного представления и оптимизации, остальные фазы объединены в одну, причем нет и явно построенного синтаксического дерева.

# Основные принципы работы синтаксического анализатора

Синтаксический анализатор (синтаксический разбор) — это часть компилятора, которая отвечает за выявление основных синтаксических конструкций входного языка. В задачу синтаксического анализа входит: найти и выделить основные синтаксические конструкции в тексте входной программы, установить тип и проверить правильность каждой синтаксической конструкции и, наконец, представить синтаксические конструкции в виде, удобном для дальнейшей генерации текста результирующей программы.

В основе синтаксического анализатора лежит распознаватель текста входной программы на основе грамматики входного языка. Как правило, синтаксические конструкции языков программирования могут быть описаны с помощью КС-грамматик, реже встречаются языки, которые, могут быть описаны с помощью регулярных грамматик. Чаще всего регулярные грамматики применимы к языкам ассемблера, а языки высокого уровня построены па основе синтаксиса КС-языков. Распознаватель дает ответ на вопрос о том, принадлежит или нет цепочка входных символов заданному языку. Однако, как и в случае лексического анализа, задача синтаксического разбора не ограничишься только проверкой принадлежности цепочки заданному языку. Необходимо выполнить все перечисленные выше задачи, которые должен решить синтаксический анализатор. В таком варианте анализатор уже не является разновидностью МП-автомата — его функции можно трактовать шире. Синтаксический анализатор должен иметь некий выходной язык, с помощью которого он передает следующим фазам компиляции не только информацию о найденных и разобранных синтаксических структурах. В таком случае он уже является преобразователем с магазинной памятью — МП-преобразователем. Синтаксический разбор — это основная часть компилятора на этапе анализа. Без выполнения синтаксического разбора работа компилятора бессмысленна, в то время как лексический разбор в принципе является необязательной фазой. Все задачи по проверке синтаксиса входного языка могут быть решены на этапе синтаксического разбора. Сканер только позволяет избавить сложный по структуре синтаксический анализатор от решения примитивных задач по выявлению и запоминанию лексем исходной программы.

Выходом лексического анализатора является таблица лексем (или цепочка лексем). Эта таблица образует вход синтаксического анализатора, который исследует только один компонент каждой лексемы — ее тип. Остальная информация о лексемах используется на более поздних фазахкомпиляции при семантическом анализе, подготовке к генерации и генерации кода результирующей программы. Синтаксический анализ (пли разбор) — это процесс, в котором исследуется таблица лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным и определении синтаксиса языка.

Синтаксический анализатор воспринимает выход лексического анализатора и разбирает его в соответствии с грамматикой входного языка. Однако в грамматике входного языка программирования обычно не уточняется, какие конструкции следует считать лексемами. Примерами конструкций, которые обычно распознаются во время лексического анализа, служат ключевые слова, константы и идентификаторы. Но эти же конструкции могут распознаваться и синтаксическим анализатором. На практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие надо оставлять синтаксическому анализатору. Обычно это определяет разработчик компилятора исходя из технологических аспектов программирования, а также синтаксиса и семантики входного языка Далее рассмотрены технические аспекты, связанные с реализацией синтаксических анализаторов для использования результатов их работы на этане генерации кода. Тем не менее, основу любого синтаксического анализатора всегда составляет распознаватель, построенный на основе какого-либо класса КС-грамматик. Поэтому главную роль и том, как функционирует синтаксический анализатор и какой алгоритм лежит в его основе, играют принципы построения распознавателей КС-языков. Без применения этих принципов невозможно выполнить эффективный синтаксический разбор предложений входного языка.

# Дерево разбора. Преобразование дерева разбора в дерево операций

Результатом работы распознавателя КС-грамматики входного языка является последовательность правил грамматики, примененных для построения входной цепочки. По найденной последовательности, зная тип распознавателя, можно построить цепочку вывода или дерево вывода. В этом случае дерево вывода выступает в качестве дерева синтаксического разбора и представляет собой результат работы синтаксического анализатора в компиляторе.

Однако ни цепочка вывода, ни дерево синтаксического разбора не являются целью работы компилятора. Дерево вывода содержит массу избыточной информации, которая для дальнейшей работы компилятора не требуется. Эта информация включает в себя все нетерминальные символы, содержащиеся и узлах дерева, — после того как дерево построено, они не несут никакой смысловой нагрузки и не представляют для дальнейшей работы интереса.

Для полного представления о типе и структуре найденной и разобранной синтаксической конструкции входного языка в принципе достаточно знать последовательность номеров правил грамматики, примененных для ее построения. Однако форма представления этой достаточной информации может быть различной как в зависимости от реализации самого компилятора, так от фазы компиляции. Эта форма напивается внутренним представлением программы.

В синтаксическом дереве внутренние узлы (вершины) соответствуют операциям, а листья представляют собой операнды. Как правило, листья синтаксического дерена сляпаны с записями в таблице идентификаторов. Структура синтаксического дерева отражает синтаксис языка программирования, на котором наш капа исходная программа.

Синтаксические деревья могут быть построены компилятором для любой части входной программы. Не всегда синтаксическому дереву должен соответствовать фрагмент кода результирующей программы — например, возможно построение синтаксических деревьев для декларативной части языка. В этом случае операции, имеющиеся в дереве, не требуют порождения объектного кода, но несут информацию о действиях, которые должен выполнить сам компилятор над соответствующими элементами. В том случае, когда синтаксическому дереву соответствует некоторая последовательность операций, влекущая порождение фрагмента объектного кода, говорят о дереве операций.

Дерево операций можно непосредственно построить из дерева вывода, порожденного синтаксическим анализатором. Для этого достаточно исключить из дерева вывода цепочки нетерминальных символов, а также узлы, не несущие семантической нагрузки при генерации кода. Примером таких узлов могут служить различные скобки, которые меняют порядок выполнения операции и операторов, но после построения дерева никакой смысловой нагрузки не несут, гак каким не соответствует никакой объектным код.

То, какой узел в дерене является операцией, а какой — операндом, никак невозможно определить из грамматики, описывающей синтаксис входного языка. Также ниоткуда не следует, каким операциям должен соответствовать объектный код в результирующей программе, а каким — нет. Все это определяется только исходя из семантики — «смысла» — языка входной программы. Поэтому только разработчик компилятора может четко определить, как при построении дерева операции должны различаться операнды и сами операции, а также то, какие операции являются семантически незначащими для порождения объектного кода.

Алгоритм преобразования дерева семантического разбора и дерево операции можно представить следующим образом.

*Шаг 1.* Если в дереве больше не содержится узлов, помеченных нетерминальными символами, то выполнение алгоритма завершено; иначе — перейти к шагу 2

*Шаг. 2.* Выбрать крайний левый узел дерена, помеченный нетерминальным символом грамматики и сделать его текущим. Перейти к шагу 3.

*Шаг 3.* Если текущий узел имеет только один нижележащий узел, то текущий узел необходимо удалить из дерена, а связанный с ним узел присоединить к узлу вышележащего уровня (исключить из дерена цепочку) и вернуться к шагу 1; иначе – перейти к шагу 4.

*Шаг 4.* Если текущий узел имеет нижележащий узел (лист дерева), помеченный терминальным символом, который не несет семантической нагрузки, тогда этот лист нужно удалить из дерева и вернуться к шагу 3; иначе — перейти к шагу 5.

*Шаг 5.* Если текущий узел имеет один нижележащий узел (лист дерева), помеченный терминальным символом, обозначающим знак операции, а остальные узлы помечены как операнды, то лист, помеченный знаком операции, надо удалить из дерева, текущий узел пометить этим знаком операции и перейти к шагу 1; иначе — перейти к шагу 6.

*Шаг 6.* Если среди нижележащих узлов для текущего узла есть узлы, помеченные нетерминальными символами грамматики, то необходимо выбрать крайний левый среди этих узлов, сделать его текущим умом и перейти к шагу 3: иначе — выполнение алгоритма завершено.

# Автоматизация построения синтаксических анализаторов (программа YACC)

При разработке различных прикладных программ часто возникает задача синтаксического разбора некоторого входного текста. Конечно, ее можно всегда решить, полностью самостоятельно построив соответствующий анализатор. И хотя задача выполнения синтаксического разбора встречается не столь часто, как задача выполнений лексического разбора, но все-таки и для ее решения были предложены соответствующие программные средства.

Автоматизированное построение синтаксических анализаторов может быть выполнено с помощью программы YACC. Программа YACC (Yet Another Compiler Compiler) предназначена для построения синтаксического анализатора контекстно-свободного языка. Анализируемый язык описывается с помощью грамматики к пиле, близком форме Бэкуса— Наура (нормальная форма Бэкуса—Наура — НФБН). Результатом работы YACC является исходный текст программы синтаксического анализатора. Анализатор, который порождается YACC, реализует восходящий LALR(l) распознаватель.

Как и программа LEX, служащая дли автоматизации построении лексических анализаторов, программа YACC тесно связана с историей операционных систем типа UNIX. Эта программа входит в поставку многих версий ОС UNIX или Linux. Поэтому чаще всего результатом работы YACC является исходный текст синтаксического распознавателя на языке С, Однако существуют версии YACC, выполняющиеся под управлением ОС, отличных от UNIX, и порождающие исходный код на других языках программирования (например, Pascal). Принцип работы YACC похож на принцип работы LEX: на вход поступает файл, содержащий описание грамматики заданного КС-языка, а на выходе получаем текст программы синтаксического распознавателя, который, естественно, можно дополнять и редактировать, как и любую другую программу на заданном языке программирования.

Исходная грамматика для YACC состоит из трех секций, разделенных символом *%%, —* секции описаний, секции правил, в которой описывается грамматика, и секции программ, содержимое которой просто копируется в выходной файл. Например, ниже приведено описание простейшей грамматики для YACC, которая соответствует грамматике арифметических выражений:

%token a

%start e

%

e : e ‘+‘ m | e ‘-‘ m | m

m : m ‘\*’ t | m ‘/’ t | t

a : a | ‘(’ e ‘)’ :

%%

Секция описаний содержит информацию отом, что идентификатор а является лексемой (терминальным символом) гpамматики, а символ е — ее начальным нетерминальным символом.

Грамматика, записана обычным образом — идентификаторы обозначают терминальные и нетерминальные символы; символьные константы типа '+' и '-' считаются терминальными символами. Символы :, |, ; принадлежат к метаязыку YACC и читаются согласно НФБН «есть по определению», «или» и «конец правила» соответственно.

В отличие от LEX, который всегда способен состроить лексический распознаватель, если входной файл содержит правильное регулярное выражение, YACC не всегда может построить распознаватель, даже если входной язык задан правильной КС-грамматикой. Ведь заданная грамматика может и не принадлежать к классу LALR(l). В этом случае YACC выдаст сообщение об ошибке (наличии неразрешимого LALR(t) конфликта в грамматике) при построении синтаксического анализатора. Тогда пользователь должен либо преобразовать грамматику, либо задать YACC некоторые дополнительные правила, которые могут облегчить построение анализатора. Например, YACC позволяет указать правила, явно задающие приоритет операций и порядок их выполнения (слева направо или справа налево).

С каждым правилом грамматики может быть связано действие, которое будет выполнено при свертке по данному правилу. Оно записывается в виде заключенной в фигурные скобки последовательности операторов языка, на котором порождается исходный текст программы распознавателя (обычно это язык С). Последовательность должна располагаться после правой части соответствующего правила. Также YACC позволяет управлять действиями, которые будут выполняться распознавателем в том случае, если входная цепочка не принадлежит заданному языку. Распознаватель имеет возможность выдать сообщение об ошибке, остановиться либо же продолжить разбор, предпринял некоторые действия, связанные с попыткой локализовать либо устранить ошибку во входной цепочке.

# Назначение семантического анализа

Практически вся языки программирования, строго говоря, не являются КС-языками. Поэтому полный разбор цепочек символов входного языка компилятор не может выполнить в рамках КС-языков с помощью КС-грамматик и МП-аптоматов. Полный распознаватель для большинства языков программирования может быть построен в рамках КЗ-языков, поскольку все реальные языки программирования контекстно-зависимы.

Итак, полный распознаватель для языка программирования можно построить на основе распознавателя КЗ-языка. Однако известно, что такой распознаватель имеет экспоненциальную зависимость требуемых для выполнения разбора цепочки вычислительных ресурсов от длины входной цепочки. Компилятор, построенный на основе такого распознавателя, будет неэффективным с точки зрения либо скорости работы, либо объема необходимой памяти. Поэтому такие компиляторы практически не используются, а все реально существующие компиляторы на этапе разбора входных цепочек проверяют только синтаксические конструкции входного языка, не учитывая его семантику.

С целью повысить эффективность компиляторов разбор цепочек входного языка выполняется в два этапа: первый — синтаксический разбор на основе распознавателя одного из известных классов КС-языков; второй — семантический анализ входной цепочки.

Для проверки семантической правильности входной программы необходимо иметь всю информацию о найденных лексических единицах языка. Эта информация помещается в таблицу лексем на основе конструкций, найденных синтаксическим распознавателем. Примерами таких конструкциями являются блоки описания констант и идентификаторов (если они предусмотрены семантикой языка*)* пли операторы, где тот или иной идентификатор встречается впервые (если описание происходит по факту первого использования). Поэтому полный семантический анализ входной программы может быть произведен только после полного завершения её синтаксического разбора.

Таким образом, входными данными для семантического анализа служат:

* таблица идентификаторов;
* результаты разбора синтаксических конструкций входного языка.

Результаты выполнения синтаксического разбора могут быть представлены в одной из форм внутреннего представления программы в компиляторе. Как правило, на этапе семантического анализа используются различные варианты деревьев синтаксического разбора, поскольку семантический анализатор интересует прежде всего структура входной программы.

Семантический анализ обычно выполняется на двух этапах компиляции: на этапе синтаксического разбора и в начале этапа подготовки к генерации кода. В первом случае всякий раз по завершении распознавания определенной синтаксической конструкции входного языка выполняется её семантическая проверка на основе имеющихся в таблице идентификаторов данных (такими конструкциями, как правило, являются процедуры, функции и блоки операторов входного языка). Во втором случае, после завершения всей фазы синтаксического разбора, выполняется полный семантическим анализ программы на основании данных в таблице идентификаторов (сюда попадает, например, поиск неописанных идентификаторов). Иногда семантический анализ выделяют в отдельный этап (фазу) компиляции.

В каждом компиляторе обычно присутствуют оба варианта семантического анализатора.

# Этапы семантического анализа

Семантический анализатор выполняет следующие основные действия:

* проверка соблюдения во входной программе семантических соглашений входного языка;
* дополнение внутреннего представления программы в компиляторе операторами и действиями, неявно предусмотренными семантикой входного языка;
* проверка элементарных семантических (смысловых) норм языков программирования, напрямую не связанных с входным языком.

Проверка соблюдения во входной программе семантических соглашений входного языка заключается в сопоставлении входных цепочек программы с требованиями семантики входного языка программирования. Каждый язык программирования имеет четко заданные и специфицированные семантические соглашения, которые не могут быть проверены на этапе синтаксического разбора. Именно их в первую очередь проверяет семантический анализатор.

Примерами таких соглашении являются следующие требования:

* каждая метка, на которую есть ссылка, должна один раз присутствовать в программе;
* каждый идентификатор должен быть описан один раз, и ни один идентификатор не может быть описан более одного раза (с учетом блочной структуры описаний);
* все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операций;
* типы переменных в выражениях должны быть согласованы между собой;
* при вызове процедур и функций число и типы фактических параметров должны быть согласованы с числом и типами формальных параметров.

Например, если оператор языка Pascal имеет вид

a := b + c:

то с точки зрения синтаксического разбора это будет абсолютно правильный оператор. Однако, мы не можем сказать, является ли этот оператор правильным с точки зрения входного языка (Pasca]), пока не проверим семантические требования для всех входящих в него лексических элементов. Такими элементами здесь являются идентификаторы a, b и с. Не зная, что они собой представляют, мы не можем не только окончательно утверждать правильность приведенного выше оператора, но и понять ого смысл. Фактически необходимо знать описание этих идентификаторов.

В том случае, если хотя бы один из них не описан, имеет мест явная ошибка. Если это числовые переменные и константы, то мы имеем дело с оператором сложения, если же это строковые переменные и константы — с оператором конкатенации строк. Кроме того, идентификатор а, например, ни в коем случае не может быть константой — иначе нарушена семантика оператора присваивания. Также невозможно, чтобы одни из идентификаторов были числами, а другие строками, или, скажем, идентификаторами массивов или структур – такое сочетание аргументов для оператора сложения недопустимо.

Следует также отметить, что от семантических соглашений зависит не только правильность оператора, но и его смысл. Действительно, операции алгебраического сложения и конкатенации строк имеют различный смысл, хотя и обозначаются в рассмотренном примере одним знаком “+”. Следовательно от семантического анализатора зависит также и код результирующей программы.

Если какое-либо из семантических требований входного языка не выполняется, то компилятор выдает сообщение об ошибке и процесс компиляции на этом, как правило, прекращается.

Дополнение внутреннего представления программы операторами и действиями, неявно предусмотренными семантикой входного языка, связано с преобразованием типов операндов в выражениях и при передаче параметров в процедуры и функции.

Если вернуться к рассмотренному выше элементарному оператору языка Pascal:

a := b + c:

то можно отметить, что здесь выполняются две операции: одна операция сложения (или конкатенации, в зависимости от типов операндов) и одна операция присвоения результата. Соответствующим образом должен быть порожден и код результирующей программы.

Однако не все так очевидно просто, допустим, что где-то перед рассмотренным оператором мы имеем описание его операндов в виде:

Var

а : real;

b : integer;

c : double;

из этого описания следует, что а — вещественная переменная языка Pascal, b — целочисленная переменная, с — вещественная переменная с двойной точностью. Тогда смысл рассмотренного оператора с точки зрения входной программы существенным образом меняется, поскольку в языке Pascal нельзя напрямую выполнять операции над операндами различных типов. Существуют правила преобразования типов, принятые для данного языка. Кто выполняет эти преобразования?

Это может сделать разработчик программы — но тогда преобразования типов в явном виде будут присутствовать в тексте входной программы (в рассмотренном примере это не так). В другом случае это делает код, порождаемый компилятором, когда преобразования типов в явном виде в тексте программы не присутствуют, но неявно предусмотрены семантическими соглашениями языка. Для этого в составе библиотек функции, доступных компилятору, должны быть функции преобразования типов Вызовы этих функции как раз и будут встроены в текст результирующей программы для удовлетворения семантических соглашении о преобразованиях типов во входном языке, хотя в тексте программы в явном виде они не присутствуют. Чтобы это произошло, эти функции должны быть встроены и во внутреннее представление программы в компиляторе. За это также отвечает семантический анализатор.

С учетом предложенных типов данных, в рассмотренном пример будут не две, а четыре операции: преобразование целочисленной переменной b в формат вещественных чисел с двойной точностью; сложение двух вещественных чисел двойной точностью; преобразование результата в вещественное число с одинарной точностью; присвоение результата переменной c. Количество операций возросло вдвое, причем добавились нетривиальные функции преобразования типов. Преобразование типов — эго только один вариант операций, неявно добавляемых компилятором в код программы на основе семантических соглашении. Другим примером такого рода операций могут служить операции вычисления адреса, когда происходит обращение к элементам сложных структур данных. Существуют и другие варианты такого рода операций.

Таким образом, и здесь действия, выполняемые семантическим анализатором, существенным образом влияют на порождаемый компиляторомкод результирующей программы.

Проверка элементарных смысловых норм языков программирования, напрямую не снизанных с входным языком, — это сервисная функция, которую предоставляют большинство современных компиляторов. Эта функция обеспечивает проверку компилятором некоторых соглашений, применимых к большинству современных языков программирования, выполнение которых связано со смыслом как всей входной программы и целом, так и отдельных её фрагментов.

# Идентификация лексических единиц языков программирования

Идентификация переменных, типов, процедур, функций и других лексических единиц языков программирования – это установление однозначного соответствия между данными объектами и их именами в тексте исходной программы. Идентификация лексических единиц языка чаще всего выполняется на этапе семантического анализа.

Как правило, большинство языков программирования требуют, чтобы в исходной программе имена лексических единиц не совпадали как между собой, так и с ключевыми словами синтаксических конструкций языка. Однако, чаще всего этого бывает недостаточно, чтобы установить однозначное соотношение между лексическими единицами и их именами, поскольку существуют дополнительные смысловые ограничения, накладываемые языком на употребление эти имен.

Например локальные переменные в большинстве языков программирования имею область видимости, которая ограничивает употребление имени переменной рамками того блока исходной программы, где эта переменная описана. Это значит, что с одной стороны, такая переменная не может быть использована вне пределов своей области видимости. С другой стороны, имя переменной может быть не уникальным, поскольку в двух различных областях видимости допускается существование двух различных переменных с одинаковыми именами. Полный перечень таких ограничений зависит от семантики конкретного языка программирования. Все они четко заданы в описании языка и не могут допускать неоднозначности в толковании, но не могут быть полностью определены на этапе лексического разбора, а потому требуют от компилятора дополнительных действий на этапах синтаксического разбора и семантического анализа. Общая направленность этих действий такова, чтобы дать каждой лексической единице языка уникальное имя в пределах всей исходной программы и потом использовать это имя при синтезе результирующей программы.

Можно дать примерный перечень действий компиляторов для идентификации переменных, констант, функций, процедур и других лексических единиц языка:

* имена локальных переменных дополняются именами тех блоков (функций, процедур), в которых эти переменные описаны;
* имена внутренних переменных и функций модулей исходной программы дополняются именем самих модулей, причем это касается только внутренних имен и не должно происходить, если переменная или функция доступна извне модуля;
* имена процедур и функций, принадлежащих объектам (классам), в объектно-ориентированных языках программирования дополняются наименованием типа объекта (класса), которому они принадлежат;
* имена процедур и функций модифицируются в зависимости от типов их формальных аргументов.

Конечно, это далеко не полный перечень возможных действий компилятора, каждая реализация компилятора может предполагать свои набор действий. То, какие из них будут использоваться и как они будут реализованы на практике, зависит от языка исходной программы и разработчиков компилятора.

Как правило, уникальные имени, которые компилятор присваивает лексическим единицам языка, используются только во внутреннем представлении исходной программы компилятором, и человек, создавший исходную программу, не сталкивается с ними. Но они могут потребоваться пользователю в некоторых случаях – например, при отладке программы, при порождении текста результирующей программы на языке ассемблера или при использовании библиотеки, созданной версией компилятора для одного языка программирования и другом языке (или даже просто в другой версии компилятора). Тогда пользователь должен знать, по каким правилам компилятор порождает уникальные имена для лексических единиц исходной программы.

Во многих современных компиляторах (и обрабатываемых ими входных языках) предусмотрены специальные настройки и ключевые слова, которые позволяют отключить процесс порождения компилятором уникальных имен для лексических единиц языка. Эти слова учтены в специальных синтаксических конструкциях языка (как прилило, это конструкции, содержащие слона export пли external). Если пользователь использует эти средства, то компилятор не применяет механизм порождения уникальных имен для указанных лексических единиц. В этом случае разработчик программы сам отвечает за уникальность имени данной лексической единицы в пределах всей исходной программы или даже в пределах всего проекта, Если требование уникальности не будет выполняться, могут возникнуть синтаксические или семантические ошибки па стадии компиляции либо же другие ошибки на более поздних этапах разработки программного обеспечения. Поскольку наиболее широко используемыми лексическими единицами в различных языках программирования являются, как правило, имена процедур и функций, то этот вопрос, прежде всего, касается именно их.

# Список использованных источников

1. Серебряков – Языки программирования: http://infonet.cherepovets.ru/citforum/programming/theory/serebryakov
2. Свободная энциклопедия – Википедия http://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80