Министерство образования Российской Федерации
Ярославский государственный университет имени П.Г. Демидова

Н.М. Бадин

Методы трансляции

Учебное пособие

Ярославль 2000

ББК В185.2я73+З973.2-018.1я73
Б15
УДК 519.682, 681.3

Методы трансляции: Учеб. пособие / Сост. Н.М. Бадин; Яросл. гос. ун-т. Ярославль, 2000. 59с.

ISBN 5-8397-0054-1

В данном пособии рассматриваются основы методов трансляции для языков программирования высокого уровня, таких как Си и Паскаль. Здесь представлен материал, содержащий как теоретическую основу, так и практические приемы для конструирования трансляторов. Описывается общая схема и отдельные этапы трансляции, алгоритмы их функционирования.

Пособие предназначено для студентов университетов, обучающихся по специальности 010200 «Прикладная математика» и по направлению 510200 «Прикладная математика и информатика». Брошюра также может быть полезна специалистам в области системного и прикладного программного обеспечения.

Ил. 6. Библиогр.: 20 назв.

Рецензенты: кафедра математического анализа Ярославского государственного педагогического университета им. К.Д. Ушинского; д-р техн. наук, профессор, директор Института микроэлектроники и информатики РАН В.А. Курчидис.

ISBN 5-8397-0054-1

Предисловие

Учебный курс "Языки программирования и методы трансляции" к настоящему времени стал в университетах одним из базовых курсов по информатике для многих специальностей и направлений. При этом "методы трансляции" - та область системного программирования, которая для современного программиста является таким же фундаментом, как, например, математический анализ для математика. Узость и специфичность методов трансляции на самом деле обманчива. Применяемые здесь алгоритмы и способы организации данных носят общий характер и используются в различных областях системного и прикладного программирования, таких как, например, организация баз данных или входные языки для различных прикладных систем (например, бухгалтерских).

Данное пособие предназначено, в первую очередь, для студентов университетов, обучающихся по специальности 010200 “Прикладная математика” и по направлению 510200 “Прикладная математика и информатика”. В нем представлены основы теории конструирования трансляторов. Здесь рассматриваются структура и классические методы построения трансляторов для языков программирования высокого уровня. К таковым относятся, например, языки Паскаль, Си и др. Представлены основные этапы трансляции, которые включают лексический, синтаксический и семантический анализ исходного текста программы, построение и обработку таблиц имен, формирование внутреннего представления программы, генерацию и оптимизацию объектного кода.

Объем материала соответствует семестровому лекционному курсу по методам трансляции, который автор в течение ряда лет читал на факультете информатики и вычислительной техники Ярославского государственного университета им. П.Г.Демидова. Данный курс является продолжением семестрового курса по формальным грамматикам и языкам. Этому курсу соответствует учебное пособие В.А. Соколова, "Формальные языки и грамматики", изд-во ЯрГУ, 1998. Поэтому здесь в первую очередь уделяется внимание практическим приемам и алгоритмам организации данных и их обработки. Более подробно разбираются те этапы транслятора, на которых производится анализ исходного текста программы.

Оглавление

Предварительные сведения

Под термином "трансляция" в применении к языкам программирования (ЯП) обычно понимается перевод в широком смысле с одного языка на другой. Это может быть, например, конвертор с языка Си на язык Ада, или преобразование ассемблерной программы в машинные коды. Когда рассматривается трансляция программы, написанной на языке высокого уровня, например, Паскаль, в процессе которой генерируется некоторый объектный код, то говорят о компиляции. В данном пособии ведется речь именно о методах построения компиляторов, т.е. о трансляции с языка высокого уровня в язык ассемблера или машинные команды.

В настоящее время разработано большое число (не одна тысяча) самых разнообразных языков программирования. Они довольно сильно отличаются друг от друга по назначению, организации, сложности, привязке к типу компьютера и его операционной системе. При этом даже для одного конкретного языка его реализации (компиляторы) могут значительно отличаться друг от друга.

Однако во всех ЯП есть нечто общее, их характеризующее. Это нечто и определяет основные (общие для всех языков) принципы организации транслятора.

1. Языки высокого уровня предназначены для повышения надежности и облегчения программирования. Поэтому их операторы и структуры данных более мощные, чем в машинных языках (5-7 машинных команд соответствуют в среднем одному оператору языка программирования высокого уровня).

2. Для повышения наглядности программ используется символическое или графическое представление конструкций языка, приближенное к человеческому восприятию (вместо кодов, отображаемых в одной из систем счисления).

3. Для любого языка определяются:

3.1. множество символов, которые можно использовать для записи правильных программ (алфавит), основные элементы;

3.2. множество правильных программ (синтаксис);

3.3. "смысл" каждой правильной программы (семантика). Любой транслятор при этом можно считать заданным в виде множества пар (x,y), где x - программа на исходном языке (исходная программа), y - программа на выходном языке.

Язык программирования, как и любая сложная система, имеет некоторую иерархию понятий, определяющую взаимосвязь между его компонентами. Эти понятия связаны между собой в соответствии с определенными правилами.

Каждая из программ, построенная в соответствии с языком имеет определенную иерархическую структуру, подчиненную правилам языка.

В связи с этим для всех языков и их программ можно выделить еще следующие общие черты:

- каждый язык должен содержать правила, позволяющие порождать программы, соответствующие этому языку или распознавать соответствие языку некоторых написанных программ.

Связи структуры программы с языком называется синтаксическим отображением.

Пример. a+b*c в большинстве языков программирования имеет явно выраженную иерархическую структуру (можно отобразить в виде дерева). В языке программирования при этом обычно существуют правила, представленные на некотором метаязыке.

Представленные программы в виде иерархии понятий (иерархической структуры) задает синтаксическую структуру программы как иерархию синтаксических понятий. Процесс нахождения синтаксической структуры заданной программы называется синтаксическим анализом.

Синтаксическая структура, правильная для одного языка может быть неправильной для другого. Например, в языке ФОРТ приведенное выражение не будет распознано. Однако для этого языка корректным будет являться выражение

abc * +

Другой характерной особенностью языка является его семантика. Семантикой определяется смысл операций языка, корректность операторов. Даже в языках имеющих одинаковую синтаксическую структуру семантика может быть различной (Си, Паскаль, …).

Описание семантики и распознавание ее корректности обычно является самой трудоемкой и объемной частью трансляторов (необходимо осуществить перебор и анализ множества вариантов).

Для их формального представления используются специальные метаязыки (языки, описывающие другие языки), грамматики (правила построения языков и отображения одного языка в другой).

1. Общая схема трансляции

В соответствии с тем, что в ЯП есть общие особенности и закономерности, в компиляторах существуют общие подсистемы и процессы.

Исходная программа, написанная на некотором языке программирования, является цепочкой знаков, которая переводится в другую цепочку (объектный код). При этом можно выделить следующие этапы:

1. Лексический анализ.

2. Синтаксический анализ.

3. Семантический (контекстный) разбор (порождение промежуточного представления).

4. Построение и исследование промежуточного представления.

5. Оптимизация.

3. Генерация кода.

На всех фазах (этапах) происходит работа с таблицами, а также анализ и исправление ошибок.

На фазе лексического анализа (ЛА) входная программа, представляющая собой поток символов, разбивается на лексемы - слова в соответствии с определениями языка. Основным формализмом, 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 за очередной лексемой, либо как полный проход, результатом которого является файл лексем.

В процессе выделения лексем ЛА может, как самостоятельно строить таблицы имен и констант, так и выдавать значения для каждой лексемы при очередном обращении к нему. В этом случае таблица имен строится в последующих фазах (например, в процессе синтаксического анализа).

На этапе ЛА обнаруживаются некоторые (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 автоматизации построения синтаксических анализаторов. Результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицу имен. 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 строится таблица символов, которую можно рассматривать как таблицу имен, пополненную информацией об описаниях (свойствах) объектов.

Основным формализмом, использующимся при семантическом (контекстном) анализе, являются атрибутные грамматики. Результатом работы фазы контекстного анализа является атрибутированное дерево программы. Информация об объектах может быть, как рассредоточена в самом дереве, так и сосредоточена в отдельных таблицах символов. В процессе контекстного анализа также могут быть обнаружены ошибки, связанные с неправильным 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, триады, тетрады 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 компиляции. 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 компилятора могут либо отсутствовать совсем, либо объединяться. В простейшем случае однопроходного компилятора нет явной фазы генерации промежуточного представления и оптимизации, остальные фазы объединены в одну, причем нет и явно построенного синтаксического дерева.

Пример программы

Рассмотрим простой распознаватель, написанный на языке Си и включающий первые два этапа компиляции: лексический и синтаксический анализ. Грамматика описывает битовое выражение вида

идентификатор << константа или идентификатор >> константа

<бвыр> -> <идент><оп><цкон>
<оп> -> >>|<<
<идент> -> <бук>{<бук>|<циф>}7
<цкон> -> <циф>{<циф>}
<бук> -> a|b|...|z
<циф> -> 1|2|...|9

Программа проверяет грамматическую корректность входного выражения и выдает, если необходимо, код сообщения об ошибке. Пробелы и аналогичные разделители игнорируются. Здесь реализован метод рекурсивного спуска по принципу: правило в грамматике - функция в программе. Каждая функция возвращает ноль, если возникла ошибка, или адрес символа входной строки, следующего за успешно распознанной лексемой.

```
main(int ac, char *av[]) {
  char s[77];
  if(ac>1) {fprintf(stderr,"Запуск: >bv \n");exit(0);}
  gets(s);
  if(bv(s)==0) printf("Ошибка!\n");
  else printf("Ok!\n");
}
/*синтаксический анализ*/
int bv(char *s) { void pr(),err();
  char *lit, *id(char*), *op(char*), *con(char*);
  pr(s);
  if(!(lit=id(s))) err(1);
  else if(!(lit=op(lit))) err(2);
       else if(!(lit=con(lit))) err(3);
```

```
            else if(!(lit=!*lit)) err(4);
return((int)lit);
}

/*лексический анализ*/
void pr(char *s) {
  char *p=s-1; int i=0;
  while(*++p!=NULL)
    if(!isspace(*p)) *(s+i++)=*p;
  *(s+i)=0;
}
char* id(char s[]) { char *p=s;
  if(!isalpha(*p++)) return(NULL);
  while(isalnum(*p)) p++;
return(p);
}
char* op(char *s) {
  if(*s=='>'&&*(s+1)=='>'||*s=='<'&&*(s+1)=='<') s+=2;
  else s=0;
  return(s);
}
char* con(char *s) {
  if(!isdigit(*s)) return(NULL);
  while(isdigit(*++s));
return(s);
}
/*печать кода ошибок*/
void err(int kod) {
  fprintf(stderr,"%i\n",kod);
}
```

2. Лексический анализ

Основная задача лексического анализа - разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов, или лексем, т.е. выделить эти слова из непрерывной последовательности символов. Все символы входной последовательности с этой точки зрения разделяются на символы, принадлежащие каким-либо лексемам, и символы, разделяющие лексемы (разделители). В некоторых случаях между лексемами может и не быть разделителей. С другой стороны, в некоторых языках лексемы могут содержать незначащие символы (пробел в Фортране). В Си разделительное значение символов - разделителей может блокироваться ('\' в конце строки внутри "...").

Обычно все лексемы делятся на классы. Примерами таких классов являются числа (целые, восьмеричные, шестнадцатеричные, действительные и т.д.), идентификаторы, строки. Отдельно выделяются ключевые слова и символы пунктуации (иногда их называют символы-ограничители). Как правило, ключевые слова - это некоторое конечное подмножество идентификаторов. В некоторых языках (например, ПЛ/1) смысл лексемы может зависеть от ее контекста и невозможно провести лексический анализ в отрыве от синтаксического.

С точки зрения дальнейших фаз анализа лексический анализатор выдает информацию двух сортов. Для синтаксического анализатора, работающего вслед за лексическим, существенна информация о последовательности классов лексем, ограничителей и ключевых слов. Для контекстного анализа, работающего вслед за синтаксическим, важна информация о конкретных значениях отдельных лексем (идентификаторов, чисел и т.д.). Поэтому общая схема работы лексического анализатора такова. Сначала выделяем отдельную лексему (возможно, используя символы- разделители). Если выделенная лексема - ограничитель, то он (точнее, некоторый его признак) выдается как результат лексического анализа. Ключевые слова распознаются либо явным выделением непосредственно из текста, либо сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых слов. Если да, то выдается признак соответствующего ключевого слова, если нет - выдается признак идентификатора, а сам идентификатор сохраняется отдельно. Если выделенная лексема принадлежит какому-либо из других классов лексем (число, строка и т.д.), то выдается признак класса лексемы, а значение лексемы сохраняется. Лексический анализатор может работать или как самостоятельная фаза компиляции, или как подпрограмма, работающая по принципу "дай лексему". В первом

случае выходом лексического анализатора является файл лексем, во втором лексема выдается при каждом обращении к лексическому анализатору (при этом, как правило, тип лексемы возвращается как значение функции "лексический анализатор", а значение передается через глобальную переменную).

С точки зрения формирования значений лексем, принадлежащих классам лексем, лексический анализатор может либо просто выдавать значение каждой лексемы и в этом случае построение таблиц переносится на более поздние фазы, либо он может самостоятельно строить таблицы объектов (идентификаторов, строк, чисел и т.д.). В этом случае в качестве значения лексемы выдается указатель на вход в соответствующую таблицу. Работа лексического анализатора описывается формализмом конечных автоматов. Однако непосредственное описание конечного автомата неудобно практически. Поэтому для описания лексических анализаторов, как правило, используют либо формализм регулярных выражений, либо формализм контекстно-свободных грамматик, а именно подкласса автоматных, или регулярных, грамматик.

Все три формализма (конечных автоматов, регулярных выражений и автоматных грамматик) имеют одинаковую выразительную мощность. По описанию лексического анализатора в виде регулярного выражения или автоматной грамматики строится конечный автомат, распознающий соответствующий язык.

Функции лексического анализатора (сканера)

Можно выделить следующие основные функции, которые должны выполняться сканером.

1. Основная задача сканера, уже довольно подробно описанная ранее, - это распознавание во входной строке лексем и определение их класса (токена).
2. Фильтрация или редактирование исходного текста. Сканер должен распознавать и убирать лишние элементы текста, которые не существенны с точки зрения дальнейшей работы компилятора. Примерами таких элементов являются всевозможные разделители типа пробела, комментарии и т.п. Конечно, должны учитываться различные ситуации. Например, в Паскале и Си пробел является разделителем лексем, а в Фортране IV - нет.

   Кроме того, здесь может потребоваться перекодировка исходного текста.

3. На этапе лексического анализа начинает формироваться таблица имён (идентификаторов). Данная функция подробно рассматривается позднее.
4. Поиск и вывод лексических ошибок. Таковых не так уж много: неверный входной символ, слишком длинный идентификатор (предупреждение), повторное определение метки (с использованием таблицы имен) и т.п. При работе сканера под управлением синтаксического анализатора также возможно, например, распознавание неверной лексемы.
5. Простейший синтаксический контроль, например, парности операторных скобок (begin и end), обязательного наличия вслед за ключевым словом if, ключевого слова then (в Паскале).

Лексическая свертка

Итак, на входе у сканера - исходный текст программы. На выходе сканера формируется последовательность токенов (типов лексем), сопровождаемая таблицами имен, констант и т.п. Такая последовательность называется лексической сверткой.

Внутреннее представление типов лексем обычно является набором целочисленных кодов. Например,

0 - лексема не определена;

1 - ключевое слово языка;

2 - идентификатор;

3 - '('

4 - ')'

и тому подобное. Тогда, проанализировав исходный текст, лексический анализатор представит его в виде последовательности кодов со ссылками, если необходимо, на элементы соответствующих таблиц. С данной лексической сверткой и будет работать синтаксический анализатор.

3. Организация таблиц имен

Практически любой компилятор использует в своей работе таблицы, хранящие различную информацию об объектах из анализируемой программы. Среди таких таблиц чаще всего используются таблицы имен, которые формируются на этапах лексического и синтаксического анализа и используются практически в течение всей работы компилятора.

Таблица имен (символов, идентификаторов) содержит не менее двух полей для каждого имени: само имя или ссылку на него (если имена различной длины) и дескриптор имени. Дескриптор - это обычно ссылка на область, где описываются характеристики данного объекта. Например, это может быть тип, метка, имя функции, формальный параметр, переменная, тип памяти (статический или динамичный). Для агрегата - его характеристики (размерность и, возможно, границы массива) и т.д. В простых случаях вся необходимая информация содержится в самой таблице имен вместе с именем.

Информация, хранимая в ТИ, используется на разных этапах компиляции, но в первую очередь при проверке правильности семантики и генерации кода. Так в любом арифметическом подвыражении необходимо знать тип аргументов, чтобы для данной операции выработать соответствующий тип результата или сообщение об ошибке (если язык запрещает данную комбинацию типов). С точки зрения операций, которые выполняются над таблицами идентификаторов, наиболее употребляемыми являются две: вставка или добавление имени и поиск имени. Операция удаления из таблицы тоже может встречаться (например, при анализе с возвратами), но все же оно гораздо реже употребляется по сравнению с первыми двумя. Надо отметить, что количество имен в программе может достигать нескольких тысяч, а это приводит к необходимости использовать эффективные как по времени, так и по памяти методы организации таблиц имен в компиляторах. В частности сканер по данному имени должен быстро определить, отведена ли ему уже ячейка в ТИ, а если нет, то быстро вставить идентификатор в таблицу.

Наконец, надо заметить, что характер организации таблицы зависит от ее возможного размера. Его максимум может быть известен, например, когда таблица содержит ключевые слова языка. В случае идентификаторов число их заранее предсказать нельзя. Возможна и комбинация данных ситуаций, что приводит к необходимости инициализировать (частично заполнять) таблицу имен в начале компиляции программы.

3.1. Простейшие методы

Прежде чем перейти к описанию наиболее эффективного метода построения таблиц с помощью хэш-функций кратко рассмотрим другие способы.

Самый простой метод - линейный для неупорядоченной таблицы. Здесь новый элемент просто добавляется в конец таблицы. Размер таблицы соответствует количеству фактически встретившихся имен. Но операция поиска делается последовательным просмотром, что довольно долго. В среднем поиск требует O(n) (n/2) операций, где n - число имен в таблице. Для наибольшего числа (n<20) (по Грису) метод весьма эффективен. Если в качестве таблицы использовать стек, то метод будет удобен для языков с блочной структурой. Под переменные одного блока отводится часть стека, которая освобождается, когда блок закрывается. Здесь не возникает проблем и с использованием одного и того же имени в разных (в том числе вложенных) блоках.

Если число объектов невелико и можно для каждого ожидаемого имени зарезервировать отдельную ячейку, то используются таблицы с прямым доступом. В данном случае имя как бы одновременно является и адресом в таблице. Простой пример такой ситуации - это когда надо посчитать частоты использования, скажем, букв в тексте. Тогда код буквы можно использовать в качестве индекса массива, элементы которого содержат число вхождений соответствующих букв.

```
int  m[26], c;
while ((c=getchar( ))!=0) m[c-'A']=(isupper(c)) ? m[c-'A']+1 : ;
```

К сожалению, метод прямого доступа в большинстве случаев приходится отвергать, т.к. здесь расчет на то, что число всех возможных объектов мало, но для языков программирования это не так. В частности, при длине идентификатора не более 6 (как у Фортрана) их число около 10^9.

Для большого количества имен вполне приемлема упорядоченная таблица с бинарным (логарифмическим) поиском. Здесь в каждый момент объекты упорядочены некоторым образом (например, по алфавиту). Поиск в таком случае производится весьма быстро методом деления отрезка пополам и время его пропорционально log(n). Недостатком данного метода является то, что при добавлении очередного объекта надо сохранять порядок в таблице. Для этого можно использовать, например, метод упорядоченных вставок.

Сначала находится какой-то объект, непосредственно за которым должен стоять добавляемый объект. Все объекты с к+1-го по n-й сдвигаются на одну позицию в таблице, освобождая место под новый объект. Тем самым, метод с упорядоченной таблицей и бинарным поиском больше подходит к

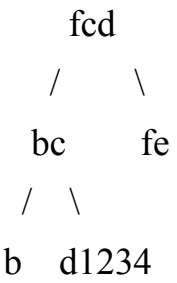
ситуации, когда заполнение таблицы предшествует поискам, т.е. упорядочение можно выполнить после заполнения. Отдельная таблица с ключевыми словами языка используется только для поиска, а потому сразу может быть упорядочена.

Развитием последнего метода является организация таблицы имен в виде бинарного дерева. В данном случае также предполагается, что объекты могут быть упорядочены некоторым образом. Каждая вершина дерева соответствует одному объекту (имени) и может иметь левого потомка, и правого b и c соответственно. Должно выполняться условие: b<a<c.

Каждое новое имя, если оно отсутствует в дереве, порождает новый лист. Так, для следующих имен, распознаваемых лексическим анализатором именно в таком порядке

fcd, bc, b, fe, d1234,

бинарное дерево будет иметь вид:



При сравнении более короткое имя дополняется справа пробелами, которые предшествуют любым другим печатным символам. Новое имя c2c станет левым потомком вершины, соответствующей имени d1234.

Алгоритм поиска по дереву по существу совпадает с алгоритмом вставки. Можно показать, что для n вершин время успешного поиска будет пропорционально log(n).

3.2. Хеширование

Наиболее эффективный и широко применяемый в компиляторах метод работы с таблицами имен дают так называемые hash-функции. H-функция отображает множество объектов (имен) во множество целых чисел от 0 до N-1, т.е. каким-то образом преобразует имя в индекс элемента таблицы, куда оно должно быть помещено. Фактически, примером такой функции является рассмотренный ранее метод прямого доступа, где hash-функция определена на множество символов ASCII и ее значение - это код символа. В литературе хеширование именуется еще "методом расстановки", "рассеивающей функцией" и пр.

Необходимым условием исследования hash-функции является наличие таблицы заданной длины N. При этом над именем совершаются некоторые арифметические действия, в результате чего сразу вычисляется адрес в таблице. Поскольку число возможных имен очень велико, а N - нет, то весьма возможны ситуации, когда h(a)=h(b), где a=b. Такая ситуация называется коллизией, т.е. по сути надо искать взаимно-однозначную функцию.

Такие функции весьма редки даже для довольно больших таблиц. Знаменитый парадокс дней рождения утверждает, что с большой вероятностью совпадут дни рождения хотя бы у двоих, если в комнате не менее 23 человек. Другими словами, для h-функции, случайным образом отображающей любой объект в 365-элементную таблицу, вероятность коллизии уже для 23 объектов равна 0.51, т.е. больше половины.

Другой простой пример h-функции - это использование первой буквы идентификатора. Номер этой буквы в алфавите совпадает с индексом имени в таблице. Очевидно, что размер таблицы в данном случае будет равен 26 или 52, а для всех идентификаторов, начинающихся с одной буквы, будут возникать коллизии.

Если hash-функция определена, то можно считать, что время ее вычисления есть некоторая константа, т.е. не зависит от числа обрабатываемых имен. Правда, данная константа может оказаться весьма большой, если алгоритм вычисления довольно сложный. В последнем примере достаточно взять младший байт имени и вычесть 64, чтобы получить индекс. В предыдущем примере надо воспользоваться генератором случайных чисел.

Тем самым при построении hash-функций надо стараться избегать сложных вычислений с использованием умножения или извлечения корня, заменяя их, если возможно, на соответствующие операции сложения, вычитания, сдвиги.

Таким образом, хорошая h-функция должна удовлетворять двум требованиям:

1) ее вычисление должно быть очень быстрым;

2) она должна минимизировать число коллизий.

Первое условие отчасти зависит от архитектуры ЭВМ, а второе - от свойств данных. Н-функция для получения хорошего рассеивания по таблице должна учитывать, по возможности, всю информацию, заключенную в имени. До сих пор мы не ограничивали длину имени, но для быстроты вычислений желательно иметь одно машинное слово. Для этого можно воспользоваться сложением или операцией "исключающее или". Для обеих операций их результат зависит от всех битов аргументов.

Виды h-функций

Для определенности будем считать, что 0<h(a)<N, где a - машинное слово, содержащее свернутую информацию об исходном имени, а N-размер таблицы. По возможности, h-функция должна рассеивать по всей таблице, а значит учитывать N. Более того, на практике исходная информация отнюдь не случайна. Например, идентификаторы часто имеют вид: SUM1, SUM2, SUM3. Однако не так трудно сделать достаточно хорошую функцию, позволяющую уменьшить число коллизий даже для неслучайной исходной информации.

Весьма широко используются на практике и дают хорошие результаты следующие h-функции.

Пусть N=2 .Метод "середины квадрата", когда в качестве значения h(*a*) берутся средние *k* битов квадрата *a*a*. В этом случае значение h-функции хорошо связано со всеми битами и дает различные адреса в таблице при условии, что *a* не содержит много левых или правых нулей подряд. К тому же эта функция достаточно сложна.

Аналогичный метод "середины квадратного корня" *a*.

Слово *a* можно разбить на несколько частей по *k* ($N=2^n$) бит, просуммировать их и взять в качестве h(*a*) младшие *k* битов суммы.

Очень хорошо работает на практике метод "деления"(по Кнуту):

$$h(a)=a \bmod N$$

При этом эффективность метода зависит от выбора N. Очевидно, что при четном N четность h совпадает с четностью *a*, что может приводить к дополнительным коллизиям. Совсем плохо, если $N=2^n$, т.к. тогда h(*a*) не зависит от старших разрядов *a*. Хороший анализ данного метода дает Кнут D. В результате он приходит к выводу, что лучше всего взять в качестве N простое число, которое не является делителем *r*+*l*, где *r*-"основание системы счисления" для множества символов компьютера (для нас *r*=256), а *k* и *l* -небольшие числа. Данный метод хорош еще и тем, что в языке Cu реализуется лишь одной операцией %.

Все рассмотренные методы построения h-функций должны проверяться в конкретных практических ситуациях. Может оказаться так, что хорошая со статистической точки зрения h-функция на зарезервированных словах дает много коллизий.

Методы разрешения коллизий

Поскольку для различных объектов h-функция может давать одинаковые значения, необходимо иметь способ и в таком случае находить свободное место в таблице. При этом метод должен быть детерминирован в том смысле, что, помещая имя после одной или нескольких коллизий в свободную ячейку, мы

должны быть уверены, что найдем его впоследствии. Фактически каждое имя определяет некоторую последовательность проб, т.е. последовательность просматриваемых ячеек в таблице. Задача сводится к определению такой последовательности.

Общий класс методов данного разрешения коллизий У.Петерсон (1957) назвал "открытой адресацией". В этом случае h-функция представляется как набор m функций $h_0,h_1,\dots ,h_m$, каждая из которых соответствует очередной пробе размещения имени, если предыдущая проба окончилась неудачей. h_0-первичная (начальная) проба и вид этой функции обсуждался выше.

Пусть *a* - имя, N - число элементов в таблице, h (i от 0 до m)-функции, отображающие имена в 0,...,N-1.

Алгоритм вычисления адреса в таблице будет выглядеть так:

1. Вычисляем $h_0(a)$ (i=0)

2. Если ячейка $h_i(a)$ пуста, то *a* ранее не встречался, размещаем его в таблице и стоп.

3. Если h_i(a) занята, то сравниваем *a* с именем в ячейке. При совпадении полагаем $h(a)=h_i(a)$ и имя найдено. Выдается информация и стоп.

4. Если имена не совпадают, то

при $i=m$ таблица оказалась полна и поместить имя *a* в нее не удалось;

при $i<m$ увеличиваем *i* на 1 и возвращаемся на шаг 2.

Надо заметить, что Д.Грис называет данный метод рехешированием. Конкретный способ рехеширования будет определяться конкретным набором функций $h_0,h_1,\dots ,h_m$. Естественно, что для любого имени *a* значение $h_i(a)$ не должно совпадать с $h_j(a)$ при разных i и j , т.к. всегда хотелось бы найти пустую ячейку, если она есть.

Простейшим, но и одним из самых плохих является способ линейного рехеширования:

$h_i(a)=(h_0(a)+i) \bmod N$, где i от 1 до N-1.

В этом случае $h_{i+1}-h_i=1$ и мы последовательно двигаемся по ячейкам (за N-1 идет 0-я) пока не встретится пустая или не дойдем до $h_0(a)$.

Интересно, что таким методом разрешения коллизий пользовались в игре "Что, Где, Когда", когда в ней не было денег. Сектор с конвертом на столе соответствовал свободной ячейке. Если конверта в нем не было, то в направлении стрелки сектора перебирались один за другим, пока не попадался сектор с конвертом.

Данный метод хорош, к сожалению, только пока таблица мало заполнена. При наличии серий подряд идущих занятых мест вероятность попадания в

отдельную свободную ячейку становится меньше, чем в смежную с занятыми. К тому же появляется тенденция к объединению серий занятых ячеек. Среднее число проб при удачном поиске для данного метода равно

$(1+1/(1-\varepsilon))/2$,

где $\varepsilon=n/N$ - коэффициент заполнения таблицы.

Метод "случайного" рехеширования снимает проблему скопления занятых ячеек за счет использования псевдослучайных чисел r:

$h_i(a)=(h_0(a)+r_i) \bmod N$, где i от 1 до N-1.

Здесь надо использовать генератор случайных чисел ,дающий все числа в диапазоне от 1 до N-1 в точности по 1 разу. При этом при каждом обращении к h-функции должна выдаваться одна и та же последовательность: $r_1, \ldots r_{N-1}$.

Так для $N=2^k$ хорошие результаты дает следующий метод генерации:

1. R=1, i=1
2. R=R*5
3. R=R mod (4*N)
4. r_i=[R/4], i=i+1, переход к шагу 2.

Фактически, на очередной итерации, здесь за r_i берутся k старших рядов из младших k+2 разрядов от R*5. Среднее число проб при успешном поиске в данном случае:

$\log_2(1/(1-\varepsilon))/\varepsilon$

Метод рехеширования "сложением " есть модификация линейного. При этом :

$h_i(a)=(i*(h_0(a)+1)) \bmod N$, где i от 1 до N-1.

Данный метод хорош, если N-простое число. Тогда h покрывает все возможные ячейки таблицы. Причем очередную пробу легко вычислить:

$h_{i+1} = h_i + h_0 + 1$

Единица нужна в случае, когда h_0=O. К сожалению, этот метод не избавляет от скопления занятых.

Наконец возможен метод "квадратичного" рехеширования:

$h_i(a)=[h_0(a)+a*i^2 +b*i+c] \bmod N$,

где a,b,c- специально подбираемые константы , а N-лучше простое число.

Метод цепочек

По сравнению с методами "открытой адресации " разрешение коллизий данным способом требует динамической организации списков. Также должна

быть хорошая hash-функция для первоначального вычисления адреса в таблице .По этому адресу в ячейке или пусто, или адрес начала списка для имен, имеющих такое же значение h-функции. Т.е. все имена с одинаковым первичным адресом объединяются в последовательный список. Рассмотрим алгоритм подробнее.

Пусть имеется хеш-таблица или таблица индексов, которая будет содержать адреса в ТИ и в начальный момент вся нулевая. Пусть переменная p содержит адрес последнего занятого элемента в ТИ. В начале p - это адрес перед первой ячейкой ТИ. Наконец, каждый элемент ТИ имеет еще поле p2 - под адрес другого элемента этой же таблицы и вначале там тоже стоят нули.

H-функция дает адрес в h-таблице, который равен нулю или нет. В первом случае сюда записывается p+1 а по адресу p+1 - само имя. Во втором случае идем по цепочке, сравнивая имена, начиная с адреса из h-таблицы, до ячейки в ТИ с p2=0. Тогда p2=p+1, а по адресу p+1 заносим имя. Увеличиваем p: p=p+1.

Данный способ является во многом более эффективным, чем методы открытой адресации. В первую очередь за счет легкого расширения ТИ, что не требует пересчета h-функции. Кроме того, когда ТИ будет заполнена, т.е. этап лексического анализа завершен, то можно освободить память из под h-таблицы, так как все лексемы будут уже представляться своими кодами и конкретным адресом в ТИ. Приходится лишь жертвовать памятью под указатели p2, что вполне допустимо. Среднее число проб при удачном поиске здесь примерно равно $1+\varepsilon/2$.

3.3. Другие вопросы организации таблиц

Рассматривая методы хеширования, мы совсем не коснулись проблемы удаления объектов из таблицы. Простое удаление может приводить к потере других объектов (вспомните линейное рехеширование). Здесь возможны два основных подхода. Специальный алгоритм для каждого метода, как, например, переключение ссылок в методе цепочек. Или, если удалений не так много, то помечать удаленные имена. При поиске трактовать их как занятые, а если он окончился неудачей, то записать имя в первую встреченную удаленную или свободную позицию.

Довольно большим недостатком хеширования является плохая расширяемость таблицы. Это, правда, верно в основном для методов "открытой адресации". Если таблица оказалась мала, приходится строить новую и пересчитывать новые значения h-функции. На практике при ожидаемом n числе имен рекомендуется брать размер $N=3n/2$, что дает некоторый компромисс между длиной таблицы и временем поиска.

Сравнивая различные методы можно заметить, что использование h-функций дает более значительные результаты по быстродействию, чем обычные методы. Так, например, даже для линейного рехеширования по сравнению с обычным или бинарным поиском: для n=512, N=1024:

метод	*среднее число проб*
неупорядоченная таблица	256
бинарный поиск	9
линейное рехеширование	1,5

При увеличении числа объектов разница растет, даже если таблица сильно заполняется:

метод	*среднее число проб*		
	ε=10%	ε=50%	ε=90%
линейное рехеширование	1,06	1,05	5,50
случайное рехеширование	1,05	1,39	2,56
метод цепочек	1,05	1,25	1,45

Правда все это только для среднего случая, а крайний может оказаться очень плохим.

4. Методы синтаксического анализа

4.1. Основные понятия и определения

Пусть G - контекстно-свободная грамматика, где N - множество нетерминальных символов, T - множество терминальных символов, P - множество правил вывода и S - аксиома. Будем говорить, что uxv выводится за один шаг из uAv (и записывать это как uAv=>uxv), если A->x - правило вывода и u и v - произвольные строки из (N∪T)*. Если u1=>u2=>...=>un, будем говорить, что из u1 выводится un, и записывать это как

u1=>*un. T.e.:

1) u=>*u для любой строки u,

2) если u=>*v и v=>*w, то u=>*w.

Аналогично, "=>+" означает выводится за один или более шагов.

Если дана грамматика G с начальным символом S, отношение =>+ можно использовать для определения L(G) - языка, порожденного G. Строки L(G) могут содержать только терминальные символы G.

Строка терминалов w принадлежит L(G) тогда и только тогда, когда S=>+w. Строка w называется предложением в G. Если S=>*u, где u может содержать нетерминалы, то u называется сентенциальной формой в G. Предложение - это сентенциальная форма, не содержащая нетерминалов.

Рассмотрим выводы, в которых в любой сентенциальной форме на каждом шаге делается подстановка самого левого нетерминала.

Такой вывод называется левосторонним. Если S=>*u в процессе левостороннего вывода, то u - левая сентенциальная форма. Аналогично определяется правосторонний вывод.

Упорядоченным графом называется пара (V,E), где V обозначает множество вершин, а E - множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид

((v,e1),(v,e2),...,(v,en)). Этот элемент указывает, что из вершины a выходят n дуг, причем первой из них считается дуга, входящая в вершину e1, второй - дуга, входящая в вершину e2, и т.д.

Дерево вывода в грамматике G=(N,T,P,S) - это помеченное упорядоченное дерево, каждая вершина которого помечена символом из множества N∪T∪{e}. Если внутренняя вершина помечена символом A, а ее

прямые потомки - символами X1,...,Xn, то A -> X1X2...Xn - правило этой грамматики.

Упорядоченное помеченное дерево D называется деревом вывода (или деревом разбора) в КС-грамматике G(S)=(N,T,P,S), если выполнены следующие условия:

(1) корень дерева D помечен S;

(2) каждый лист помечен либо a<-T, либо e;

(3) каждая внутренняя вершина помечена нетерминалом;

(4) если N - нетерминал, которым помечена внутренняя вершина и X1,...,Xn - метки ее прямых потомков в указанном порядке, то N->X1...Xk - правило из множества P.

Автомат с магазинной памятью (сокращенно МП-автомат) - это семерка P=(Q,T,Г,d,q0,Z0,F), где

(1) Q - конечное множество символов состояний, представляющих всевозможные состояния управляющего устройства;

(2) T - конечный входной алфавит;

(3) Г - конечный алфавит магазинных символов;

(4) d - функция переходов - отображение множества Qx(T U {e})xГ в множество конечных подмножеств QxГ*, т.е. d:Qx(T U {e})xГ -> {QxГ*};

(5) q0<-Q - начальное состояние управляющего устройства;

(6) Z0<-Г - символ, находящийся в магазине в начальный момент (начальный символ);

(7) F<=Q - множество заключительных состояний.

Конфигурацией МП-автомата называется тройка (q,w,u)<-QxT*xГ*, где

(1) q - текущее состояние управляющего устройства;

(2) w - неиспользованная часть входной цепочки; первый символ цепочки w находится под входной головкой; если w=e, то считается, что вся входная лента прочитана;

(3) u - содержимое магазина; самый левый символ цепочки u считается верхним символом магазина; если u=e, то магазин считается пустым.

Такт работы МП-автомата P будем представлять в виде бинарного отношения |-, определенного на конфигурациях. При этом будем писать (q,aw,Zu)|-(q',w,vu), если множество d(q,a,Z) содержит (q',v), где q, q'<-Q, a<-T∪{e}, w<-T*, Z<-Г, u,v<-Г*. Начальной конфигурацией МП-автомата P называется конфигурация вида (q0,w,Z0), где w<-T*, т.е. управляющее устройство находится в начальном состоянии, входная лента содержит

цепочку, которую нужно распознать, а в магазине есть только начальный символ Z0. Заключительная конфигурация - это конфигурация вида (q,e,u), где q<-F, u<-Г*.

Говорят, что цепочка w допускается МП - автоматом P, если (q0,w,Z0)|-*(q,e,u) для некоторых q<-F и u<-Г*. Языком, определяемым (или допускаемым) автоматом P (обозначается L(P)), называют множество цепочек, допускаемых автоматом P.

Иногда допустимость определяют несколько иначе: цепочка w допускается МП-автоматом P, если (q0,w,Z0)|-*(q,e,e). Эти определения эквивалентны.

4.2. Нисходящий синтаксический анализ. Метод рекурсивного спуска

Основная проблема предсказывающего разбора - определение правила вывода, которое нужно применить к нетерминалу. Процесс предсказывающего разбора (сверху-вниз) с точки зрения построения дерева разбора можно проиллюстрировать рис. 4.1.

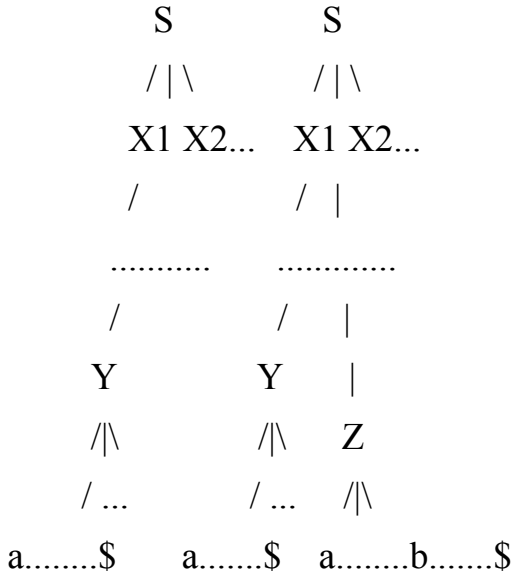


Рис. 4.1.

Фрагменты недостроенного дерева соответствуют сентенциальным формам вывода. Вначале дерево состоит только из одной вершины, соответствующей аксиоме S. В этот момент по первому символу входного потока предсказывающий анализатор должен определить правило S->X1 X2 ..., которое должно быть применено к S. Затем необходимо определить правило,

которое должно быть применено к X1, и т.д., до тех пор, пока в процессе такого построения сентенциальной формы, соответствующей левому выводу, не будет применено правило Y->a .... Этот процесс затем применяется для следующего самого левого нетерминального символа сентенциальной формы.

Таблично-управляемый предсказывающий анализатор имеет входной буфер, таблицу анализа и выход. Входной буфер содержит распознаваемую строку, за которой следует $ - правый концевой маркер, признак конца строки. Магазин содержит последовательность символов грамматики с $ на дне. Вначале магазин содержит начальный символ грамматики на верхушке и $ на дне. Таблица анализа - это двумерный массив M[A,a], где A - нетерминал, и a - терминал или символ $.

Анализатор управляется программой, которая работает следующим образом. Она рассматривает X - символ на верхушке магазина и a - текущий входной символ. Эти два символа определяют действие анализатора. Имеются три возможности.

1. Если X=a=$, анализатор останавливается и сообщает об успешном окончании разбора.

2. Если X=a#$, анализатор удаляет X из магазина и продвигает указатель входа на следующий входной символ.

3. Если X - нетерминал, программа заглядывает в таблицу M[X,a]. По этому входу хранится либо правило для X, либо ошибка. Если, например, M[X,a]={X->UVW}, анализатор заменяет X на верхушке магазина на WVU {на верхушке U}. Будем считать, что анализатор в качестве выхода просто печатает использованные правила вывода. Если M[X,a]=error, анализатор обращается к подпрограмме анализа ошибок. Поведение анализатора может быть описано в терминах конфигураций автомата разбора.

Выше был рассмотрен таблично-управляемый вариант предсказывающего анализа, когда магазин явно использовался в процессе работы анализатора. Можно предложить другой вариант предсказывающего анализатра, когда каждому нетерминалу сопоставляется, вообще говоря, рекурсивная процедура и магазин образуется неявно при вызовах этих процедур.

Процедуры рекурсивного спуска могут быть записаны, как это изображено на рис. 4.2. В процедуре N для случая, когда имеется альтернатива N->ui->*e (не может быть более одной альтернативы, из которой выводится e!), приведены два варианта 1.1 и 1.2.

В варианте 1.1 делается проверка, принадлежит ли следующий входной символ FOLLOW(N). Если нет - выдается ошибка. Во втором варианте этого

не делается, так что анализ ошибки откладывается на процедуру, вызвавшую N.

```
procedure N;{N -> u1 | u2 | ... | uk}
 begin if InSym<-FIRST(ui) {только одному!}
      then if parse(ui)
          then exit(N->ui)
          else error()
          end
       else
 Вариант 1: если имеется правило N->ui =>* e то
   Вариант 1.1:  if InSym<-FOLLOW(N)
                      then exit(N->e)
                      else error()
                      end;
   Вариант 1.2:  exit(N->e)
 Вариант 2: нет правила N->ui =>*e
          error()
 end   end;

procedure parse(u);
{из u не выводится e!}
begin v:=u;
     while v#e do
          {v=Xz}
          if X-терминал a
          then if InSym<>a
              then return(false)
              end
          else {X-нетерминал B}
              B;
          end;
          v:=z
     end;
     return(true)
end;
```

Рис. 4.2

Диаграммы переходов для рекурсивного спуска

Как правило, непосредственное программирование рекурсивного спуска из грамматики приводит к большому числу процедур. Число этих процедур можно уменьшить, заменив в некоторых правилах рекурсию циклом. Для этого можно воспользоваться диаграммой переходов грамматики, которая строится следующим образом.

Пусть имеется LL(1)-грамматика. Тогда для каждого нетерминала построение диаграммы включает следующие шаги.

Шаг 1. Вводим начальное и заключительное состояния.

Шаг 2. Для каждого правила вывода A->X1X2...Xn строим путь из начального в конечное состояние с дугами, помеченными X1,...,Xn. Если анализатор, построенный по диаграмме переходов, оказывается в состоянии s и дуга, помеченная терминалом a, ведет в состояние t, то если очередной входной символ равен a, анализатор продвигает вход на одну позицию вправо и переходит в состояние t. Если же дуга помечена нетерминалом A, анализатор входит в начальное состояние для A без продвижения входа. После того как он достигает заключительного состояния для A, он переходит в состояние t, что означает "чтение" A из входа при переходе из состояния s в состояние t. Наконец, если есть дуга из s в t, помеченная e, то анализатор переходит из s в t, не читая входа.

Если следовать программе рекурсивного спуска, то переход по e должен всегда выбираться в качестве последней альтернативы. Диаграммы переходов могут быть упрощены подстановкой одной в другую.

4.3. Предсказывающий нисходящий разбор

Рассмотрим теперь более подробно нисходящий синтаксический анализ при помощи метода рекурсивного спуска. Последний имеет довольно общий характер (применим для грамматик без левых рекурсий) и удобен для LL(k) грамматик.

В общем случае каждому нетерминалу исходной грамматики соответствует своя подпрограмма, которая по k очередным входным лексемам пытается выяснить, по какой продукции надо раскрывать данный нетерминал (т.е. строить поддерево). Если k>1, то мы имеем фактически анализ с возвратами, что не очень эффективно. В этом случае лучше использовать табличные методы распознавания.

Во многих случаях, аккуратно записав грамматику, исключив из нее левую рекурсию и применив к полученной грамматике левую факторизацию,

можно получить грамматику, для которой применим метод рекурсивного спуска без возвратов, т.е. так называемый предсказывающий разбор.

Для построения предсказывающего разбора на каждом шаге мы должны знать входной символ a, нетерминальный символ A, который раскрывается, и у которого имеется только одна из правых частей A->a1|a2|...an, начинающаяся с этого символа a. Такая ситуация характерна для большинства языков программирования, когда идет речь о выборе оператора по ключевому слову:

stmt -> "if" expr "then" stmt "else" stmt

| "while" expr "do" stmt

| "begin" stmt_list "end"

В предсказывающем разборе входной символ (очередная лексема) однозначно определяет для каждого нетерминала вызываемую рекурсивную процедуру. Последовательность вызовов процедур в процессе обработки входной строки неявно определяет дерево грамматического разбора для этой строки.

На самом деле для некоторого входного символа может не быть непосредственно правой части, которая с него начинается, а может быть правая часть, начинающаяся с нетерминала. Тогда надо вызвать соответствующую ему процедуру и поискать подходящую для этого входного символа правую часть и т.д. Например, для грамматики, описывающей арифметические выражения со скобками и одной операцией сложения

E -> E+T | Tбез рекурсии: E -> TE'

T -> (E) | id | con E'-> +TE' | e

T -> (E) | id | con

Предсказывающий разбор основывается на информации о том, какой первый символ может быть сгенерирован правой частью продукции. Для каждой правой части α можно определить множество FIRST(α), которое состоит из тех терминальных символов (лексем), с которых начинаются все строки, выводимые из α. Если из α может быть выведена пустая строка, то и она принадлежит FIRST(α). Например, для данной грамматики: FIRST(+TE')={+}, FIRST(TE')={(,id,con}.

Таким образом, с любым символом грамматики можно связать множество FIRST, которое строится следующим образом:

1) если X - терминал, то FIRST(X)=X;

2) если X->ε, то добавить ε к FIRST(X);

3) если X->Y1Y2..Yk, то все не ε символы FIRST(Y1) добавляем к FIRST(X), если ε принадлежит FIRST(Y1), то добавляем и из FIRST(Y2) и т.д.

Аналогичным образом строится и FIRST для любой строки символов. ε будет ему принадлежать, если пустая строка выводима из каждого символа строки.

Для нетерминалов грамматики получим:

FIRST(E)=FIRST(T)={(,id,con}

FIRST(E')={+, ε}

Теперь определим, как строится метод рекурсивного спуска без возвратов, т.е. предсказывающий разбор. Программа состоит из процедур, соответствующих нетерминалам грамматики. Каждая такая процедура должна выполнять две вещи.

Во-первых, процедура решает по входному символу *a*, какая из продукций используется на данном шаге для раскрытия нетерминала. Выбирается та правая часть α, для которой *a* принадлежит FIRST(α).

Если для какого-нибудь входного символа существует две такие правые части, то возникает конфликт и данный метод грамматического разбора не применим к такой грамматики. Продукция с ε справа применяется, если ни какая другая правая часть для входного символа не подошла.

Во-вторых, процедура по найденной правой части производит дальнейший анализ. Если стоит нетерминал, то вызывается соответствующая ему процедура. Если стоит терминальный символ, то он должен совпадать с входным символом. В этом случае считывается следующий входной символ. В некоторый момент символы могут не совпасть, что будет означать возникновение ошибки.

Напишем некоторый C подобный код для нисходящего разбора данной грамматики. Лексический анализатор реализован функцией scan(), которая формирует во внешней переменной lex тип очередной распознанной на входе лексемы. Для простоты в программе сканер просто читает входной символ. Идентификаторы представлены одним символом i, а константы 1.

Если начало строки распознано как правильное выражение, то анализ с успехом заканчивается. Ошибка выдается сразу, как распознается и возвращается 0.

Разбор строки (1) получается таким:

E=>TE'=>(E)E'=>(TE')E'=>(1E')E'=>(1ε)E'=>(1ε)ε=>(1)

при входе: (1)

Из первого пункта описания того, что должна делать процедура при рекурсивном спуске, следует, что данный метод применим, если для любого нетерминала A и различных A->x и A->y выполняется: FIRST(x) и FIRST(y) не

пересекаются! Оказывается этого еще не достаточно, чтобы использовать для грамматики предсказывающий разбор. Например, при анализе продукции

A->xBCy

и входном символе *a* может оказаться, что *a* входит как в FIRST(B) так и в FIRST(C), но и ε входит в FIRST(B). Тогда становится непонятно, в каком направлении вести дальнейший разбор.

Чтобы решить данную проблему введем еще одно множество, связанное только с нетерминалом. Это множество включает в себя все терминальные символы, которые могут стоять сразу за данным нетерминалом в какой-либо сентенциальной форме: FOLLOW(A)=*a* | S*=>xAay. Данное множество строится следующим образом:

1) если A->xBy, то все не ε символы из FIRST(y) входят в FOLLOW(B);

2) если A->xB или A->xBy, но ε принадлежит FIRST(y), то FOLLOW(A) входит в FOLLOW(B).

Для рассматриваемой грамматики имеем:

FOLLOW(E')=FOLLOW(E)={)}

FOLLOW(T)=FIRST(E')иFOLLOW(E')={+,)}

Обратим внимание, что на основе множеств FIRST и FOLLOW строится нерекурсивный (табличный) предсказывающий разбор.

Вторым условием применимости предсказывающего разбора к грамматике является то, что для любого ε-порождающего нетерминала его множества FIRST и FOLLOW не должны пересекаться (это условие выполнялось для нашей грамматики). Такая грамматика является LL(1) грамматикой.

Удобная с точки зрения анализа, такая грамматика вызывает трудности при ее построении. К тому же простые преобразования с целью избавления от леворекурсивности и неоднозначности (левая факторизация) приводят к разрастанию такой грамматики.

Невозможно и в целом ответить на вопрос о существовании для произвольной грамматики эквивалентной LL(1). Поэтому чаще всего такие грамматики применяют при распознавании операторов управления, а арифметические выражения распознают на базе операторного предшествования.

Обработка ошибок в предсказывающем разборе

Ошибка возникает, тогда, когда при разборе соответствующего не ε-порождающего нетерминала входная лексема не входит в его набор FIRST.

Для ε-порождающего нетерминала в таком случае берется как раз ε-продукция.

Стратегия переполоха для обработки ошибки основывается на наборе синхронизирующих символов, до которых все остальные пропускаются. Выбор такого набора сильно определяет и эффективность данной стратегии. Ниже приводятся несколько правил для построения синхронизирующего набора.

1. Положить в набор для нетерминала A все символы из FOLLOW(A). Но это множество может быть и пустым. Кроме того, ';' есть конец оператора в языке C, поэтому идущее далее ключевое слово не попадет в FOLLOW(A), а будет пропущено при отсутствии ';'.

2. Можно воспользоваться иерархической структурой языка, которая часто имеет место. Например, выражение встречается внутри оператора, оператор - внутри блока и т.д. Можно добавить к синхронизирующему множеству более младшей конструкции символы, с которых могут начинаться более старшие конструкции.

Например - ключевые слова, с которых могут начинаться операторы, можно добавить к синхронизирующему множеству для нетерминала, порождающего выражения.

3. Если добавить в синхронизирующее множество для A символы из FIRST(A), то, возможно, удастся возобновить разбор, если один из них встретится на входе.

В приведенных программах рекурсивного спуска использовалась процедура реакции на синтаксические ошибки error(). В простейшем случае эта процедура выдает диагностику и завершает работу анализатора. Но можно попытаться некоторым разумным образом продолжить работу.

Для разбора сверху вниз можно предложить следующий простой алгоритм.

Если в момент обнаружения ошибки на верхушке магазина оказался нетерминальный символ N и для него нет правила, соответствующего входному символу, то сканируем вход до тех пор, пока не встретим символ либо из FIRST(N), либо из FOLLOW(N). В первом случае разворачиваем N по соответствующему правилу, во втором - удаляем N из магазина.

Если на верхушке магазина терминальный символ, то можно выкинуть все терминальные символы с верхушки магазина вплоть до первого (сверху) нетерминального символа и продолжать так, как это было описано выше.

4.4. Восходящий синтаксический анализ. Алгоритмы типа "перенос-свертка"

Основа

В процессе разбора снизу-вверх типа сдвиг-свертка строится дерево разбора входной строки, начиная с листьев (снизу) к корню (вверх). Этот процесс можно рассматривать как "свертку" строки w к начальному символу грамматики. На каждом шаге свертки подстрока, которую можно сопоставить правой части некоторого правила вывода, заменяется символом левой части этого правила вывода, и если на каждом шаге выбирается правильная подстрока, то в обратном порядке прослеживается правосторонний вывод.

Рассмотрим грамматику арифметических выражений. Строка a+b*c может быть сведена к S. В строке a+b*c ищется подстрока, которую можно сопоставить с правой частью некоторого правила вывода. Этому удовлетворяют подстроки a, b и c. Если выбрать самое левое a и заменить его на F - левую часть правила F->id, то получим строку F+b*c.

Теперь правой части того же правила можно сопоставить подстроки b и c. Эти свертки представляют собой в обратном порядке правосторонний вывод:

E->E+T->E+T*F->E+T*c->E+F*c->E+b*c->T+b*c->F+b*c->a+b*c

E -> E + T		a+b*c
E -> T		F+b*c
T -> T*F		T+b*c
T -> F		E+b*c
F -> id	E+F*c	
		E+T*c
		E+T*F
		E+T
		E
а)		б)

Рис. 4.3.

Подстрока сентенциальной формы, которая может быть сопоставлена правой части некоторого правила вывода, свертка по которому к левой части правила соответствует одному шагу в обращении правостороннего вывода, называется основой строки. В приведенном выше выводе основы подчеркнуты. Самая левая подстрока, которая сопоставляется правой части некоторого правила вывода A->v, не обязательно является основой,

поскольку свертка по правилу A->v может дать строку, которая не может быть сведена к аксиоме. Если, скажем, в примере 3.5 заменить a на F и b на F, то получим строку F+F*c, которая не может быть сведена к S.

Формально, основа правой сентенциальной формы z - это правило вывода A->v и позиция в z, в которой может быть найдена строка v такие, что в результате замены v на A получается предыдущая сентенциальная форма в правостороннем выводе z. Таким образом, если S=>*uAw=>uvw, то A->v в позиции, следующей за u, это основа строки uvw. Строка w справа от основы содержит только терминальные символы.

Вообще говоря, грамматика может быть неоднозначной, поэтому не единственным может быть правосторонний вывод uvw и не единственной может быть основа. Если грамматика однозначна, то каждая правая сен тенциальная форма грамматики имеет в точности одну основу.

Замена основы в сентенциальной форме на нетерминал левой части называется отсечением основы. Обращение правостороннего вывода может быть получено с помощью повторного применения отсечения основы, начиная с разбираемой строки w. Если w - слово в рассматриваемой грамматике, то w=Zn, n-я правая сентенциальная форма еще неизвестного правого вывода

S => Z0 => Z1 => Z2 => ... => Zn-1 => Zn =w

Чтобы восстановить этот вывод в обратном порядке, выделяем основу Vn в Zn и заменяем Vn на левую часть некоторого правила вывода An -> Vn, получая (n-1)-ю правую сентенциальную форму Zn-1. Затем повторяем этот процесс, т.е. выделяем основу Vn-1 в Zn-1 и сворачиваем эту основу, получая правую сентенциальную форму Zn-2.

Если, повторяя этот процесс, мы получаем правую сентенциальную форму, состоящую только из начального символа S, то останавливаемся и сообщаем об успешном завершении разбора. Обращение последовательности правил, использованных в свертках, есть правый вывод входной строки.

Таким образом, главная задача анализатора типа сдвиг-свертка - это выделение и отсечение основы.

4.5. LR(k)-анализаторы

В названии LR(k) символ L означает, что разбор осуществляется слева-направо, R - что строится правый вывод в обратном порядке, k - число входных символов, на которые заглядывает вперед анализатор при разборе. Когда k опущено, имеют в виду 1.

LR-анализ привлекателен по нескольким причинам:

- LR-анализ - наиболее мощный метод анализа без возвратов типа сдвиг-свертка;

- LR-анализ может быть реализован довольно эффективно;

- практически LR-анализаторы могут быть построены для всех конструкций языков программирования;

- класс грамматик, которые могут быть проанализированы LR-методом, строго включает класс грамматик, которые могут быть анализированы предсказывающими анализаторами (сверху вниз типа LL).

Схематически структура LR-анализатора следующая. Он состоит из входа, выхода, магазина, управляющей программы и таблицы анализа, которая имеет две части - действий и переходов. Управляющая программа одна и та же для всех анализаторов, разные анализаторы различаются таблицами анализа. Программа анализатора читает символы из входного буфера по одному за шаг.

В процессе анализа используется магазин, в котором хранятся строки вида S0X1S1X2S2...XmSm (Sm - верхушка магазина). Каждый Xi - символ грамматики (терминальный или нетерминальный), а Si - символ, называемый состоянием.

Каждый символ состояния выражает информацию, содержащуюся в магазине ниже него, а комбинация символа состояния на вершине магазина и текущего входного символа используется для индексации таблицы анализа и определяет решение о сдвиге или свертке.

В реализации символы грамматики не обязательно должны размещаться в магазине. Однако их использование удобно для упрощения понимания поведения LR- анализатора.

Таблица анализа состоит из двух частей: действия (action) и переходов (goto). Начальное состояние этого ДКА - это состояние, помещенное на верхушку магазина LR-анализатора в начале работы.

Конфигурация LR-анализатора - это пара, первая компонента которой - содержимое магазина, а вторая - не просмотренный вход:

(S0 X1 S1 X2 S2 ... Xm Sm, Ai Ai+1 ... An $)

Эта конфигурация соответствует правой сентенциальной форме

X1 X2 ... Xm Ai Ai+1 ... An

Префиксы правых сентенциальных форм, которые могут появиться в магазине анализатора, называются активными префиксами.

Активный префикс - это такой префикс правой сентенциальной формы, который не переходит правую границу основы этой формы.

Очередной шаг анализатора определяется текущим входным символом Ai и символом состояния на верхушке магазина Sm.

Элемент таблицы действий action[Sm,Ai] для состояния Sm и входа Ai, может иметь одно их четырех значений:

1) shift S, сдвиг, где S - состояние,

2) reduce A->w, свертка по правилу грамматики A -> w,

3) accept, допуск,

4) error, ошибка.

Конфигурации, получающиеся после каждого из четырех типов шагов, следующие

1. Если action[Sm,Ai]=shift S, то анализатор выполняет шаг сдвига, переходя в конфигурацию

(S0 X1 S1 X2 S2 ... Xm Sm Ai S, Ai+1 ... An $)

В магазин помещаются как входной символ Ai, так и следующее состояние S, определяемое action[Sm,Ai]. Текущим входным символом становится Ai+1.

2. Если action[Sm,Ai]=reduce A->w, то анализатор выполняет свертку, переходя в конфигурацию

(S0 X1 S1 X2 S2 ... Xm-r Sm-r A S, Ai Ai+1 ... An $)

где S=goto[Sm-r,A] и r - длина w, правой части правила вывода.

Функция goto таблицы анализа, построенная по грамматике G, - это функция переходов детерминированного конечного автомата, распознающего активные префиксы G.

Анализатор сначала удаляет из магазина 2r символов (r символов состояния и r символов грамматики), так что на верхушке оказывается состояние Sm-r.

Затем анализатор помещает в магазин как A - левую часть правила вывода, так и S - содержимое таблицы goto[Sm-r,A]. На шаге свертки текущий входной символ не меняется.

Для LR- анализаторов Xm-r+1 ... Xm - последовательность символов грамматики, удаляемых из магазина, всегда соответствует w - правой части правила вывода, по которому делается свертка.

После осуществления шага свертки генерируется выход LR-анализатора, т.е. исполняются семантические действия, связанные с правилом, по которому делается свертка, например печатаются номера правил, по которым делается свертка.

3. Если action[Sm,Ai]=accept, то разбор завершен.

4. Если action[Sm,Ai]=error, анализатор обнаружил ошибку, то выполняются действия по диагностике и восстановлению.

Ниже приведен алгоритм LR-анализа. Все LR-анализаторы ведут себя одинаково. Разница между ними заключается в различном содержании таблиц действий и переходов.

Алгоритм LR-анализа

```
loop Пусть S - состояние на верхушке магазина;
    if action[S,InSym]=shift S'
    then поместить InSym и затем S'
        на верхушку магазина;
        прочитать в InSym следующий
        входной символ
    else if action[S,InSym]=reduce N->w
          then удалить из магазина
            2*|w| символов;
            пусть теперь на верхушке магазина
            состояние S';
            поместить на верхушку магазина N, а
            затем состояние goto[S',InSym];
            вывести правило N->w
          else if action[S,InSym]=accept
               then return
               else error()
              end
          end
     end
end;
```

5. Виды ошибок и их обработка компилятором

Одной из главных функций для любого компилятора является выявление и обработка ошибок, неминуемо встречающихся в исходном тексте программы.

С точки зрения серьезности ошибки можно разделить на

- предупреждающие (например, слишком длинный идентификатор), которые позволяют продолжить компиляцию программы и сгенерировать объектный код;
- серьезные (например, отсутствует ';'), которые позволяют продолжить компиляцию, но объектный код не генерируется;
- фатальные (например, у компилятора не хватило памяти для работы), которые прерывают компиляцию программы.

Ошибки можно также классифицировать по их виду. Здесь обычно выделяются лексические (например, неверный символ), синтаксические (например, отсутствует операнд), семантические (например, неверный тип операнда), логические (программа зацикливается).

5.1. Конфликты типа перенос-свертка

Если грамматика не является LR(1), то анализатор типа сдвиг- свертка для нее может достигнуть конфигурации, в которой он, зная содержимое магазина и следующий входной символ, не может решить, делать ли сдвиг или свертку (конфликт сдвиг/свертка), или не может решить, какую из нескольких сверток применить (конфликт свертка/свертка). В частности, неоднозначная грамматика не может быть LR.

Пример. Рассмотрим следующую грамматику оператора if-then-else:

```
St -> if Ex then St
    | if Ex then St else St
    | ...
```

Если анализатор типа сдвиг-свертка находится в конфигурации

Магазин	Вход
... if Ex then St	else ... $

то нельзя определить, является ли if Ex then St основой, вне зависимости от того, что лежит в магазине ниже. Это конфликт сдвиг/свертка. В зависимости

от того, что следует на входе за else, правильной может быть свертка по St -> if Ex then St или сдвиг else, а затем разбор другого St и завершение альтернативы if Ex then St else St.

Таким образом нельзя сказать, нужно ли в этом случае делать сдвиг или свертку, так что грамматика не LR(1).

Данная грамматика может быть преобразована к LR(1)-виду следующим образом:

St -> CondSt | UnCondSt

CondSt -> IfThenSt | IfThenElseSt

FullSt -> IfThenElseSt | UnCondSt

IfThenElseSt -> if Ex then FullSt else St

IfThenSt ->if Ex then St

5.2. Восстановление после синтаксических ошибок

Синтаксическая ошибка возникает, когда анализатор попадает в состояние, не допустимое ни в одной программе, синтаксически правильной для заданного языка. При возникновении ошибки, анализатор должен правильно ее распознать, выдать соответствующее сообщение, изменить свое состояние так, чтобы продолжить дальнейший анализ программы. Такой выход из ошибочной ситуации должен быть корректным, с тем чтобы минимизировать количество порождаемых наведенных (фиктивных) сообщений.

Существует несколько общих стратегий обработки ошибочной ситуации в программе.

1. "Корректировка". Здесь выполняется удаление, вставка или замена символа, например, при отсутствии операции можно подставить одну из допустимых (сложение).

2. "Паника". Анализатор игнорирует недопустимый и идущие следом символы пока не встретится один из так называемых опорных символов, например, ';' или '}'. Опорные символы для каждого нетерминала выбираются заранее по грамматике.

3. "Правила". Выполняется расширение исходной грамматики за счет правил, описывающих наиболее часто встречающиеся ошибки. Тем самым ошибка перестает быть исключительной ситуацией.

Для восходящего анализа одним из простейших методов является следующий. При синтаксической ошибке просматриваем магазин от вершины, пока не найдем состояние s с переходом на выделенный нетерминал A.

Затем сканируются входные символы, пока не будет найден такой, который допустим после A. В этом случае на верхушку магазина помещается состояние goto[s,A] и разбор продолжается. Для нетерминала A может иметься несколько таких вариантов.

Обычно A - это нетерминал, представляющий одну из основных конструкций языка, например оператор. Тогда s - это, например, точка с запятой или end.

При более детальной проработке реакции на ошибки можно в каждой пустой клетке анализатора поставить обращение к своей подпрограмме. Такая подпрограмма может вставлять или удалять входные символы или символы магазина, менять порядок входных символов.

6. Семантический анализ и внутреннее представление программы

Синтаксический анализ позволяет распознать входную цепочку символов на соответствие определенным правилам. Однако, только синтаксического анализа недостаточно для организации процесса компиляции или интерпретации.

Необходим набор действий, определяющих: семантическую (смысловую) нагрузку объектов программы; отделение синтаксиса от семантики; организация контекстного анализа полученного промежуточного представления (без синтаксиса) перед генерацией кода или интерпретацией.

В процессе компиляции программы часто используют промежуточное представление программы, предназначенное прежде всего для удобства генерации кода и/или проведения различных оптимизаций программы. Сама форма представления зависит от целей его использования.

6.1. Атрибуты и их анализ

Выходом синтаксического анализа является представление программы в виде дерева разбора, описывающего иерархическое строение составляющих её вхождений понятий и лексем.

Процесс дальнейшей компиляции связан с систематическим обходом этого представления программы, порождающим строку на выходном языке. При этом предполагается, что результат обработки некоторого вхождения понятия полностью определяется непосредственными связями вершины, соответствующей этому вхождению, причем связи в дереве рассматриваются либо по предшествованию, либо по следованию. Такая локальность обработки достигается за счет снабжения атрибутами вершин дерева разбора программы.

В атрибутном дереве программы вершины соответствуют тем же понятиям, что были выделены в дереве разбора при синтаксическом анализе, но снабжены дополнительной информацией о тех свойствах (атрибутах) соответствующих вхождений понятий и лексем, в которых учитывается ещё и контекст данного его вхождения.

Необходимость учета контекста для построения выходной строки может быть продемонстрирована хотя бы на таком простом примере. Анализируя на основании контекстно-свободной грамматики фрагмент исходного текста программы "; *a*=*b*;" обнаружим, что имеем дело с оператором присваивания,

обе части которого есть идентификаторы. Данная информация ещё не дает однозначного ответа на вопрос: совпадают ли типы переменных *a* и *b* или значение *b* предварительно должно быть преобразовано к типу переменной *a*.

Вхождению понятия или лексемы, выделенному в дереве разбора программы, может быть сопоставлена совокупность атрибутов и их значений. При этом набор атрибутов зависит от того, какой конструкции языка соответствует это вхождение, а значения атрибутов зависят ещё и от вида поддерева, корнем которого является рассматриваемая вершина грамматического дерева.

Например, для такого типа лексемы, как идентификатор, обычно определён атрибут "вид", значением которого могут быть признаки метки, простой переменной, процедуры и пр. Если значение атрибута "вид" для идентификатора совпадает с признаком переменной, то в этом случае для него определён атрибут "тип", значениями которого могут являться признаки целого, вещественного, логического и т.п.

Для любого оператора может быть определён атрибут "помеченность" со значениями "не имеет метки", "имеет метку, на которую есть переход", "имеет метку, на которую нет перехода". Для выражения обычно определены такие атрибуты, как "тип" (как для простой переменной) и "число промежуточных переменных для реализации выражения" со значениями, являющимися натуральными числами.

Нахождение значений атрибутов для вхождений понятий программы - это построение атрибутного дерева и этот процесс называется семантическим (контекстным) анализом. Для части лексем определение значений атрибутов может быть осуществлено на этапе лексического или синтаксического анализа, однако для большинства конструкций может потребоваться обособленный этап контекстного анализа.

6.2. Представление программы

Наиболее часто используемыми формами внутреннего представления являются ориентированный граф (или, в частности, абстрактное синтаксическое дерево), триады, тетрады, префиксная или постфиксная запись. Промежуточное представление программы может в различной степени быть близким либо к исходной программе, либо к машине. Например, представление может содержать адреса переменных, и тогда оно уже не может быть перенесено на другую машину.

С другой стороны, внутреннее представление может содержать раздел описаний программы, и тогда информацию об адресах можно извлечь из

обработки описаний. В то же время ясно, что первое более эффективно, чем второе.

Некоторые формы промежуточного представления лучше годятся для различного рода оптимизаций (например, косвенные триады - для перемещения кода), некоторые - хуже (например, префиксная запись для этого плохо подходит).

Ориентированный граф

Простейшей формой промежуточного представления является синтаксическое дерево программы. Более полную информацию о входной программе дает ориентированный ациклический граф (ОАГ), в котором в одну вершину объединены вершины синтаксического дерева, представляющие общие подвыражения.

Древо организуется следующим образом. Операндам, которые в простейшем случае являются переменными или константами, соответствуют висячие вершины (листья). Операции соответствует вершина - отец, у которого вершины - сыновья соответствуют операндам данной операции. Фактически, листья - это указатели на таблицу имен. Каждая вершина кодируется записью с полем для операции и полями для указателей на потомков. Вершины размещены в массиве записей и индексом (входом) вершины служит указатель на нее.

Трехадресный код

Трехадресный код- это последовательность операторов вида x:= y op z, где x,y и z - имена, константы или сгенерированные компилятором временные объекты. Здесь op - двуместная операция, например операция плавающей или фиксированной арифметики, логическая или побитовая. В правую часть может входить только один знак операции.

Составные выражения должны быть разбиты на подвыражения, при этом могут появиться временные имена (переменные). Смысл термина "трехадресный код" в том, что каждый оператор обычно имеет три адреса: два для операндов и один для результата.

Трехадресный код - это линеаризованное представление синтаксического дерева или ОАГ, в котором явные имена соответствуют внутренним вершинам дерева или графа. Например, выражение x+y*z может быть компилировано в последовательность операторов

```
t1=y*z
t2=x+t1
```

где t1 и t2 - имена, сгенерированные компилятором.

В виде трехадресного кода представляются не только двуместные операции, входящие в выражения. В таком же виде представляются операторы управления программы и одноместные операции. В этом случае некоторые из компонент трехадресного кода могут не использоваться. Например, условный оператор

if A>B then S1 else S2

может быть представлен следующим кодом:

```
t=A-B
JLE t,S2
S1
S2
```

Здесь JLE - двуместная операция условного перехода, не вырабатывающая результата. Переход выполняется, когда значение $t \leq 0$.

Разбиение арифметических выражений и операторов управления делает трехадресный код удобным при генерации машинного кода и оптимизации. Использование имен промежуточных значений, вычисляемых в программе, позволяет легко переупорядочивать трехадресный код.

Трехадресный код - это абстрактная форма промежуточного кода. В реализации трехадресный код может быть представлен записями с полями для операции и операндов. Рассмотрим три реализации трехадресного кода: тетрады, триады и косвенные триады.

Тетрада - это запись с четырьмя полями, которые будем называть op, arg1, arg2 и result. Поле op содержит код операции. В операторах с унарными операциями типа x:=-y или x:=y arg2 не используется.

В некоторых операциях (типа "передать параметр") могут не использоваться ни arg2, ни result. Условные и безусловные переходы помещают в result метку перехода.

Обычно содержимое полей arg1, arg2 и result - это указатели на входы таблицы символов для имен, представляемых этими полями. Временные имена вносятся в таблицу символов по мере их генерации.

Чтобы избежать внесения новых имен в таблицу символов, на временное значение можно ссылаться, используя позицию вычисляющего его оператора. В этом случае трехадресные операторы могут быть представлены записями только с тремя полями: op, arg1 и arg2. Поля arg1 и arg2 - это либо указатели на таблицу символов (для имен, определенных программистом, или констант), либо указатели на триады (для временных значений). Такой способ представления трехадресного кода называют *триадами*.

Триады соответствуют представлению синтаксического дерева или ОАГ с помощью массива вершин.

Числа в скобках - это указатели на триады, а имена - это указатели на таблицу символов. На практике информация, необходимая для интерпретации различного типа входов в поля arg1 и arg2, кодируется в поле op или дополнительных полях.

Для представления триадами трехместной операции типа x[i]:=y требуется два входа, как это показано на рис. 6.1 а), представление x:=y[i] двумя операциями показано на рис. 6.1 б).

	op	arg1	arg2
(0)	[]=	x	I
(1)	=	(0)	Y

а) x[i]:=y

	Op	arg1	arg2
(0)	=[]	y	i
(1)	=	x	(0)

б) x:=y[i]

Рис. 6.1

Трехадресный код может быть представлен не списком триад, а списком указателей на них. Такая реализация обычно называется косвенными триадами.

При генерации объектного кода каждой переменной, как временной, так и определенной в исходной программе, назначается память периода исполнения, адрес которой обычно хранится в таблице генератора кода. При использовании тетрады этот адрес легко получить через эту таблицу.

Более существенно преимущество тетрад проявляется в оптимизирующих компиляторах, когда может возникнуть необходимость перемещать операторы. Если перемещается оператор, вычисляющий x, не требуется изменений в операторе, использующем x. В записи же триадами перемещение оператора, определяющего временное значение, требует изменения всех ссылок на этот оператор в массивах arg1 и a rg2. Из-за этого тройки трудно использовать в оптимизирующих компиляторах.

В случае применения косвенных триад оператор может быть перемещен переупорядочиванием списка операторов. При этом не надо менять указатели на op, arg1 и arg2. Этим косвенные триады похожи на тетрады. Кроме того, эти два способа требуют примерно одинаковой памяти. Как и в случае простых триад, при использовании косвенных триад выделение памяти для временных значений может быть отложено на этап генерации кода.

По сравнению с тетрадами при использование косвенных триад можно сэкономить память, если одно и то же временное значение используется более одного раза.

Линеаризованные представления

В качестве промежуточных представлений весьма распространены линеаризованные представления. Линеаризованное представление позволяет относительно легко хранить промежуточное представление на внешней памяти и обрабатывать его в порядке чтения.

Самая распространенная форма линеаризованного представления - это запись дерева либо в порядке его обхода снизу-вверх (постфиксная запись, или обратной польской), либо в порядке обхода его сверху-вниз (префиксная запись, или прямой польской).

Таким образом, постфиксная запись (обратная польская) - это список вершин дерева, в котором каждая вершина следует непосредственно за своими потомками. Для выражения a=b*(-c)+b*c постфиксная запись может быть представлена следующим образом:

a b c - * b c * + =

В постфиксной записи вершины синтаксического дерева явно не присутствуют. Они могут быть восстановлены из порядка, в котором следуют вершины и из числа операндов соответствующих операций. Восстановление вершин аналогично вычислению выражения в постфиксной записи с использованием стека. В префиксной записи сначала указывается операция, а затем ее операнды. Например, для приведенного выше выражения имеем:

= a + * b - c * b - c

Рассмотрим детальнее одну из реализаций префиксного представления - Лидер. Лидер - это аббревиатура от 'ЛИнеаризованное ДЕРево'. Это машинно-независимая языково-ориентированная префиксная запись. В этом представлении сохраняются все объявления и каждому из них присваивается свой уникальный номер, который используется для ссылки на объявление. Рассмотрим пример (рис. 6.2).

```
module M;
var X,Y,Z: integer;
procedure DIF(A,B:integer):integer;
  var R:integer;
  begin R:=A-B;
      return(R);
```

```
    end DIF;
  begin Z:=DIF(X,Y);
  end M.
```

Рис. 6.2

Соответствующий образ в Лидере изображен на рис. 6.3.

```
program 'M'
var int
var int
var int
procbody proc int int end int
  var int
  begin assign var 1 7 end
          int int mi par 1 5 end par 1 6 end
      result 0 int var 1 7 end
      return
  end

begin assign var 0 3 end int
    icall 0 4 int var 0 1 end int var 0 2 end end
end
```

Рис. 6.3

Рассмотрим его более детально:

program 'M' Имя модуля используется для редактора связей

var int Это образ переменных var X,Y,Z:integer;

var int переменным X,Y,Z присваиваются номера

var int 1,2,3 на уровне 0

procbody proc Объявление процедуры с двумя

int int end целыми параметрами, возвращающей целое.

int Процедура получает номер 4 на уровне 0 и параметры имеют номера 5, 6 на уровне 1.

var int Локальная переменная R имеет номер 7 на уровне 1

```
begin          Начало тела процедуры
assign         Оператор присваивания
var 1 7 end    Левая часть присваивания (R)
int            Тип присваиваемого значения
int mi         Целое вычитание
par 1 5 end    Уменьшаемое (A)
par 1 6 end    Вычитаемое (B)
result 0       Результат процедуры уровня 0
int            Результат имеет тип целый
var 1 7 end    Результат - переменная R
return         Оператор возврата
end            Конец тела процедуры

begin          Начало тела модуля
assign         Оператор присваивания
var 0 3 end    Левая часть - переменная Z
int            Тип присваиваемого значения
icall 0 4      Вызов локальной процедуры DIF
int var 0 1 end  Фактические параметры X и Y
int var 0 2 end
end            Конец вызова
end            Конец тела модуля
```

7. Распределение памяти в компиляторе

Распределение памяти может быть отдельной фазой, выполняемой перед фактической генерацией кода, т.е. при получении машинного кода, относительные (или абсолютные) адреса операндов вставляются в адресные поля машинных команд. Другая возможность заключается в том, чтобы отложить распределение памяти до стадии трансляции с языка ассемблера и предоставить генератору кода возможность работать только с символическими адресами. В этом случае выходом генератора является программа на языке ассемблера, и распределение памяти выполняется так же, как и в ассемблере. Независимо от используемого метода память должна быть выделена под все переменные, явно или неявно определенные в исходной программе, под все временные ячейки, остающиеся в промежуточном представлении, и под все появляющиеся в программе имена. Кроме того, должны быть выполнены начальные установки для соответствующих переменных.

В языках высокого уровня распределением памяти в значительной степени управляет пользователь. Раз заданы определения данных, единственно важным решением, остающимся на долю компилятора, является вопрос, где должны храниться эти переменные.

Память распределяется либо в области, непосредственно следующей за рассматриваемой подпрограммой, либо в отдельной части программы, где собираются вместе данные всех независимых сегментов, отделяя таким образом фиксированную и переменную части программы. В таких допускающих блочную структуру языках, как Си и Паскаль, необходимы специальные средства для отслеживания определений переменных в различных блоках.

В то время как эффективное использование памяти для переменных, описанных в исходной программе, является задачей программиста, временные переменные, требующиеся для промежуточных результатов и определяемые компилятором, должны распределяться компилятором. То же самое справедливо, как было показано в предыдущем разделе, для использования общих регистров и для эффективного использования базовых и индексных регистров. Эффективность в смысле распределения памяти обычно означает минимизацию количества памяти, используемой во время выполнения программы.

Классы памяти

Существуют следующие основные типы управляемой программистом памяти:

- статическая (внутренняя и внешняя),
- автоматическая,
- управляемая (внутренняя и внешняя),
- базированная.

Поскольку компилятор только следует заданным ему описаниям класса, для программиста важно знать основные преимущества различных типов памяти.

Статическая память распределяется во время компиляции и резервируется для переменных на все время работы программы.

Внутренние переменные могут использоваться только в процедурах или сегментах, в которых они описаны. Не существует никаких соотношений между этими переменными, за исключением тех, которые определены программистом, например, с помощью оператора EQUIVALENCE (в ФОРТРАНе) или фразы REDEFINES (в КОБОЛе). Память для внешних статических переменных, таких, как переменные EXTERNAL в Си, или переменные, определенные в LINKAGE STORAGE SECTION программы на КОБОЛе, распределяется перед выполнением и существует все время независимо от того, используются эти переменные или нет. Различие между внешней и внутренней статической памятью заключается в том, что на переменные, принадлежащие первому классу, можно ссылаться не только в процедуре, в которой они впервые определены и в которой под них выделена память, но также и в других частях программы. Все языки высокого уровня допускают определение таких классов памяти.

Автоматическая память всегда внутренняя. На нее можно ссылаться только в процедуре, в которой она определена, и распределение ее производится при выполнении программы при активировании процедуры. Это приводит к более эффективному использованию памяти. Пространство резервируется только при входе в процедуру и немедленно освобождается при завершении процедуры. Очевидным примером использования преимуществ автоматической памяти является описание массивов с переменными размерами. Если допускается только статическая память, пространство должно быть отведено постоянно для максимально возможного массива. Если компилятор может обрабатывать класс автоматической памяти, массив описан в отдельной подпрограмме или процедуре и его размер определяется передачей в качестве параметра при вызове процедуры, то будет зарезервировано только фактически необходимое для массива пространство памяти даже в случае, если подпрограмма разрабатывалась для максимально возможного размера. Этот метод часто используется программистами, работающими на Си, для

сохранения памяти без потери общности для подпрограмм, использующих массивы переменного размера.

Управляемая, или *базированная,* память распределяется только при выполнении оператора ALLOC (в Си) и освобождается для других целей, если встречается оператор FREE. Это предоставляет программисту значительную гибкость в распределении памяти, но делает его полностью ответственным за эффективное использование памяти.

Классы памяти заносятся в таблицу идентификаторов синтаксическим анализатором и интерпретатором из явных или неявных описаний данных. Подпрограмма распределения памяти использует эти данные для присваивания адресов всем переменным каждого класса, чтобы обеспечить отведение всей требуемой памяти и корректность последующих ссылок на нее.

8. Оптимизация объектного кода

Любая оптимизация - это поиск оптимального, т.е. наилучшего, решения. Для программ под оптимизацией подразумевают замену заложенного в программе алгоритма на более эффективный по быстродействию или по использованию памяти. Здесь нахождение оптимума или очень трудно или совсем невозможно, поскольку существуют такие рекурсивные функции, что для любого алгоритма, их вычисляющего, всегда найдется другой, который для достаточно больших размеров входных данных работает в произвольное число раз быстрее исходного алгоритма. С этой точки зрения для программ вместо оптимизации следовало бы употреблять термин улучшение, что, правда, не исключает возможности находить оптимальный алгоритм в отдельных случаях.

Оптимизация объектного кода основана на промежуточном представлении программы. При этом оптимизация разделяется обычно на машинно-независимую и машинно-зависимую. К первой, например, относится выделение общих подвыражений, а ко второй - распределение переменных по регистровой памяти.

Машинно-независимая оптимизация

Она основана на эквивалентных преобразованиях внутреннего представления, например, тетрад. Рассмотрим некоторые распространенные виды такой оптимизации.

1) Выделение общих подвыражений на линейном участке программы (где нет передачи управления), например, во фрагменте программы

```
d=d+c*b;
a=d+c*b;
c=d+c*b;
```

можно только один раз вычислить c*b и дважды d+c*b.

Выделение общих подвыражений основывается на двух положениях.

Во-первых, поскольку на линейном участке переменной может быть несколько присваиваний, то при выделении общих подвыражений необходимо различать вхождения переменных до и после присваивания. Для этого каждая переменная снабжается номером. Вначале номера всех переменных устанавливаются равными 0. При каждом присваивании переменной ее номер увеличивается на 1.

Во-вторых, выделение общих подвыражений осуществляется при обходе дерева выражения снизу вверх слева направо. При достижении очередной

вершины (пусть операция, примененная в этой вершине, есть бинарная 'op'; в случае унарной операции рассуждения те же) просматриваем общие подвыражения, связанные с op.

Если имеется выражение, связанное с op и такое, что его левый операнд есть общее подвыражение с левым операндом нового выражения, а правый операнд - общее подвыражение с правым операндом нового выражения, то объявляем новое выражение общим с найденным и в новом выражении запоминаем указатель на найденное общее выражение.

Базисом построения служит переменная: если операндами обоих выражений являются одинаковые переменные с одинаковыми номерами, то они являются общими подвыражениями.

Если выражение не выделено как общее, оно заносится в список операций, связанных с op.

2) Удаление бесполезных операторов.

Бесполезные операторы - такие операторы, которые не влияют на результат работы программы. Чаще всего они возникают вследствие ошибочной логики, как например, в следующем фрагменте

```
goto L;
L: …
```

Или если есть участок программы, на который управление ни при каких исходных данных не будет передано.

3) Вычисление константных выражений (свёртка).

Операнды в таких выражениях могут быть как явными константами, так и именованными. Также возможно, к моменту анализа выражения все его операнды уже определены, например,

```
#define  pi  3,1415
…
const=2;
…
a=pi/const;
```

4) Нахождение инвариантов циклов.

Инвариантом цикла называется такой оператор, стоящий в теле цикла, операнды которого не зависят от переменной цикла, т.е. значение оператора не меняется во время выполнения цикла. Такой инвариант естественно поставить перед циклом, что позволит сэкономить время выполнение программы.

5) Упрощение логических выражений на основе булевой алгебры.

Машинно-зависимая оптимизация

Рассмотрим теперь некоторые простые правила распределения регистров при наличии общих подвыражений. Если число регистров ограничено, можно выбрать один из следующих двух вариантов.

1. При обнаружении общего подвыражения с подвыражением в уже просмотренной части дерева (и, значит, с уже распределенными регистрами) проверяем, расположено ли его значение на регистре.

Если да, и если регистр после этого не менялся, заменяем вычисление поддерева на значение в регистре. Если регистр менялся, то вычисляем подвыражение заново.

2. Вводим еще один проход. На первом проходе распределяем регистры. Если в некоторой вершине обнаруживается, что ее поддерево общее с уже вычисленным ранее, но значение регистра потеряно, то в такой вершине на втором проходе необходимо сгенерировать команду сброса регистра в рабочую память.

Выигрыш в коде будет, если стоимость команды сброса регистра + доступ к памяти в повторном использовании этой памяти не превосходит стоимости заменяемого поддерева. Поскольку стоимость команды MOVE известна, можно сравнить стоимости и принять оптимальное решение: то ли метить предыдущую вершину для сброса, то ли вычислять полностью поддерево.

9. Генерация объектного кода. Виды объектного кода

Задача генератора кода - построение эквивалентной машинной программы по программе на входном языке. Обычно в качестве входного для генератора кода служит некоторое промежуточное представление программы. В свою очередь, генерация кода состоит из ряда специфических, относительно независимых подзадач: распределение памяти (в частности, распределение регистров), выбор команд, генерация объектного (или загрузочного) модуля.

Конечно, независимость этих подзадач относительна: например, при выборе команд нельзя не учитывать схему распределения памяти, и, наоборот, схема распределения памяти (регистров, в частности) необходимо ведет к генерации той или иной последовательности команд. Однако удобно и практично эти задачи все же разделять и при этом особо обращать внимание на их взаимодействие.

В какой-то мере схема генератора кода зависит от формы промежуточного представления. Ясно, что генерация кода из дерева отличается от генерации кода из триад, а генерация кода из префиксной записи отличается от генерации кода из ориентированного графа. В то же время все генераторы кода имеют и много общего и основные применяемые алгоритмы отличаются, как правило, только в деталях, связанных с используемым промежуточным представлением.

Техника генерации кода может основываться на однозначном соответствии структуры промежуточного представления и описывающей это представление грамматики. Для генерации более качественного кода может быть применен некоторый подход. Этот подход основан на понятии "сопоставления образцов": командам машины сопоставляются некоторые "образцы", вхождения которых ищутся в промежуточном представлении программы, и делается попытка "покрыть" промежуточную программу такими образцами. Если это удается, то по образцам восстанавливается программа уже в кодах.

Генерацию кода можно представить как результат выхода на соответствующее место из точек анализа допустимых вариантов. Каждый вызов такой процедуры раскрывается в цепочку кода объектной машины. Особенности этой цепочки определяются особенностями объектного языка.

Пример: c = a + b a, b, c - цепи

ассемблер PDP c - mem [0], a - mem [2], R5 - индекс mem b - mem [16].

```
MOV 2(R5), R'
ADD 16(R5), R'
MOV R', '(R5)
```

Виды объектного кода

Результатом работы компилятора является объектный код, который может быть представлен различным образом.

Во-первых, объектным кодом может быть последовательность абсолютных машинных команд. Адреса в таких командах стоят уже реальные и при загрузке кода в память он сразу может исполняться. Для абсолютного кода характерен малый размер и одномодульность программ. Чаще всего такой код генерируется для встроенных управляющих микропроцессорных устройств.

Наиболее распространенным видом объектного кода является последовательность перемещаемых машинных команд. На выходе компилятора формируется объектный модуль с командами, где, во-первых, могут быть неразрешенные ссылки на другие модули (вызов библиотечных подпрограмм), и, во-вторых, относительные адреса в машинных командах. Такой подход является гораздо более гибким, поскольку программа может компоноваться редактором связей операционной системы из нескольких объектных модулей, как сформированных программистом, так и библиотечных. Кроме того, код может быть расположен в оперативной памяти по разному, а относительные адреса у команд преобразуются в реальные (абсолютные) при загрузке кода в память на выполнение.

В качестве объектного кода может также выступать ассемблерный текст программы. Тем самым можно достичь некоторой машино-независимости компилятора, экономии памяти реализуя компилятор в несколько проходов, большей эффективности программы за счет более глубокой оптимизации на этапе ассемблирования.

Литература

1. *Ахо А., Ульман Дж.Д.* Теория синтаксического анализа, перевода и компиляции. - М.: Мир, 1978. Т.1. 616с.; Т.2. 488с.
2. *Aho A.V., Sethi R., Ullman J.D.* Compiles: Principles, Techniques and Tools, Stanford University, Stanford, California, 1988. 796p.
3. *Ахо А., Ульман Дж.Д.* Принципы машинного проектирования. М.: Мир, 1983. 352с.
4. *Грис Д.* Конструирование компиляторов для цифровых вычислительных машин. М.:Мир, 1975. 544с.
5. *Рейуорт-Смит В.Дж.* Теория формальных языков. Вводный курс. М.: Радио и связь, 1988. 128с.
6. *Соколов В.А.* Формальные языки и грамматики: Курс лекций. Яросл. гос. ун-т. Ярославль, 1998. 123 с.
7. *Вайнгартен Ф.* Компиляция языков программирования. М.: Мир, 1977. 192с.
8. *Касьянов В.И., Поттосин И.В.* Методы построения компиляторов. Новосибирск: Наука, 1986. 344с.
9. *Льюис Ф., Розенкрац Д., Стирнз Р.* Теоретические основы проектирования компиляторов. М.:Мир, 1979. 656с.
10. *Хантер Р.* Проектирование и конструирование компиляторов. М.: Финансы и статистика, 1984. 232с.
11. *Зелковиц М., Шоу А., Бынон Дж.* Принципы разработки программного обеспечения. М.: Мир, 1982.
12. *Бекхауз Р.Ч.* Синтаксис языков программирования. М.:Мир, 1986. 281с.
13. *Бек Л.* Введение в системное программирование. М.: Мир, 1988.
14. *Хэндрикс Д.* Компилятор языка Си для микро-ЭВМ. М.: Радио и Связь, 1989. 239с.
15. *Кнут Д.* Искусство программирования для ЭВМ, Т.3. М.: Мир, 1978.
16. *Агафонов В.Н.* Синтаксический анализ языков программирования. - Новосибирск: Изд-во НГУ, 1981.
17. *Данные* в языках программирования. М.: Мир, 1983.
18. *Касьянов В.Н.* Введение в теорию оптимизации программ. Новосибирск: Изд-во ВЦ СО АН СССР, 1985.

19. *Мейер Б., Бодуэн К.* Методы программирования. Т. 1, 2. М.: Мир, 1982.

20. *Маккиман У., Хорнинг Дж., Уортман Д.* Генератор компиляторов. М.: Статистика, 1980.

Бадин Николай Михайлович

МЕТОДЫ ТРАНСЛЯЦИИ

(план 1999 г.)

Корректор А.А. Антонова

Лицензия ЛР № 020319 от 30.12.96 г.
Подписано в печать 18.05.2000. Формат 60×84/16. Бумага тип.
Усл. печ. л. 3,5. Уч.-изд. л. 2,6. Тираж 100 экз. Заказ .

Оригинал-макет подготовлен в редакционно-издательском отделе ЯрГУ

Отпечатано на ризографе

Ярославский государственный университет им. П.Г. Демидова
150000 Ярославль, ул. Советская, 14.