Основные принципы построения трансляторов

Выше были рассмотрены теоретические вопросы, связанные с теорией формальных языков и грамматик. В этой главе пойдет речь о том, как эти теоретические аспекты применяются на практике при построении трансляторов.

Трансляторы, компиляторы и интерпретаторы — общая схема работы

Определение транслятора, компилятора, интерпретатора

Для начала дадим несколько определений — что же все-таки такое есть уже многократно упоминавшиеся трансляторы и компиляторы.

Формальное определение транслятора

Транслятор — это программа, которая переводит программу на исходном (входном) языке в эквивалентную ей программу на результирующем (выходном) языке.

В этом определении слово «программа» встречается три раза, и это не ошибка и не тавтология. В работе транслятора, действительно, участвуют три программы.

Во-первых, сам транслятор является программой[1]. То есть, транслятор — это часть программного обеспечения (ПО), он представляет собой набор машинных команд и данных и выполняется компьютером, как и все прочие программы в рамках операционной системы (ОС). Все составные части транслятора представляют собой динамически загружаемые библиотеки или модули этой программы со своими входными и выходными данными.

Во-вторых, исходными данными для работы транслятора служит программа на исходном языке программирования — некоторая последовательность предложений входного языка. Эта программа называется *входной* или *исходной программой*. Обычно это символьный файл, но этот файл должен содержать текст программы, удовлетворяющий синтаксическим и семантическим требованиям входного языка. Кроме того, этот файл несет в себе некоторый смысл, определяемый семантикой входного языка. Часто файл, содержащий текст исходной программы, называют исходным файлом.

---

[1] Теоретически возможна реализация транслятора с помощью аппаратных средств. Автору встречались такого рода разработки, однако широкое практическое применение их не известно. В таком случае и все составные части транслятора могут быть реализованы в виде аппаратных средств и их фрагментов — вот тогда схема распознавателя, который является составной частью транслятора, может получить вполне практическое воплощение!

В-третьих, выходными данными транслятора является программа на результирующем языке. Эта программа называется *результирующей программой*. Результирующая программа строится по синтаксическим правилам выходного языка транслятора, а ее смысл определяется семантикой выходного языка.

Важным пунктом в определении транслятора является эквивалентность исходной и результирующей программ. Эквивалентность этих двух программ означает совпадение их смысла с точки зрения семантики входного языка (для исходной программы) и семантики выходного языка (для результирующей программы). Без выполнения этого требования сам транслятор теряет всякий практический смысл.

Итак, чтобы создать транслятор необходимо, прежде всего, выбрать входной и выходной языки. С точки зрения преобразования предложений входного языка в эквивалентные им предложения выходного языка транслятор выступает как переводчик. Например, трансляция программы с языка С в язык ассемблера по сути ничем не отличается от перевода, скажем, с русского языка на английский, с той только разницей, что сложность языков несколько иная (о том, почему не существует трансляторов с естественных языков, сказано в предыдущей главе). Поэтому и само слово «транслятор» (английское: translator) означает «переводчик».

Результатом работы транслятора будет результирующая программа, но только в том случае, если текст исходной программы является правильным — не содержит ошибок с точки зрения синтаксиса и семантики входного языка. Если исходная программа неправильная (содержит хотя бы одну ошибку), то результатом работы транслятора будет сообщение об ошибке (с дополнительными пояснениями и указанием места ошибки в исходной программе). В этом смысле транслятор сродни переводчику, например, с английского, которому подсунули неверный текст.

Определение компилятора. Отличие компилятора от транслятора

Кроме понятия «транслятор» широко употребляется также близкое ему по смыслу понятие «компилятор».

Компилятор — это транслятор, который осуществляет перевод исходной программы в эквивалентную ей результирующую программу на языке машинных команд или на языке ассемблера.

Таким образом, компилятор отличается от транслятора лишь тем, что его результирующая программа всегда должна быть написана на языке машинных кодов или на языке ассемблера. Результирующая программа транслятора, в общем случае, может быть написана на любом языке — возможен, например, транслятор программ с языка Pascal на язык С.

ВНИМАНИЕ

> Всякий компилятор является транслятором, но не наоборот — не всякий транслятор будет компилятором. Например, упомянутый выше транслятор с языка Pascal на C компилятором не является[2].

Результирующая программа компилятора называется *объектной программой* или *объектным кодом*, а исходную программу в этом случае часто называют *исходным кодом*. Файл, в который записана объектная программа, обычно называется *объектным файлом*. Даже в том случае, когда результирующая программа порождается на языке машинных команд, между объектной программой (объектным файлом) и исполняемой программой (исполняемым файлом) есть существенная разница. Порожденная компилятором программа не может непосредственно выполняться на компьютере (более подробно об этом рассказано в главе «Современные системы программирования»).

Само слово «компилятор» происходит от английского термина «compiler» («составитель», «компоновщик»). Термин, вероятно, обязан своему происхождению способности компиляторов составлять объектную программу из фрагментов машинных кодов, соответствующим синтаксическим конструкциям исходной программы (то, как это происходит, описано в главе «Генерация и оптимизация кода»).

Результирующая программа, созданная компилятором, строится на языке машинных кодов или ассемблера, то есть на языках, которые обязательно ориентированы на определенную вычислительную систему. Следовательно, такая результирующая программа всегда предназначена для выполнения на вычислительной системе с определенной архитектурой.

ПРИМЕЧАНИЕ

> Следует упомянуть, что в современных системах программирования существуют компиляторы, в которых результирующая программа создается не на языке машинных команд и не на языке ассемблера, а на некотором промежуточном языке. Сам по себе этот промежуточный язык не может непосредственно исполняться на компьютере, а требует специального промежуточного интерпретатора для выполнения написанных на нем программ. Хотя в данном случае термин «транслятор» был бы, наверное, более правильным, в литературе употребляется понятие «компилятор», поскольку промежуточный язык является языком низкого уровня, будучи родственным машинным командам и языкам ассемблера.

Вычислительная система, на которой должна выполняться результирующая (объектная) программа, созданная компилятором, называется *целевой вычислительной системой*.

---

[2] В некоторых литературных источниках эти два понятия не разделяют между собой, хотя разница между ними все-таки существует.

В понятие целевой вычислительной системы входит не только архитектура аппаратных средств компьютера, но и операционная система (ОС), а зачастую также и набор динамически подключаемых библиотек, которые необходимы для выполнения объектной программы. При этом следует помнить, что объектная программа ориентирована на целевую вычислительную систему, но не может быть непосредственно выполнена на ней без дополнительной обработки.

Целевая вычислительная система не всегда является той же вычислительной системой, на которой работает сам компилятор. Часто они совпадают, но бывает так, что компилятор работает под управлением вычислительной системы одного типа, а строит объектные программы, предназначенные для выполнения на вычислительных системах совсем другого типа.

ПРИМЕЧАНИЕ

> Здесь есть один «подводный камень», связанный с терминологической путаницей. Термины «объектный код» и «объектный файл» возникли достаточно давно. Однако сейчас появилось понятие «объектно-ориентированное программирование», где под термином «объект» подразумевается совершенно иное, нежели в «объектном коде». Следует помнить, что «объектный код» никакого отношения к «объекту» с точки зрения объектно-ориентированного программирования не имеет! И хотя в результате компиляции программ, написанных на объектно-ориентированных языках, тоже получаются объектный код и объектные файлы, это никак не связывает между собой эти различные по смыслу термины.

Компиляторы, безусловно, самый распространенный вид трансляторов (многие считают их вообще единственным видом трансляторов, хотя это и не так). Они имеют самое широкое практическое применение, которым обязаны широкому распространению всевозможных языков программирования. Далее всегда будем говорить о компиляторах, подразумевая, что результирующая программа порождается на языке машинных кодов или языке ассемблера (если это не так, то это будет специально указываться отдельно).

Естественно, трансляторы и компиляторы, как и все прочие программы, разрабатывают люди — обычно это группа разработчиков. В принципе, они могли бы создавать его непосредственно на языке машинных команд, однако объем кода и данных современных компиляторов таков, что их создание на языке машинных команд практически невозможно в разумные сроки при разумных трудозатратах. Поэтому практически все современные компиляторы также создаются с помощью компиляторов (чаще всего в этой роли выступают предыдущие версии компиляторов той же фирмы-производителя). И в этом качестве компилятор сам является результирующей программой для другого компилятора, которая ничем не отличается от всех прочих порождаемых результирующих программ[3].

---

[3] Здесь возникает извечный вопрос «о курице и яйце». Конечно, в первом поколении самые первые компиляторы писались непосредственно на машинных командах, но потом, с появлением компиляторов, от этой практики отошли. Даже самые

Определение интерпретатора. Разница между интерпретаторами и трансляторами

Кроме схожих между собой понятий «транслятор» и «компилятор» существует принципиально отличное от них понятие интерпретатора.

Интерпретатор — это программа, которая воспринимает исходную программу на входном (исходном) языке и выполняет ее.

Интерпретатор, также как и транслятор, анализирует текст исходной программы. Однако он не порождает результирующую программу, а сразу же выполняет исходную программу в соответствии с ее смыслом, заданным семантикой входного языка. Таким образом, результатом работы интерпретатора будет результат, определенный смыслом исходной программы, в том случае, если эта программа синтаксически и семантически правильная с точки зрения входного языка, или сообщение об ошибке в противном случае.

ВНИМАНИЕ

> В отличие от трансляторов, интерпретаторы не порождают результирующую программу — в этом принципиальная разница между ними.

Чтобы исполнить исходную программу, интерпретатор так или иначе должен преобразовать ее в язык машинных кодов, поскольку иначе выполнение программ на компьютере невозможно. Он, конечно же, делает это, однако эти машинные коды недоступны — их не видит пользователь интерпретатора. Машинные коды порождаются интерпретатором, исполняются и уничтожаются по мере надобности — так, как того требует конкретная реализация интерпретатора. Пользователь же видит только результат выполнения этих кодов — то есть результат выполнения исходной программы (требование об эквивалентности исходной программы и порожденных машинных кодов и в этом случае, безусловно, также должно выполняться).

Более подробно вопросы, связанные с реализацией интерпретаторов и их отличием от компиляторов, рассмотрены далее в соответствующем разделе.

Этапы компиляции. Общая схема работы компилятора

На рис. 2.1 представлена общая схема работы компилятора. Из нее видно, что в целом процесс компиляции состоит из двух основных этапов — анализа и синтеза.

На этапе анализа выполняется распознавание текста исходной программы, создание и заполнение таблиц идентификаторов. Результатом его работы служит некое внутреннее представление программы, понятное компилятору.

На этапе синтеза на основании внутреннего представления программы и информации, содержащейся в таблице идентификаторов, порождается текст результирующей программы. Результатом этого этапа является объектный код.

ответственные части компиляторов создаются, как минимум, с применением языка ассемблера — а он тоже обрабатывается компилятором.

Кроме того, в составе компилятора присутствует часть, ответственная за анализ и исправление ошибок, которая при наличии ошибки в тексте исходной программы должна максимально полно информировать пользователя о типе ошибки и месте ее возникновения. В лучшем случае компилятор может предложить пользователю вариант исправления ошибки.

Эти этапы, в свою очередь, состоят из более мелких этапов, называемых фазами компиляции. Состав фаз компиляции на рис. 2.1 приведен в самом общем виде, их конкретная реализация и процесс взаимодействия могут, конечно, различаться в зависимости от версии компилятора. Однако в том или ином виде все представленные фазы практически всегда присутствуют в каждом конкретном компиляторе [5, 15, 21, 33, 34, 58].

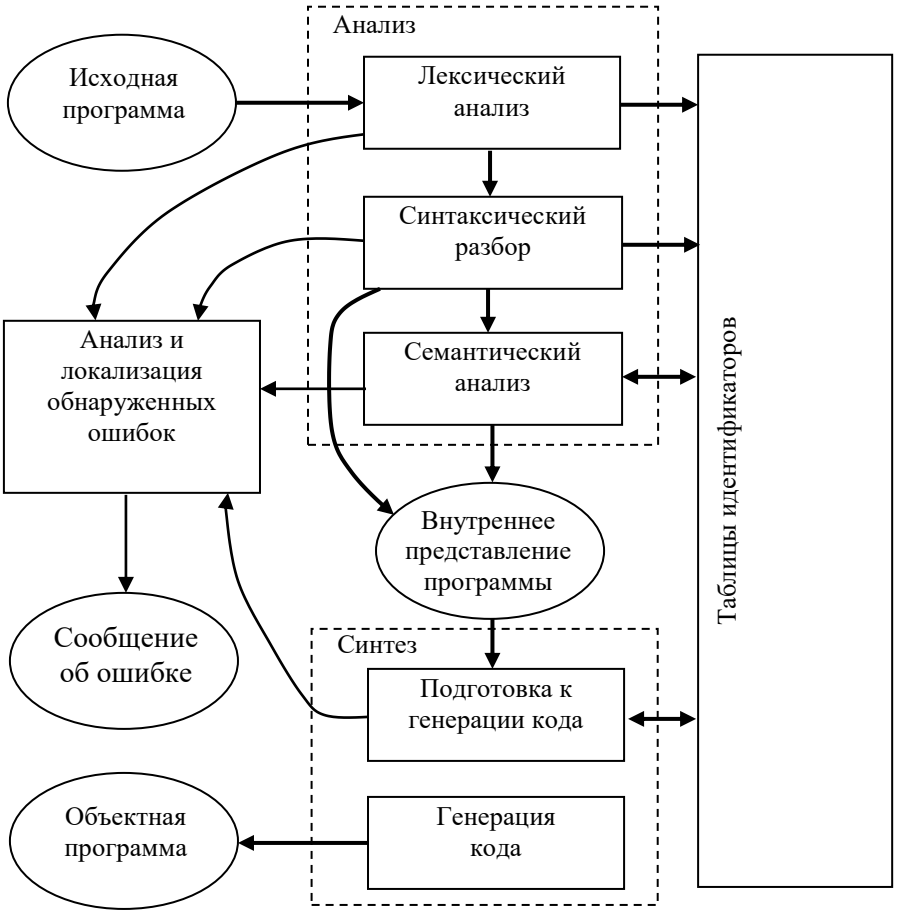

Рис. 2.1. Общая схема работы компилятора

Компилятор в целом с точки зрения теории формальных языков выступает в «двух ипостасях», выполняет две основные функции.

Во-первых, он является распознавателем для языка исходной программы. То есть он должен получить на вход цепочку символов входного языка, проверить ее принадлежность языку и, более того, выявить правила, по которым эта цепочка построена (поскольку сам ответ на вопрос о принадлежности «да» или «нет» представляет мало интереса). Интересно, что генератором цепочек входного языка выступает пользователь — автор исходной программы.

Во-вторых, компилятор является генератором для языка результирующей программы. Он должен построить на выходе цепочку выходного языка по определенным правилам, предполагаемым языком машинных команд или языком ассемблера. В случае машинных команд распознавателем этой цепочки будет выступать целевая вычислительная система, под которую создается результирующая программа.

Далее дается перечень основных фаз (частей) компиляции и краткое описание их функций. Более подробная информация дана в главах учебного пособия, соответствующих этим фазам.

Лексический анализ (сканер) — это часть компилятора, которая читает литеры программы на исходном языке и строит из них слова (лексемы) исходного языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического разбора. С теоретической точки зрения лексический анализатор не является обязательной частью компилятора. Однако существует причины, которые определяют его присутствие практически во всех компиляторах (более подробно они описаны в главе «Лексические анализаторы»).

Синтаксический разбор — это основная часть компилятора на этапе анализа. Она выполняет выделение синтаксических конструкций в тексте исходной программы, обработанном лексическим анализатором. На этой же фазе компиляции проверяется синтаксическая правильность программы. Синтаксический разбор играет главную роль — роль распознавателя текста входного языка программирования (этот процесс описан в главе «Синтаксические анализаторы»).

Семантический анализ — это часть компилятора, проверяющая правильность текста исходной программы с точки зрения семантики входного языка. Кроме непосредственно проверки, семантический анализ должен выполнять преобразования текста, требуемые семантикой входного языка (например, такие, как добавление функций неявного преобразования типов). В различных реализациях компиляторов семантический анализ может частично входить в фазу синтаксического разбора, частично — в фазу подготовки к генерации кода (в данной книге семантический анализ более подробно рассмотрен в главе «Генерация и оптимизация кода»).

Подготовка к генерации кода — это фаза, на которой компилятором выполняются предварительные действия, необходимые для с синтеза результирующей программы, но не ведущие к порождению текста на выходном языке. Обычно в эту фазу входят действия, связанные с идентификацией элементов языка, распределением памяти и т. п. (они рассмотрены в главе «Генерация и оптимизация кода»).

Генерация кода — это фаза, непосредственно связанная с порождением текста результирующей программы. Это основная фаза на этапе синтеза результирующей программы. Кроме непосредственно порождения текста результирующей программы,

генерация обычно включает в себя также оптимизацию — процесс, связанный с обработкой уже порожденного текста. Иногда оптимизацию выделяют в отдельную фазу компиляции, так как она оказывает существенное влияние на качество и эффективность результирующей программы (подробности вы найдете в главе «Генерация и оптимизация кода»).

Таблицы идентификаторов (иногда — «таблицы символов») — это специальным образом организованные структуры данных, служащие для хранения информации об элементах исходной программы, которые затем используются для порождения текста результирующей программы. В конкретной реализации компилятора может быть как одна, так и несколько таблиц идентификаторов. Элементами исходной программы, информацию о которых необходимо хранить в процессе компиляции, являются переменные, константы, функции и т. п. — конкретный состав этих элементов зависит от входного языка. Термин «таблицы» вовсе не предполагает, что это хранилище данных должно быть организовано именно в виде таблиц или массивов информации — возможные методы их организации подробно рассмотрены далее, в разделе «Таблицы идентификаторов. Организация таблиц идентификаторов».

Представленное на рис. 2.1 деление процесса компиляции на фазы служит скорее методическим целям и на практике может не соблюдаться столь строго. Далее в главах и разделах этого учебного пособия рассматриваются различные варианты технической организации представленных фаз компиляции. При этом указано, как они могут быть связаны между собой. Здесь рассмотрим только общие аспекты такого рода взаимосвязи.

Во-первых, на фазе лексического анализа лексемы выделяются из текста входной программы постольку, поскольку они необходимы для фазы синтаксического разбора. Во-вторых, как будет показано ниже, синтаксический разбор и генерация кода могут выполняться одновременно. Таким образом, эти три фазы компиляции могут работать комбинированно, а вместе с ними может выполняться и подготовка к генерации кода. Далее рассмотрены технические вопросы реализации основных фаз компиляции, которые тесно связаны с понятием *прохода*.

Таблицы идентификаторов. Организация таблиц идентификаторов

Назначение и особенности построения таблиц идентификаторов

Проверка правильности семантики и генерация кода требуют знания характеристик переменных, констант, функций и других элементов, встречающихся в программе на исходном языке. Все эти элементы в исходной программе, как правило, обозначаются идентификаторами. Выделение идентификаторов и других элементов исходной программы происходит на фазе лексического анализа. Их характеристики определяются на фазах синтаксического разбора, семантического анализа и подготовки к генерации кода. Состав возможных характеристик и методы их определения зависят от семантики входного языка.

В любом случае компилятор должен иметь возможность хранить все найденные идентификаторы и связанные с ними характеристики в течение всего процесса компиляции, чтобы иметь возможность использовать их на различных фазах

компиляции. Для этой цели, как было сказано выше, в компиляторах используются специальные хранилища данных, называемые *таблицами символов* или *таблицами идентификаторов*.

Любая таблица идентификаторов состоит из набора полей, количество которых равно числу различных идентификаторов, найденных в исходной программе. Каждое поле содержит в себе полную информацию о данном элементе таблицы. Компилятор может работать с одной или несколькими таблицам идентификаторов — их количество зависит от реализации компилятора. Например, можно организовывать различные таблицы идентификаторов для различных модулей исходной программы или для различных типов элементов входного языка.

Состав информации, хранимой в таблице идентификаторов для каждого элемента исходной программы, зависит от семантики входного языка и типа элемента. Примеры такой информации указаны в [5, 18, 34, 42, 58, 59]. Конкретное наполнение таблиц идентификаторов зависит от реализации компилятора. Не вся информация, хранимая в таблице идентификаторов, заполняется компилятором сразу — он может несколько раз выполнять обращение к данным в таблице идентификаторов на различных фазах компиляции. Например, имена переменных могут быть выделены на фазе лексического анализа, типы данных для переменных — на фазе синтаксического разбора, а область памяти связывается с переменной только на фазе подготовки к генерации кода. На различных фазах компиляции компилятор вынужден многократно обращаться к таблице для поиска информации и записи новых данных.

ВНИМАНИЕ

> Компилятору приходится выполнять поиск необходимого элемента в таблице идентификаторов по имени чаще, чем помещать новый элемент в таблицу, потому что каждый идентификатор может быть описан только один раз, а использован — несколько раз.

Отсюда можно сделать вывод, что таблицы идентификаторов должны быть организованы таким образом, чтобы компилятор имел возможность максимально быстрого поиска нужного ему элемента [5, 42].

Простейшие методы построения таблиц идентификаторов

Простейший способ организации таблицы состоит в том, чтобы добавлять элементы в порядке их поступления. Тогда таблица идентификаторов будет представлять собой неупорядоченный массив информации, каждая ячейка которого будет содержать данные о соответствующем элементе таблицы.

Поиск нужного элемента в таблице будет в этом случае заключаться в последовательном сравнении искомого элемента с каждым элементом таблицы, пока не будет найден подходящий. Тогда, если за единицу времени принять время, затрачиваемое компилятором на сравнение двух элементов (как правило, это сравнение двух строк), то для таблицы, содержащей N элементов, в среднем будет выполнено N/2 сравнений [11].

Заполнение такой таблицы будет происходить элементарно просто — добавлением нового элемента в ее конец, и время, требуемое на добавление элемента ($T_з$) не будет зависеть от числа элементов в таблице N. Но если N велико, то поиск потребует значительных затрат времени. Время поиска ($T_п$) в такой таблице можно оценить как $T_п = O(N)$. Поскольку поиск в таблице идентификаторов является наиболее часто выполняемой компилятором операцией, а количество различных идентификаторов в реальной исходной программе достаточно велико (от нескольких сотен до нескольких тысяч элементов), то такой способ организации таблиц идентификаторов является неэффективным.

Поиск может быть выполнен более эффективно, если элементы таблицы упорядочены (отсортированы) согласно некоторому естественному порядку. Поскольку поиск осуществляется по имени идентификатора, наиболее естественным решением будет расположить элементы таблицы в прямом или обратном алфавитном порядке. Эффективным методом поиска в упорядоченном списке из N элементов является *бинарный* или *логарифмический поиск*.

Алгоритм логарифмического поиск заключается в следующем: искомый символ сравнивается с элементом $(N + 1)/2$ в середине таблицы. Если этот элемент не является искомым, то мы должны просмотреть только блок элементов, пронумерованных от 1 до $(N + 1)/2–1$, или блок элементов от $(N + 1)/2 + 1$ до N в зависимости от того, меньше или больше искомый элемент того, с которым его сравнили. Затем процесс повторяется над нужным блоком в два раза меньшего размера. Так продолжается до тех пор, пока либо искомый элемент будет найден, либо алгоритм дойдет до очередного блока, содержащего один или два элемента.

Так как на каждом шаге число элементов, которые могут содержать искомый элемент, сокращается в 2 раза, максимальное число сравнений равно $1 + \log_2(N)$. Тогда время поиска элемента в таблице идентификаторов можно оценить как $T_п = O(\log_2 N)$. Для сравнения: при $N = 128$ бинарный поиск потребует максимум 8 сравнений, а поиск в неупорядоченной таблице — в среднем 64 сравнения.

Этот метод называют «бинарным поиском», поскольку на каждом шаге объем рассматриваемой информации сокращается в два раза, а «логарифмическим» — поскольку время, затрачиваемое на поиск нужного элемента в массиве, имеет логарифмическую зависимость от общего количества элементов в нем.

Недостатком логарифмического поиска является требование упорядочивания таблицы идентификаторов. Так как массив информации, в котором выполняется поиск, должен быть упорядочен, то время его заполнения уже будет зависеть от числа элементов в массиве. Таблица идентификаторов зачастую просматривается компилятором еще до того, как она заполнена полностью, поэтому требуется, чтобы условие упорядоченности выполнялось на всех этапах обращения к ней. Следовательно, для построения такой таблицы можно пользоваться только алгоритмом прямого упорядоченного включения элементов.

При добавлении каждого нового элемента в таблицу сначала надо определить место, куда поместить новый элемент, а потом выполнить перенос части информации в таблице, если элемент добавляется не в ее конец. Если пользоваться стандартными алгоритмами, применяемыми для организации упорядоченных массивов данных [11], то среднее время, необходимое на помещение всех элементов в таблицу, можно оценить следующим образом:

$$T_з = O(N*\log_2 N) + k*O(N^2)$$

Здесь k — некоторый коэффициент, отражающий соотношение между временами, затрачиваемыми на выполнение операции сравнения и операции переноса данных.

В итоге при организации логарифмического поиска в таблице идентификаторов мы добиваемся существенного сокращения времени поиска нужного элемента за счет увеличения времени на помещение нового элемента в таблицу. Поскольку добавление новых элементов в таблицу идентификаторов происходит существенно реже[4], чем обращение к ним, этот метод следует признать более эффективным, чем метод организации неупорядоченной таблицы.

Построение таблиц идентификаторов по методу бинарного дерева

Можно сократить время поиска искомого элемента в таблице идентификаторов, не увеличивая значительно время, необходимое на ее заполнение. Для этого надо отказаться от организации таблицы в виде непрерывного массива данных.

Существует метод построения таблиц, при котором таблица имеет форму бинарного дерева. Каждый узел дерева представляет собой элемент таблицы, причем корневой узел является первым элементом, встреченным при заполнении таблицы. Дерево называется бинарным, так как каждая вершина в нем может иметь не более двух ветвей. Для определенности будем называть две ветви «правая» и «левая».

Рассмотрим алгоритм заполнения бинарного дерева. Будем считать, что алгоритм работает с потоком входных данных, содержащим идентификаторы (в компиляторе этот поток данных порождается в процессе разбора текста исходной программы). Первый идентификатор, как уже было сказано, помещается в вершину дерева. Все дальнейшие идентификаторы попадают в дерево по следующему алгоритму:

Шаг 1. Выбрать очередной идентификатор из входного потока данных. Если очередного идентификатора нет, то построение дерева закончено.

Шаг 2. Сделать текущим узлом дерева корневую вершину.

Шаг 3. Сравнить очередной идентификатор с идентификатором, содержащемся в текущем узле дерева.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, если равен — сообщить об ошибке и прекратить выполнение алгоритма (двух одинаковых идентификаторов быть не должно!), иначе — перейти к шагу 7.

Шаг 5. Если у текущего узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 6.

Шаг 6. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.

---

[4] Как минимум при добавлении нового идентификатора в таблицу компилятор должен проверить, существует или нет там такой идентификатор, так как в большинстве языков программирования ни один идентификатор не может быть описан более одного раза. Следовательно, каждая операция добавления нового элемента влечет, как правило, не менее одной операции поиска.

Шаг 7. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 8.

Шаг 8. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину правой вершиной текущего узла и вернуться к шагу 1.

Рассмотрим в качестве примера последовательность идентификаторов `GA`, `D1`, `M22`, `E`, `A12`, `BC`, `F`. На рис. 2.2 проиллюстрирован весь процесс построения бинарного дерева для этой последовательности идентификаторов.

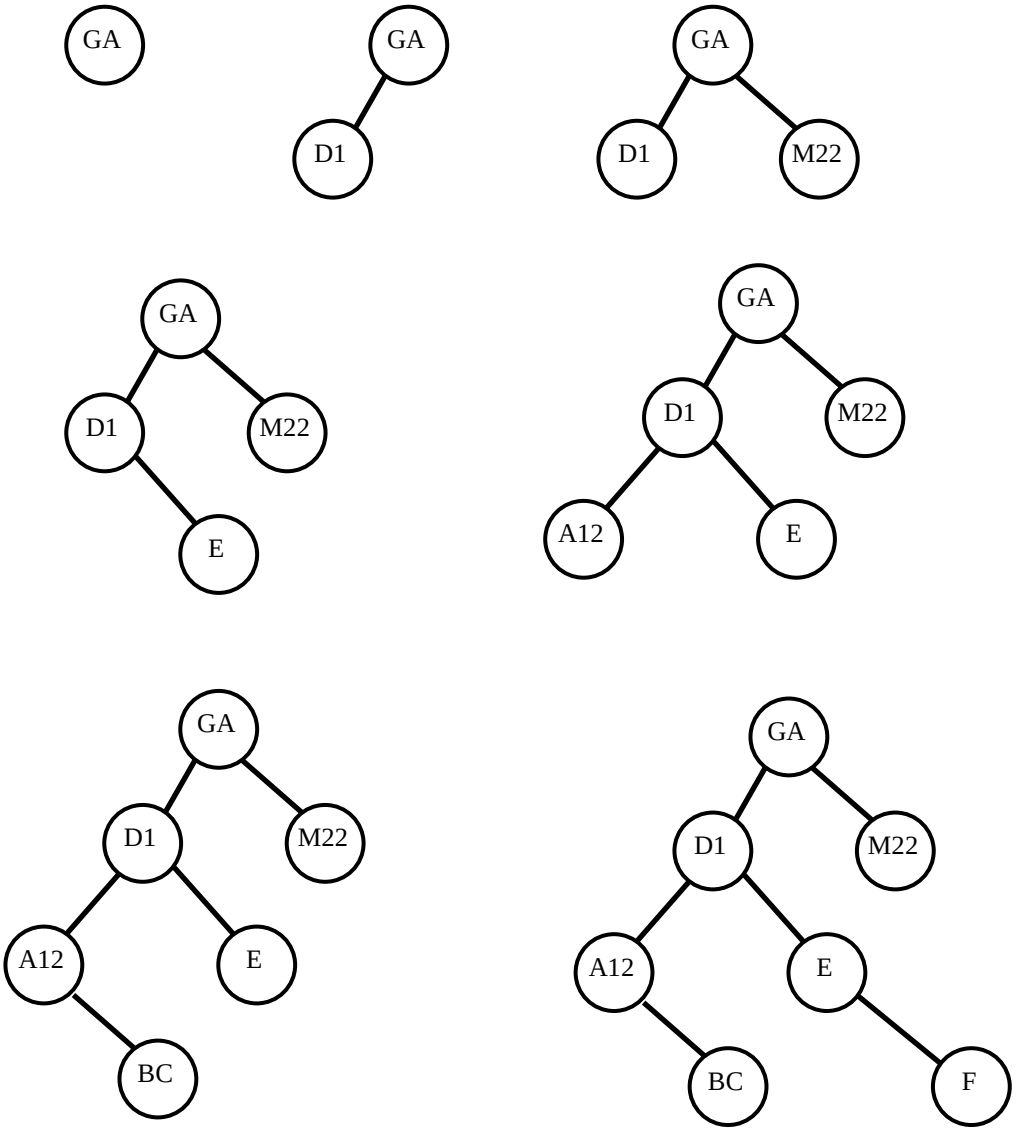

Рис. 2.2. Заполнение бинарного дерева для последовательности идентификаторов идентификаторов GA, D1, M22, E, A12, BC, F

Поиск нужного элемента в дереве выполняется по алгоритму, схожему с алгоритмом заполнения дерева:

Шаг 1. Сделать текущим узлом дерева корневую вершину.

Шаг 2. Сравнить искомый идентификатор с идентификатором, содержащемся в текущем узле дерева.

Шаг 3. Если идентификаторы совпадают, то искомый идентификатор найден, алгоритм завершается, иначе надо перейти к шагу 4.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, иначе — перейти к шагу 6.

Шаг 5. Если у текущего узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершается.

Шаг 7. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершается.

Например, произведем поиск в дереве, изображенном на рис. 2.2, идентификатора `A12`. Берем корневую вершину (она становится текущим узлом), сравниваем идентификаторы `GA` и `A12`. Искомый идентификатор меньше — текущим узлом становится левая вершина `D1`. Опять сравниваем идентификаторы. Искомый идентификатор меньше — текущим узлом становится левая вершина `A12`. При следующем сравнении искомый идентификатор найден.

Если искать отсутствующий идентификатор — например, `A11` — то поиск опять пойдет от корневой вершины. Сравниваем идентификаторы `GA` и `A11`. Искомый идентификатор меньше — текущим узлом становится левая вершина `D1`. Опять сравниваем идентификаторы. Искомый идентификатор меньше — текущим узлом становится левая вершина `A12`. Искомый идентификатор меньше, но левая вершина у узла `A12` отсутствует, поэтому в данном случае искомый идентификатор не найден.

Для данного метода число требуемых сравнений и форма получившегося дерева зависят от того порядка, в котором поступают идентификаторы. Например, если в рассмотренном выше примере вместо последовательности идентификаторов `GA`, `D1`, `M22`, `E`, `A12`, `BC`, `F` взять последовательность `A12`, `GA`, `D1`, `M22`, `E`, `BC`, `F`, то полученное дерево будет иметь иной вид. А если в качестве примера взять последовательность идентификаторов `A12`, `BC`, `D1`, `E`, `F`, `GA`, `M22`, то дерево выродится в упорядоченный однонаправленный связный список. Эта особенность является недостатком данного метода организации таблиц идентификаторов. Другим недостатком является необходимость работы с динамическим выделением памяти при построении дерева.

Если предположить, что последовательность идентификаторов в исходной программе является статистически неупорядоченной (что в целом соответствует действительности), то можно считать, что построенное бинарное дерево будет невырожденным. Тогда среднее время на заполнение дерева ($T_з$) и на поиск элемента в нем ($T_п$) можно оценить следующим образом [4 т.2]:

$$T_з = O(N*\log_2 N)$$

$$T_п = O(\log_2 N)$$

В целом метод бинарного дерева является довольно удачным механизмом для организации таблиц идентификаторов. Он нашел свое применение в ряде компиляторов. Иногда компиляторы строят несколько различных деревьев для идентификаторов разных типов и разной длины [18, 58, 59].

Хеш-функции и хеш-адресация

Принципы работы хеш-функций

Логарифмическая зависимость времени поиска и времени заполнения таблицы идентификаторов — это самый хороший результат, которого можно достичь за счет применения различных методов организации таблиц. Однако в реальных исходных программах количество идентификаторов столь велико, что даже логарифмическую зависимость времени поиска от их числа нельзя признать удовлетворительной. Необходимы более эффективные методы поиска информации в таблице идентификаторов.

Лучших результатов можно достичь, если применить методы, связанные с использованием хеш-функций и хеш-адресации.

Хеш-функцией F называется некоторое отображение множества входных элементов **R** на множество целых неотрицательных чисел **Z**: F(r) = n, r∈**R**, n∈**Z**. Множество допустимых входных элементов **R** называется областью определения хеш-функции. Множеством значений хеш-функции F называется подмножество **M** из множества целых неотрицательных чисел **Z**: **M**⊆**Z**, содержащее все возможные значения, возвращаемые функцией F: ∀r∈**R**: F(r)∈**M** и ∀m∈**M**: ∃r∈**R**: F(r)=m. Процесс отображения области определения хеш-функции на множество значений называется «хешированием».

Сам термин «хеш-функция» происходит от английского термина «hash function» (hash — «мешать», «смешивать», «путать»). Вместо термина «хеширование» иногда используются термины «рандомизация», «переупорядочивание».

При работе с таблицей идентификаторов хеш-функция должна выполнять отображение имен идентификаторов на множество целых неотрицательных чисел. Областью определения хеш-функции будет множество всех возможных имен идентификаторов.

Хеш-адресация заключается в использовании значения, возвращаемого хеш-функцией, в качестве адреса ячейки из некоторого массива данных. Тогда размер массива данных должен соответствовать области значений используемой хеш-функции. Следовательно, в реальном компиляторе область значений хеш-функции никак не должна превышать размер доступного адресного пространства компьютера.

Метод организации таблиц идентификаторов, основанный на использовании хеш-адресации, заключается в размещении каждого элемента таблицы в ячейке, адрес которой возвращает хеш-функция, вычисленная для этого элемента. Тогда в идеальном случае для размещения любого элемента в таблице идентификаторов

достаточно только вычислить его хеш-функцию и обратиться к нужной ячейке массива данных. Для поиска элемента в таблице необходимо вычислить хеш-функцию для искомого элемента и проверить, не является ли заданная ею ячейка массива пустой: если она не пуста — элемент найден, если пуста — не найден. Первоначально таблица идентификаторов должна быть заполнена информацией, которая позволила бы говорить о том, что все ее ячейки являются пустыми.

На рис. 2.3 проиллюстрирован метод организации таблиц идентификаторов с использованием хеш-адресации. Трем различным идентификаторам A_1, A_2, A_3 соответствуют на рисунке три значения хеш-функции n_1, n_2, n_3. В ячейки, адресуемые n_1, n_2, n_3, помещается информация об идентификаторах A_1, A_2, A_3. При поиске идентификатора A_3 вычисляется значение адреса n_3 и выбираются данные из соответствующей ячейки таблицы.

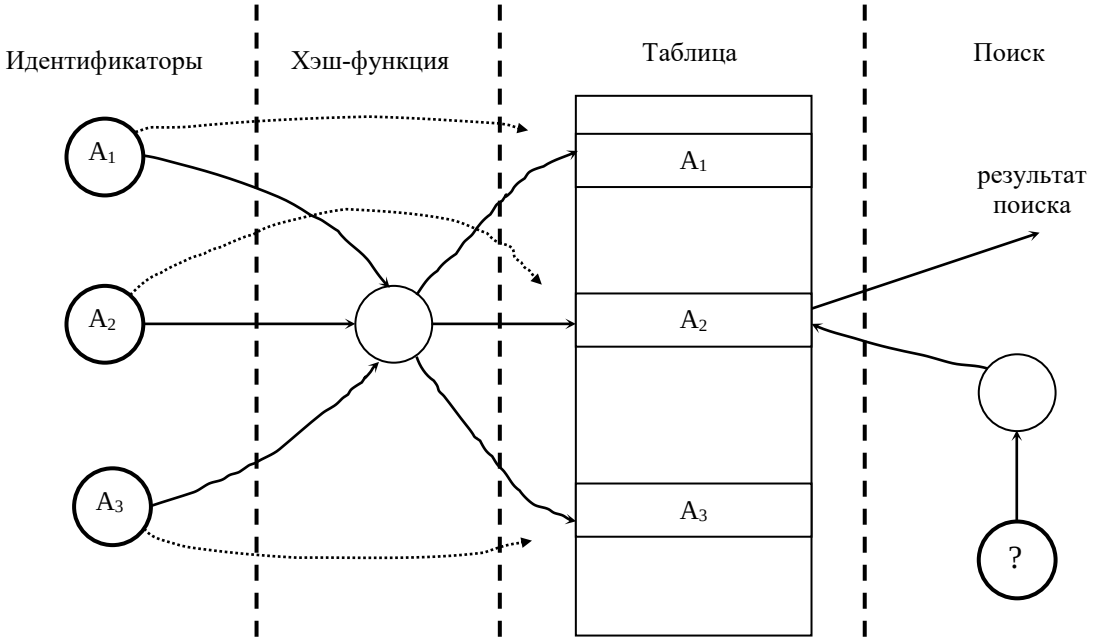

Рис. 2.3. Организация таблицы идентификаторов с использованием хеш-адресации

Этот метод весьма эффективен, поскольку как время размещения элемента в таблице, так и время его поиска определяются только временем, затрачиваемым на вычисление хеш-функции, которое в общем случае несопоставимо меньше времени, необходимого для выполнения многократных сравнений элементов таблицы.

Метод имеет два очевидных недостатка. Первый из них — неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен соответствовать области значений хеш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше. Второй недостаток — необходимость соответствующего разумного выбора хеш-функции. Этому существенному вопросу посвящены следующие два подпункта.

Построение таблиц идентификаторов на основе хеш-функций

Существуют различные варианты хеш-функций. Получение результата хеш-функции — «хеширование» — обычно достигается за счет выполнения над цепочкой символов некоторых простых арифметических и логических операций. Самой простой хеш-функцией для символа является код внутреннего представления в компьютере литеры символа. Эту хеш-функцию можно использовать и для цепочки символов, выбирая первый символ в цепочке. Так, если двоичное ASCII представление символа `A` есть двоичный код 00100001_2, то результатом хеширования идентификатора `ATable` будет код 00100001_2.

Хеш-функция, предложенная выше, очевидно не удовлетворительна: при использовании такой хеш-функции возникнет проблема — двум различным идентификаторам, начинающимся с одной и той же буквы, будет соответствовать одно и то же значение хеш-функции. Тогда при хеш-адресации в одну ячейку таблицы идентификаторов по одному и тому же адресу должны быть помещены два различных идентификатора, что явно невозможно. Такая ситуация, когда двум или более идентификаторам соответствует одно и то же значение хеш-функции, называется *коллизией*.

Естественно, что хеш-функция, допускающая коллизии, не может быть напрямую использована для хеш-адресации в таблице идентификаторов. Причем достаточно получить хотя бы один случай коллизии на всем множестве идентификаторов, чтобы такой хеш-функцией нельзя было пользоваться непосредственно. Но в примере взята самая элементарная хеш-функция. А возможно ли построить хеш-функцию, которая бы полностью исключала возникновение коллизий?

Очевидно, что для полного исключения коллизий хеш-функция должна быть взаимно однозначной: каждому элементу из области определения хеш-функции должно соответствовать одно значение из ее множества значений, и каждому значению из множества значений этой функции должен соответствовать только один элемент из области ее определения. Тогда любым двум произвольным элементам из области определения хеш-функции будут всегда соответствовать два различных ее значения. Теоретически для идентификаторов такую хеш-функцию построить можно, так как и область определения хеш-функции (все возможные имена идентификаторов), и область ее значений (целые неотрицательные числа) являются бесконечными счетными множествами. Теоретически можно организовать взаимно однозначное отображение одного счетного множества на другое[5].

Практически существует ограничение, делающее создание взаимно однозначной хеш-функции для идентификаторов невозможным. Дело в том, что в реальности область значений любой хеш-функции ограничена размером доступного адресного пространства компьютера. При организации хеш-адресации значение, используемое в качестве адреса таблицы идентификаторов, не может выходить за пределы,

---

[5] Элементарным примером такого отображения для строки символов является сопоставление ей двоичного числа, полученного путем конкатенации кодов символов, входящих в строку. Фактически, сама строка тогда будет выступать адресом при хеш-адресации. Практическая ценность такого отображения весьма сомнительна.

заданные разрядностью адреса компьютера[6]. Множество адресов любого компьютера с традиционной архитектурой может быть велико, но всегда конечно, то есть ограничено. Организовать взаимно однозначное отображение бесконечного множества на конечное даже теоретически невозможно. Можно учесть, что длина принимаемой во внимание части имени идентификатора в реальных компиляторах также практически ограничена — обычно она лежит в пределах от 32 до 128 символов (то есть и область определения хеш-функции конечна). Но и тогда количество элементов в конечном множестве, составляющем область определения функции, будет превышать их количество в конечном множестве области значений функции (количество всех возможных идентификаторов все равно больше количества допустимых адресов в современных компьютерах). Таким образом, создать взаимно однозначную хеш-функцию практически ни в каком варианте невозможно. Следовательно, невозможно избежать возникновения коллизий.

Для решения проблемы коллизии можно использовать много способов. Одним из них является метод *рехеширования* (или расстановки). Согласно этому методу, если для элемента A адрес h(A), вычисленный с помощью хеш-функции h, указывает на уже занятую ячейку, то необходимо вычислить значение функции $n_1 = h_1(A)$ и проверить занятость ячейки по адресу n_1. Если и она занята, то вычисляется значение $h_2(A)$ и так до тех пор, пока либо не будет найдена свободная ячейка, либо очередное значение $h_i(A)$ совпадет с h(A). В последнем случае считается, что таблица идентификаторов заполнена, и места в ней больше нет — выдается информация об ошибке размещения идентификатора в таблице.

Такую таблицу идентификаторов можно организовать по следующему алгоритму размещения элемента:

Шаг 1. Вычислить значение хеш-функции n = h(A) для нового элемента A.

Шаг 2. Если ячейка по адресу n пустая, то поместить в нее элемент A и завершить алгоритм, иначе i := 1 и перейти к шагу 3.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая, то поместить в нее элемент A и завершить алгоритм, иначе перейти к шагу 4.

Шаг 4. Если $n = n_i$, то сообщить об ошибке и завершить алгоритм, иначе i := i + 1 и вернуться к шагу 3.

Тогда поиск элемента A в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1. Вычислить значение хеш-функции n = h(A) для искомого элемента A.

Шаг 2. Если ячейка по адресу n пустая, то элемент не найден, алгоритм завершен, иначе сравнить имя элемента в ячейке n с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершен, иначе i := 1 и перейти к шагу 3.

---

[6] Можно, конечно, организовать адресацию с использованием внешних накопителей для организации виртуальной памяти, но накладные затраты для такой адресации будут весьма велики. И даже в таком варианте адресное пространство никогда не будет бесконечным.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая или $n = n_i$, то элемент не найден и алгоритм завершен, иначе сравнить имя элемента в ячейке n_i с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершен, иначе i := i + 1 и повторить шаг 3.

Алгоритмы размещения и поиска элемента схожи по выполняемым операциям. Поэтому они будут иметь одинаковые оценки времени, необходимого для их выполнения.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм размещает элементы в пустых ячейках таблицы, выбирая их определенным образом. При этом элементы могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хеш-функции, что приведет к возникновению новых, дополнительных коллизий. Таким образом, количество операций, необходимых для поиска или размещения в таблице элемента, зависит от заполненности таблицы.

Для организации таблицы идентификаторов по методу рехеширования необходимо определить все хеш-функции h_i для всех i. Чаще всего функции h_i определяют как некоторые модификации хеш-функции h. Например, самым простым методом вычисления функции $h_i(A)$ является ее организация в виде $h_i(A) = (h(A) + p_i) \bmod N_m$, где p_i — некоторое вычисляемое целое число, а N_m — максимальное значение из области значений хеш-функции h. В свою очередь, самым простым подходом здесь будет положить $p_i = i$. Тогда получаем формулу $h_i(A) = (h(A) + i) \bmod N_m$. В этом случае при совпадении значений хеш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной хеш-функцией h(A).

Этот способ нельзя признать особенно удачным — при совпадении хеш-адресов элементы в таблице начинают группироваться вокруг них, что увеличивает число необходимых сравнений при поиске и размещении. Среднее время поиска элемента в такой таблице в зависимости от числа операций сравнения можно оценить следующим образом [18]:

$$T_п = O((1 – Lf/2)/(1 – Lf))$$

Здесь Lf — (load factor) степень заполненности таблицы идентификаторов — отношение числа занятых ячеек N таблицы к максимально допустимому числу элементов в ней: $Lf = N/N_m$.

Рассмотрим в качестве примера ряд последовательных ячеек таблицы n_1, n_2, n_3, n_4, n_5 и ряд идентификаторов, которые надо разместить в ней: A_1, A_2, A_3, A_4, A_5 при условии, что $h(A_1) = h(A_2) = h(A_5) = n_1$; $h(A_3) = n_2$; $h(A_4) = n_4$. Последовательность размещения идентификаторов в таблице при использовании простейшего метода рехеширования показана на рис. 2.4. В итоге после размещения в таблице для поиска идентификатора A_1 потребуется 1 сравнение, для A_2 — 2 сравнения, для A_3 — 2 сравнения, для A_4 — 1 сравнение и для A_5 — 5 сравнений.

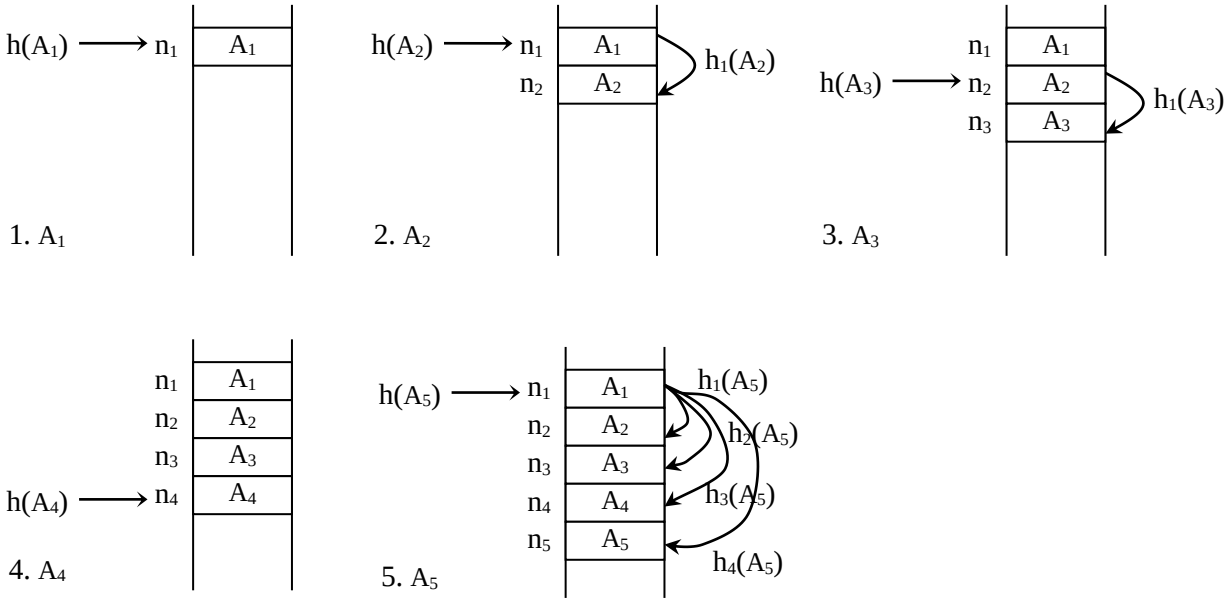

Рис. 2.4. Заполнение таблицы идентификаторов при использовании простейшего рехеширования

Даже такой примитивный метод рехеширования является достаточно эффективным средством организации таблиц идентификаторов при неполном заполнении таблицы. Имея, например, заполненную на 90% таблицу для 1024 идентификаторов, в среднем необходимо выполнить 5.5 сравнений для поиска одного идентификатора, в то время, как даже логарифмический поиск дает в среднем от 9 до 10 сравнений. Сравнительная эффективность метода будет еще выше при росте числа идентификаторов и снижении заполненности таблицы.

Среднее время на помещение одного элемента в таблицу и на поиск элемента в таблице можно снизить, если применить более совершенный метод рехеширования. Одним из таких методов является использование в качестве p_i для функции $h_i(A) = (h(A) + p_i) \bmod N_m$ последовательности псевдослучайных целых чисел p_1, p_2, …, p_k. При хорошем выборе генератора псевдослучайных чисел длина последовательности k будет $k = N_m$. Тогда среднее время поиска одного элемента в таблице можно оценить следующим образом [18]:

$$E_п = O((1/Lf)*\log_2(1 - Lf))$$

Существуют и другие методы организации функций рехеширования $h_i(A)$, основанные на квадратичных вычислениях или, например, на вычислении по формуле: $h_i(A) = (h(A)*i) \bmod N_m$, если N_m — простое число. В целом рехеширование позволяет добиться неплохих результатов для эффективного поиска элемента в таблице (лучших, чем бинарный поиск и бинарное дерево), но эффективность метода сильно зависит от заполненности таблицы идентификаторов и качества используемой хеш-функции — чем реже возникают коллизии, тем выше эффективность метода. Требование неполного заполнения таблицы ведет к неэффективному использованию объема доступной памяти.

Построение таблиц идентификаторов по методу цепочек

Неполное заполнение таблицы идентификаторов при применении хеш-функций ведет к неэффективному использованию объема памяти, доступного компилятору. Причем объем неиспользуемой памяти будет тем выше, чем больше информации

хранится для каждого идентификатора. Этого недостатка можно избежать, если дополнить таблицу идентификаторов некоторой промежуточной хеш-таблицей.

В ячейках хеш-таблицы может храниться либо пустое значение, либо указатель на некоторую область памяти из таблицы идентификаторов. Тогда хеш-функция вычисляет адрес, по которому происходит обращение сначала к хеш-таблице, а потом уже через нее — к самой таблице идентификаторов. Если соответствующая ячейка таблицы идентификаторов пуста, то ячейка хеш-таблицы будет содержать пустое значение. Тогда вовсе не обязательно иметь в самой таблице идентификаторов ячейку для каждого возможного значения хеш-функции — таблицу можно сделать динамической так, чтобы ее объем рос по мере заполнения.

Такой подход позволяет добиться двух положительных результатов: во-первых, нет необходимости заполнять пустыми значениями таблицу идентификаторов — это можно сделать только для хеш-таблицы; во-вторых, каждому идентификатору будет соответствовать строго одна ячейка в таблице идентификаторов. Пустые ячейки в таком случае будут только в хеш-таблице, и объем неиспользуемой памяти не будет зависеть от объема информации, хранимой для каждого идентификатора — для каждого значения хеш-функции будет расходоваться только память, необходимая для хранения одного указателя на основную таблицу идентификаторов.

На основе этой схемы можно реализовать еще один способ организации таблиц идентификаторов с помощью хеш-функций, называемый «*метод цепочек*». Для метода цепочек в таблицу идентификаторов для каждого элемента добавляется еще одно поле, в котором может содержаться ссылка на любой элемент таблицы. Первоначально это поле всегда пустое (никуда не указывает). Также для этого метода необходимо иметь одну специальную переменную, которая всегда указывает на первую свободную ячейку основной таблицы идентификаторов (первоначально — указывает на начало таблицы).

Метод цепочек работает по следующему алгоритму:

Шаг 1. Во все ячейки хеш-таблицы поместить пустое значение, таблица идентификаторов пуста, переменная `FreePtr` (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов; i := 1.

Шаг 2. Вычислить значение хеш-функции n_i для нового элемента A_i. Если ячейка хеш-таблицы по адресу n_i пустая, то поместить в нее значение переменной `FreePtr` и перейти к шагу 5; иначе перейти к шагу 3.

Шаг 3. Положить j := 1, выбрать из хеш-таблицы адрес ячейки таблицы идентификаторов m_j и перейти к шагу 4.

Шаг 4. Для ячейки таблицы идентификаторов по адресу m_j проверить значение поля ссылки. Если оно пустое, то записать в него адрес из переменной `FreePtr` и перейти к шагу 5; иначе j := j + 1, выбрать из поля ссылки адрес m_j и повторить шаг 4.

Шаг 5. Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента A_i (поле ссылки должно быть пустым), в переменную `FreePtr` поместить адрес за концом добавленной ячейки. Если больше нет идентификаторов, которые надо разместить в таблице, то выполнение алгоритма закончено, иначе i := i + 1 и перейти к шагу 2.

Поиск элемента в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1. Вычислить значение хеш-функции n для искомого элемента A. Если ячейка хеш-таблицы по адресу n пустая, то элемент не найден и алгоритм завершен, иначе положить j := 1, выбрать из хеш-таблицы адрес ячейки таблицы идентификаторов m_j.

Шаг 2. Сравнить имя элемента в ячейке таблицы идентификаторов по адресу m_j с именем искомого элемента A. Если они совпадают, то искомый элемент найден и алгоритм завершен, иначе перейти к шагу 3.

Шаг 3. Проверить значение поля ссылки в ячейке таблицы идентификаторов по адресу m_j. Если оно пустое, то искомый элемент не найден и алгоритм завершен; иначе j := j + 1, выбрать из поля ссылки адрес m_j и перейти к шагу 2.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм размещает элементы в ячейках таблицы, связывая их друг с другом последовательно через поле ссылки. При этом элементы не могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хеш-функции. Таким образом, дополнительные коллизии не возникают. В итоге в таблице возникают своеобразные цепочки связанных элементов, откуда происходит и название данного метода — «метод цепочек».

На рис. 2.5 проиллюстрировано заполнение хеш-таблицы и таблицы идентификаторов для примера, который ранее был рассмотрен на рис. 2.4 для метода простейшего рехеширования. После размещения в таблице для поиска идентификатора A_1 потребуется 1 сравнение, для A_2 — 2 сравнения, для A_3 — 1 сравнение, для A_4 — 1 сравнение и для A_5 — 3 сравнения (сравните с результатами простого рехеширования).

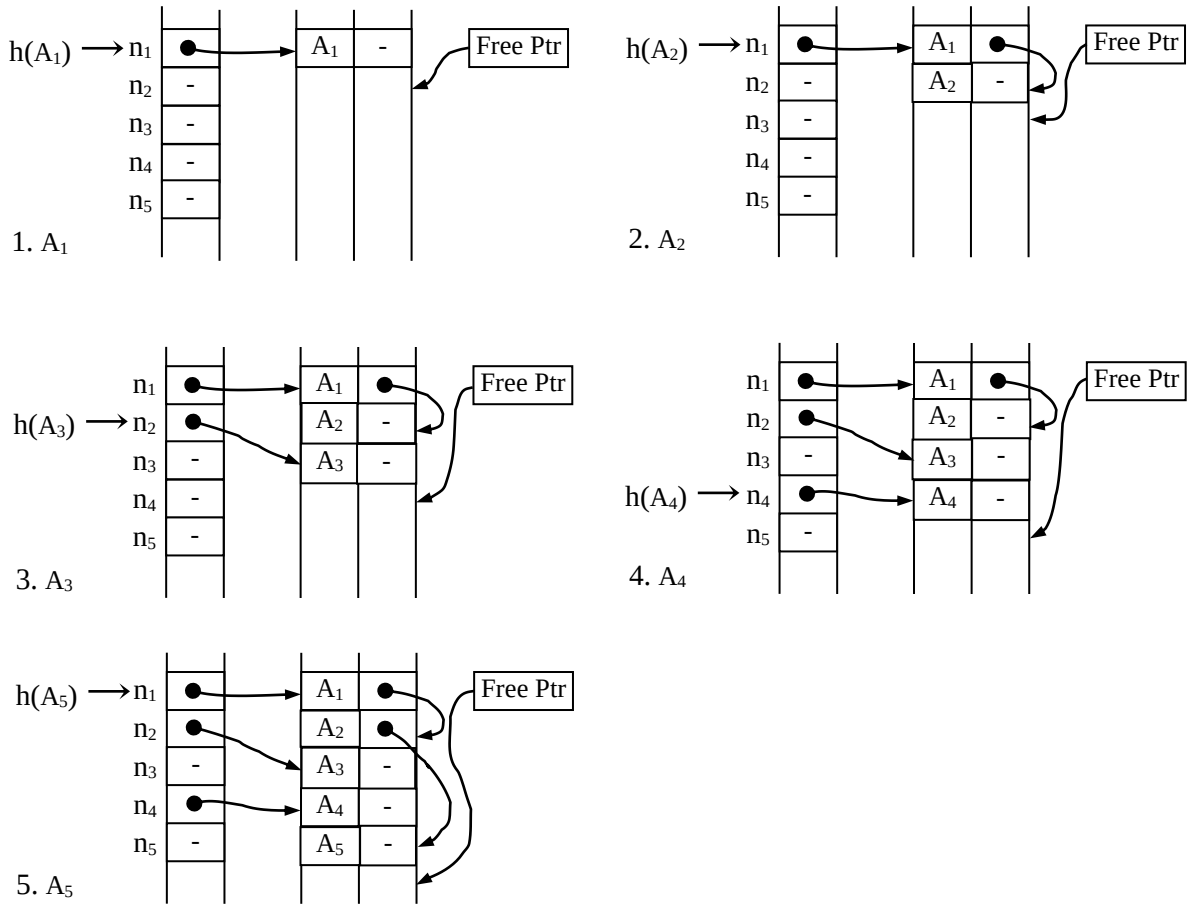

Рис. 2.5. Заполнение хеш-таблицы и таблицы идентификаторов при использовании метода цепочек

Метод цепочек является очень эффективным средством организации таблиц идентификаторов. Среднее время на размещение одного элемента и на поиск элемента в таблице для него зависит только от среднего числа коллизий, возникающих при вычислении хеш-функции. Накладные расходы памяти, связанные с необходимостью иметь одно дополнительное поле указателя в таблице идентификаторов на каждый ее элемент, можно признать вполне оправданными. Этот метод позволяет более экономно использовать память, но требует организации работы с динамическими массивами данных.

Комбинированные способы построения таблиц идентификаторов

Выше в примере была рассмотрена весьма примитивная хеш-функция, которую никак нельзя назвать удовлетворительной. Хорошая хеш-функция распределяет поступающие на ее вход идентификаторы равномерно на все имеющиеся в распоряжении адреса. Существует большое множество хеш-функций. Каждая из них стремится распределить адреса под идентификаторы по своему алгоритму, но, как было показано выше, идеального хеширования достичь невозможно.

В реальных компиляторах так или иначе используется хеш-адресация. Алгоритм хеш-функции составляет «ноу-хау» разработчиков компилятора. Обычно при разработке хеш-функции создатели компилятора стремятся свести к минимуму количество коллизий не на всем множестве возможных идентификаторов, а на тех их вариантах, которые наиболее часто встречаются во входных программах. Конечно,

принять во внимание все допустимые исходные программы невозможно. Чаще всего выполняется статистическая обработка встречающихся имен идентификаторов на некотором множестве типичных исходных программ, а также принимаются во внимание соглашения о выборе имен идентификаторов, общепринятые для входного языка. Хорошая хеш-функция — это шаг к значительному ускорению работы компилятора, поскольку обращение к таблицам идентификаторов выполняются многократно на различных фазах компиляции.

То, какой конкретно метод применяется в компиляторе для организации таблиц идентификаторов, зависит от реализации компилятора. Компилятор может иметь несколько таблиц идентификаторов, организованных на основе различных методов.

Как правило, применяются комбинированные методы. В этом случае, как и для метода цепочек, в таблице идентификаторов организуется специальное дополнительное поле ссылки. Но в отличие от метода цепочек оно имеет несколько иное значение. При отсутствии коллизий для выборки информации из таблицы используется хеш-функция, поле ссылки остается пустым. Если же возникает коллизия, то с помощью поля ссылки организуется поиск идентификаторов, для которых значения хеш-функции совпадают, по одному из рассмотренных выше методов. При хорошо построенной хеш-функции коллизии будут возникать редко, поэтому количество идентификаторов, для которых значения хеш-функции совпали, будет не столь велико. Тогда и время поиска одного среди них будет незначительным (в принципе, при высоком качестве хеш-функции подойдет даже перебор по неупорядоченному списку).

Такой подход имеет преимущество по сравнению с методом цепочек, поскольку не требует использования промежуточной хеш-таблицы. Недостатком метода является необходимость работы с динамически распределяемыми областями памяти. Эффективность такого метода, очевидно, в первую очередь зависит от качества применяемой хеш-функции, а во вторую — от метода организации дополнительных хранилищ данных.

Хеш-адресация — это метод, который применяется не только для организации таблиц идентификаторов в компиляторах. Данный метод нашел свое применение и в операционных системах, и в системах управления базами данных. Интересующиеся читатели могут обратиться к соответствующей литературе [4 т.2, 5, 18, 59].

Контрольные вопросы и задачи

Вопросы

1. Перечислите основные этапы и фазы компиляции.
2. Верно ли что любой компилятор является транслятором? Может ли существовать транслятор, который является компилятором? Может ли существовать компилятор, который является интерпретатором?
3. Можно ли построить компилятор, который не содержит лексический анализатор? Как могут быть связаны между собой лексический и синтаксический анализ?

4. Какие фазы работы компилятора, приведенной в этом пособии, будут отсутствовать у интерпретатора?
5. Какая информация может храниться в таблице идентификаторов?
6. Исходя из каких характеристик оценивается эффективность того или иного метода организации таблицы?
7. Какие существуют способы организации таблиц идентификаторов?
8. Что такое коллизия? Почему она происходит при использовании хэш-функций для организации таблиц идентификаторов?
9. В чем заключается преимущества и недостатки метода цепочек по сравнению с методом рехеширования?
10. Метод логарифмического поиска позволяет значительно сократить время поиска идентификатора в таблице. Однако он же значительно увеличивает время на добавление нового идентификатора в таблицу. Почему, тем не менее, можно говорить о преимуществах этого метода по сравнению с поиском методом прямого перебора?
11. Проблемы создания хорошей хэш-функции связаны с ограниченной разрядной сеткой ЭВМ и, следовательно, ограниченным размером доступного адресного пространства. Но, как сказано в литературе [14, 62], размер адресного пространства можно увеличить, используя внешние накопители данных (прежде всего жесткие диски) и механизм виртуальной памяти. Почему эти возможности не используются компиляторами при организации хэш-функций?

Упражнения

1. Напишите программу, реализующую метод логарифмического поиска в упорядоченном массиве строк. В качестве исходных данных для заполнения массива возьмите любой текстовый файл, считая, что все слова в нем являются идентификаторами.

1. Напишите программу, реализующую метод построения бинарного дерева. Используйте динамические структуры данных для организации дерева. В качестве исходных данных для заполнения дерева возьмите любой текстовый файл, считая, что все слова в нем являются идентификаторами. Проверьте эффективность метода на случайных текстовых файлах, а также попробуйте в качестве источника данных файл с упорядоченным текстом.
2. Напишите программу, создающую таблицу идентификаторов с помощью хэш-функций на основе метода простого рехэширования. В качестве исходных данных для заполнения дерева возьмите любой текстовый файл, считая, что все слова в нем являются идентификаторами. Организуйте программу таким образом, чтобы в ней можно было бы легко подменять используемую хэш-функцию. Подсчитывая число коллизий и среднее количество сравнений для поиска идентификатора, сравните результаты для различных хэш-функций. В качестве исходных данных для хэш-функции можно предложить:
   - коды первых двух букв идентификатора;

- коды последних двух букв идентификатора;
- код первой и код последней букв идентификатора;
- коды первой, последней и средней букв идентификатора.

3. Напишите программу, строящую таблицу идентификаторов по комбинированному способу: при возникновении коллизии новый идентификатор помещается в динамический неупорядоченный список через ссылку в поле основной таблице идентификаторов. При поиске внутри списка используется простой перебор. Подсчитывая число коллизий и среднее количество сравнений для поиска идентификатора, сравните результаты для хэш-функций, предложенных в задаче №5. Попробуйте предложить свой вариант хеш-функции и сравните полученные результаты.