Государственное образовательное учреждение
высшего профессионального образования
Сургутский государственный университет
Ханты-Мансийского автономного округа – Югры

Факультет автоматики и телекоммуникаций

Кафедра автоматики и компьютерных систем

Гришмановский Павел Валерьевич,
кандидат технических наук, доцент

Теория языков программирования и методы трансляции

Учебно-методическое пособие

Сургут, 2011

Содержание

Введение

Целью изучения дисциплины «Теория языков программирования и методы трансляции» является формирование у студентов систематизированных знаний в области построения трансляторов языков высокого уровня и организации вычислительного процесса средствами вычислительной техники.

Задачи преподавания дисциплины:

– студент должен знать основные этапы процесса трансляции, способы задания и описания искусственных языков;
– студент должен иметь представление об основных методах и подходах решения задач, связанных с конкретными шагами принципиальной схемы трансляции, об основных классах языков и о допустимых преобразованиях, совершаемых над грамматиками языков;
– студент должен приобрести навыки по изучению конструкций искусственного языка и применению оптимальных методов для решения поставленной задачи.

Освоение дисциплины заключается в изучении теоретического материала в соответствии с планом, представленным в настоящем учебно-методическом пособии. При этом может использоваться литература, указанная в списке рекомендуемых источников. Практическое освоение ряда разделов дисциплины осуществляется в результате выполнения комплекса лабораторных работ (лабораторного практикума).

Лабораторный практикум по дисциплине «Теория языков программирования и методы трансляции» направлен на изучение и практическое освоение основных алгоритмов грамматического разбора, методов их программной реализации и построения элементов трансляторов программ. Для выполнения каждой лабораторной работы необходимо изучение соответствующего раздела теоретической части курса.

Программную реализацию распознавателей и трансляторов, их отладку и тестирование необходимо выполнять с учетом всех методических рекомендаций, приведенных в описании каждой лабораторной работы. Для программной реализации может быть использована любая распространенная среда программирования. Для обеспечения возможности проверки результатов работы преподавателем, рекомендуется выполнять реализацию на языках C/C++ с использованием только стандартных библиотек и текстового интерфейса (тип приложения – консольное), что обеспечит хорошую переносимость на уровне исходного кода.

Так как программы, создаваемые в ходе выполнения лабораторных работ, выполняют преимущественно преобразование входного текста в выходной, рекомендуется имена входного и выходного файлов передавать в качестве параметров командной строки.

По каждой лабораторной работе оформляется отчет, который должен содержать следующие элементы:

1. Титульный лист.
2. Цель работы.
3. Задание на лабораторную работу.
4. Основная часть:
   – формализованное описание решения задачи;
   – другая информация (в соответствии с методическими рекомендациями по каждой лабораторной работе);
   – описание принципов программной реализации (желательна иллюстрация блок-схемами алгоритмов, схемами и диаграммами, фрагментами листинга и т.п.).
5. Выводы (в соответствии с методическими рекомендациями по каждой лабораторной работе).

К отчету для проверки преподавателем прикладывается архив в формате RAR, ZIP или 7z, содержащий:

- исходный код (или файлы проектов) разработанного приложения (распознавателя или транслятора) с подробными комментариями;
- входные файлы, использовавшиеся для тестирования, и соответствующие им выходные файлы;
- файл readme.txt с указанием среды разработки и, при необходимости, существенных особенностей настройки среды разработки, размещения файлов, компиляции и сборки проектов, использования параметров командной строки приложения и т.п.

Тематический план дисциплины

Тема 1. Введение.

Понятие языков и трансляторов. Группы языков и парадигмы программирования. Свойства искусственных языков. Аспекты стандартизации языков программирования.

Понятие транслятора. Виды трансляторов. Структура транслятора и этапы трансляции. Методы трансляции. Интерпретация и компиляция.

Тема 2. Формальные языки и грамматики.

Формальный язык. Способы задания языка. Понятие формальной грамматики. Способы задания грамматик.

Бэкусова нормальная форма (Нормальная форма Бэкуса–Наура). Грамматики Хомского. Иерархия грамматик Хомского и абстрактные машины.

Вывод и грамматический разбор. Стратегии синтаксического анализа. СУ-схемы. Транслирующие грамматики. Атрибутные транслирующие грамматики.

Тема 3. Трансляция на основе польской инверсной записи.

Определение польской инверсной записи. Алгоритм трансляции выражений. Алгоритм Дейкстры формирования польской инверсной записи. Трансляция выражений, содержащих переменные с индексом.

Трансляция операторов на основе польской инверсной записи. Оператор присваивания. Динамические деревья. Трансляция условных операторов.

Тема 4. Регулярные грамматики, языки и их свойства.

Преобразование контекстно-свободных и регулярных грамматик в автоматные. Диаграмма состояния автоматной грамматики. Конечный автомат-распознаватель автоматных языков. Построение конечного автомата по автоматной грамматике.

Лексический анализ. Сканер. Блок лексического анализа.

Тема 5. Контекстно-свободные грамматики.

Типы контекстно-свободных грамматик. Редуцированные и приведенные грамматики. Допустимые преобразования. Устранение левой рекурсии, факторизация, удаление несущественных символов. Условия порождения бесконечных языков. Распознаватель для контекстно-свободных языков. Нормальные формы Хомского и Грейбах.

Тема 6. Нисходящие стратегии синтаксического анализа.

Нисходящий распознаватель. Неформальное описание нисходящего анализа. Алгоритм нисходящего анализа.

Тема 7. Методы детерминированного синтаксического анализа на основе восходящей стратегии.

Восходящий распознаватель. Неформальное описание восходящего анализа. Алгоритм восходящего анализа. Отношения предшествования. Грамматики предшествования. Распознаватель для грамматик предшествования.

Построение отношений предшествования. Матрица предшествования. Построение функций предшествования с помощью алгоритма Флойда. Построение функций предшествования с помощью графа линеаризации. Метод простого предшествования.

Тема 8. LR(k)- и LL(k)-грамматики.

Особенности LR(k)- и LL(k)-грамматик и распознавателей. Правила подстановок Флойда-Эванса. Методы детерминированного синтаксического анализа на основе нисходящей стратегии. K-предсказывающий алгоритм разбора.

Тема 9. Контекстный анализ.

Атрибутная индукция. Генерация. Основные задачи процесса генерации. Оптимизация.

Лабораторная работа № 1
Создание простейшего распознавателя

Цель работы

Активировать имеющиеся навыки программирования, приобрести навыки автоматного программирования, получить практический опыт построения простейших распознавателей (лексических анализаторов).

Содержание работы

Построение распознавателя комментариев языков C и C++ с использованием автоматной модели.

Задание

1. Построить детерминированный конечный автомат распознавателя многострочных комментариев языка C и выполнить его программную реализацию.
2. Усовершенствовать распознаватель для корректного удаления комментариев языков C (многострочных) и C++ (однострочных).

Методические рекомендации к заданию

В соответствии с п.1 Задания, распознаватель должен, проведя лексический анализ входного файла, содержащего текст программы на языке C, сформировать выходной (новый) файл, содержащий тот же текст, но с удаленными из него комментариями вида `/*...*/`. Рассматриваются только комментарии языка C и не производится анализ каких-либо иных лексем. Следует учесть, что такие комментарии *не могут быть вложенными*.

Конечный автомат распознавателя проще всего представить в виде традиционного графа (диаграммы) состояний, в котором состояния представлены узлами, а переходы – дугами. Условием перехода из одного состояния в другое (событием, входным сигналом) является значение *одного* очередного символа, считанного из входного потока. При любом переходе из одного состояния в другое может производиться запись в выходной поток как последнего считанного символа, так и произвольного символа или любой цепочки символов. Таким образом, каждая дуга в графе должна быть маркирована и входным символом a, и выходной цепочкой символов γ, т.е. записью вида «a / γ». При этом для компактной и наглядной записи выражений *допускается*:

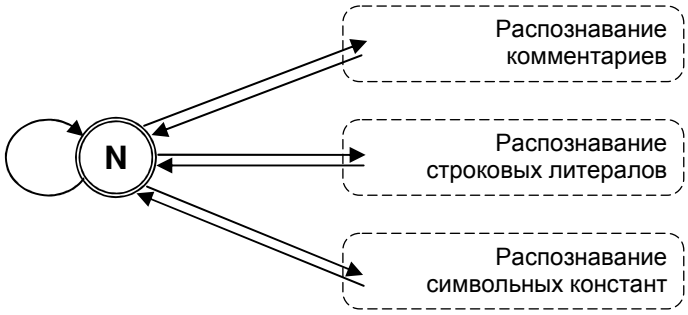
– указывать допустимые символы входной цепочки в виде множества, например, запись «$\{a, b, c, 0..9\}$ / …» будет означать принадлежность считанного символа указанному множеству, т.е. его равенство одному из элементов этого множества;
– использовать некоторое обозначение или метасимвол для входного символа в записи условия, например, запись «**c** $\notin \{0..9\}$ / …» будет означать, что считанный символ не является десятичной цифрой (является любым другим возможным символом);
– использовать некоторое обозначение для особых (типичных) ситуаций, например «**else**», «**default**» и т.п. для обозначения всех остальных случаев, т.е. образованных исключением из алфавита языка (полного множества возможных входных символов) всех тех символов, которые указаны в условиях перехода всех остальных дуг, исходящих из этого же состояния;
– использовать квантор $\forall$, если переход осуществляется при чтении любого возможного символа (немаркированная дуга означает безусловный переход из одного состояния в другое, при котором чтение очередного символа не производится);
– использовать некоторое обозначение или метасимвол для обозначения конца входной цепочки (при реализации программы – конца файла), например «**EOF**», «$\perp$» и т.п.;

– использовать некоторое обозначение или метасимвол для обозначения множества пустых символов – символов-разделителей, – таких как пробел, перевод строки, возврат каретки, табуляция и т.п., например «**B**», «**blank**», «□» и т.п.;
– не указывать выходную цепочку вместе с символом «/» или указывать «… / λ», если при данном переходе ничего не выводится в выходной поток;
– использовать некоторое обозначение для входного символа при его выводе в выходной поток, например: «**c** ∈ {*0..9*} / **c**» – символы входного потока, являющиеся цифрами, дублируются в выходной поток; «∀**c** / **c**» – дублирование любого считанного символа в выходной поток.

Для построения распознавателя необходим *детерминированный* и *полностью определенный* конечный автомат. Это означает, что в каждом состоянии для каждого возможного входного символа должен быть определен *ровно один* переход. Конечный автомат распознавателя будет являться *недетерминированным*, если хотя бы в одном состоянии некоторый считанный символ приводит к одновременному выбору двух или более направлений перехода, что невозможно реализовать без использования параллельных процессов, рекурсии и/или последующего обратного движения по входной цепочке (возврата). Конечный автомат распознавателя будет являться *определенным не полностью* или *недоопределенным*, если хотя бы в одном состоянии существует хотя бы один входной символ из множества возможных, для которого не задан переход, что приведет к неразрешимой ситуации при реализации распознавателя. По этой причине необходимо ввести конец файла как входной символ.

После реализации простейшего лексического анализатора в соответствии с п.1 Задания необходимо оценить его адекватность. При выполнении п.2 Задания в первую очередь следует уточнить диаграмму состояний распознавателя для многострочных и однострочных комментариев. При этом необходимо учесть все особенности комментариев и некоторых других лексем языка C++, в частности:
– многострочные комментарии не могут быть вложенными;
– многострочный комментарий может использоваться вместо символа-разделителя и его удаление не должно приводить к «склеиванию» лексем и их неправильной интерпретации транслятором;
– окончанию «`*/`» многострочного комментария может предшествовать символ «`*`» в любом количестве;
– комментарии не могут являться частью строковой или символьной константы (начинаться или заканчиваться внутри нее – комментарии не должны распознаваться внутри строковых и символьных констант, а строковые и символьные константы не должны распознаваться внутри комментариев);
– строковые и символьные константы могут включать в себя как символы одинарную и двойную кавычки в виде последовательностей «`\'`» и «`\"`», которые не являются завершением записи константы, а также любые управляющие последовательности, запись которых начинается с символа «`\`»);
– строковая константа может занимать несколько строк в исходном тексте программы;
– любой из символов перевода строки или возврата каретки («\r» или «\n») завершают однострочный комментарий, при этом они являются равнозначными и сохраняются в выходной поток.

Общий вид автомата, соответствующего этим требованиям, будет следующим (конечное состояние, соответствующее концу входного потока, и ведущие в него дуги не изображены):



На данной диаграмме видно, что переход между состояниями, которые находятся в частях автомата, соответствующих распознаванию комментариев, строковых и символьных констант, невозможен, т.е. эти конструкции не могут перекрываться или быть вложенными друг в друга.

При реализации необходимо обеспечить корректное завершение программы (запись символов, закрытие файлов и т.п.) при обнаружении конца входного файла в любом состоянии автомата.

Фрагмент возможной реализации (упрощенный) лексического анализатора на языке C с использованием стандартных библиотек показан ниже.

```
typedef enum States { Normal, Slash, Comment, ... } States;
                          /* перечисление всех состояний автомата */

int main(int argc, char ** argv)
{
FILE * fi, * fo;          /* входной и выходной файл */
States State = Normal;    /* текущее состояние */
int c;                    /* считанный символ (только один текущий!) */

  fi = fopen(argv[1], "rb");
  if (!fi)
  {
    fprintf(stderr, "Input file \"%s\" open error.\n", argv[1]);
    return 1;
  }
  fo = fopen(argv[2], "wb");
  if (!fo)
  {
    fclose(fi);
    fprintf(stderr, "Output file \"%s\" open error.\n", argv[2]);
    return 2;
  }

  while ((c=fgetc(fi)) != EOF)  /* считываем символ и проверяем, */
  {                             /* не конец ли файла */
    switch (State)           /* обрабатываем считанный символ в */
    {                        /* зависимости от текущего состояния */
      case Normal:
        if (c == '/')        /* если встретили слэш, */
          State = Slash;     /* то перешли в соответствующее состояние */
        else if (c == ...)   /* если еще что-то, то аналогично */
          State = ...;       /* и т.д. */
        ...
        else
          fputc(c, fo);      /* дублируем считанный символ */
        break;               /* в выходной поток */
```

```
      case Slash:            /* если предыдущим был слэш, то ... */
        if (c == '*')
          State = Comment;
        else
          State = Normal;
        break;

      ...     /* и т.д. обрабатываем все состояния автомата */
    }
  }

  fclose(fi);    /* не забывайте закрывать файлы! */
  fclose(fo);    /* особенно выходной, т.к. он был открыт для записи */
  return 0;
}
```

В отчете по лабораторной работе должны быть полностью отражены оба варианта распознавателя (по каждому пункту задания). Основная часть должна содержать графы состояний разработанных конечных автоматов и протоколы работы лексических анализаторов. В выводах по работе необходимо привести оценку адекватности разработанных лексических анализаторов.

Лабораторная работа № 2
Распознаватель числовых констант

Цель работы

Получить практический опыт использования нормальной формы Бэкуса-Наура для формального описания регулярных языков и построения лексических анализаторов на основе порождающих грамматик, закрепить навыки автоматного программирования.

Содержание работы

Построение распознавателя числовых констант языков C и C++ с использованием нормальной формы Бэкуса–Наура и детерминированных конечных автоматов, используя в качестве основы распознаватель комментариев, созданный в лабораторной работе № 1.

Задание

1. Определить грамматику, порождающую цепочки, соответствующие записи числовых констант языков C/C++, используя нормальную форму Бэкуса–Наура.
2. Построить конечный автомат лексического анализатора этих цепочек.
3. Реализовать синтаксический анализатор на основе разработанного конечного автомата.

Методические рекомендации

В первую очередь необходимо формализовать предметную область с использованием естественного языка, т.е. определить множество типов данных в соответствии с вариантом индивидуального задания и возможные способы записи констант этих типов. При этом можно также использовать метасимволы, регулярные выражения, синтаксические диаграммы Вирта и другие нотации для наглядного представления синтаксиса. Необходимо учесть, что могут существовать различные способы записи констант для одних и тех же значений, и возможное наличие символов-модификаторов, уточняющих тип константы. Особое внимание следует обратить на необязательные части в записи константы (например, знак числа, дробная часть, символ-модификатор и др.).

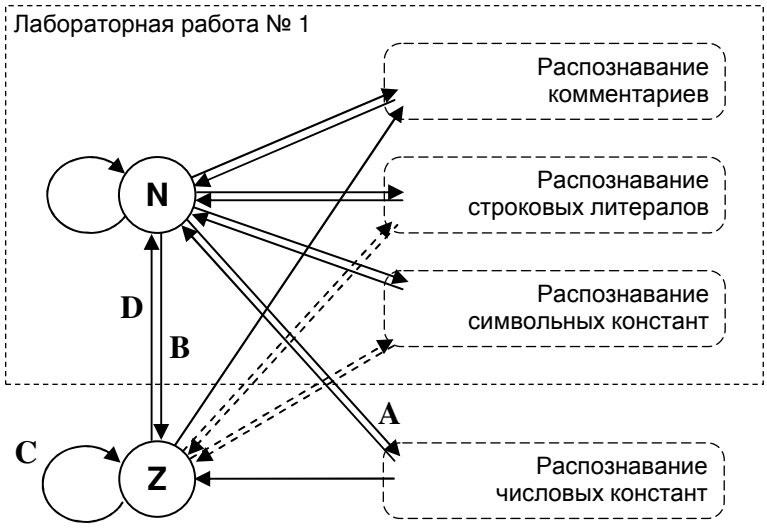
В соответствии с формализованным описанием необходимо построить грамматику, порождающую заданные константы, в нормальной форме Бэкуса–Наура. Учитывая, что язык заданных лексем является регулярным, желательно также использовать регулярную (лево- или праволинейную) грамматику, или даже привести ее к виду автоматной грамматики, что упростит в дальнейшем построение конечного автомата и реализацию лексического анализатора. В записи правил грамматики можно использовать метасимвол «|» для компактной записи правых частей правил, но не допускается использование других метасимволов расширений нотации Бэкуса–Наура и регулярных выражений. Построенная грамматика (с целью упрощения) не должна порождать комментарии, символьные константы и строковые литералы, а также числовые константы, не принадлежащие группе типов данных, заданной вариантом индивидуального задания.

В соответствии с построенной грамматикой необходимо определить конечный автомат, который должен идентифицировать успешное распознавание цепочки, ошибки распознавания и пропуск непринятых цепочек. Автомат не должен принимать числовые константы тех типов, которые не соответствуют варианту индивидуального задания – в этих случаях должна идентифицироваться ошибка распознавания или осуществляться пропуск цепочек (их игнорирование без попытки начать распознавание). Например, считав из входного потока цепочку «123.», распознаватель вещественных констант продолжит чтение и анализ символов, а распознаватель целочисленных – выдаст ошибку

(символ «точка» не может присутствовать в записи этих констант) и перейдет в некоторое исходное состояние, ожидая начала следующей подходящей цепочки во входном потоке. Цепочка «123», если за ней следует не алфавитно-цифровой символ (например, пробел), будет принята как константа типа `int` и не будет принята распознавателем вещественных констант, так как лексема закончилась, но она не содержит признаков константы вещественного типа (десятичной точки или символа «e»). Цепочка «a123» будет просто проигнорирована обоими автоматами, т.к. никакая числовая константа не может начинаться с буквы (эта лексема – идентификатор)

Для корректного распознавания числовых констант необходимо также распознавать комментарии, символьные константы и строковые литералы, т.к. числовые константы как лексемы не могут находиться внутри них. Допустим, автомат в лабораторной работе № 1 распознавал цепочки некоторого языка L' – комбинация цепочек (любых допустимых и в любом порядке) языков комментариев L_R, символьных констант L_C и строковых литералов L_S. Иными словами, $L' = L_R \cup L_C \cup L_S$. Грамматика, построенная в данной работе, задает язык числовых констант L_N и, следовательно, автомат будет являться лексическим анализатором для некоторого языка $L = L_N \cup L' = L_N \cup L_R \cup L_C \cup L_S$.

Таким образом, в качестве основы следует взять конечный автомат из лабораторной работы № 1 и дополнить его новыми состояниями (и соответствующими переходами):



Здесь множество **A** содержит символы, которые могут являться первыми в записи распознаваемых констант. Начиная с этого символа и далее, в области «Распознавание числовых констант», входная цепочка должна дублироваться в выходной поток и завершаться либо типом данных константы (цепочка принята), либо сообщением об ошибке (цепочка не принята). Множество **B** – символы, с которых могут начинаться лексемы, внутри которых распознавание констант невозможно (например, идентификаторы и ключевые слова). Таким образом, состояние **Z** играет роль состояния, блокирующего распознавание констант. Автомат находится в таком состоянии, пока поступают «запрещающие» символы множества **C** (например, буквы, цифры и др.). Следует отметить, что множества **C** и **B** могут отличаться. Вновь распознавание константы может начаться только после «разрешающего» символа из множества **D** (например, символ операции, скобка и т.п.) или разделителя (например, пробел и т.п.).

Распознаватель, реализованный в соответствии с построенным автоматом, должен в результате лексического анализа входного файла (текст программы на языках C/C++) сформировать файл отчета, содержащий все числовые константы из входного файла в том порядке, в каком они были найдены, с указанием типа этих констант. При отсутствии в записи константы модификаторов (суффиксов, уточняющих тип), полагать, что константа имеет базовый тип вне зависимости от ее значения. Если цепочка не была принята, то необходимо вывести соответствующий результат и показать символ, приведший к ошибке

распознавания. Например, отчет распознавателя целочисленных констант может иметь следующий вид:

```
123             int
123ul           unsigned long
123.            ERROR
1e              ERROR
0x123L          long
...
```

Основная часть отчета по лабораторной работе должна содержать:

– формализованное описание правил записи лексем на естественном языке;
– запись грамматики в нотации Бэкуса–Наура;
– граф состояний конечного автомата для представленной грамматики, включающий распознавание комментариев, символьных констант и строковых литералов;
– протоколы работы лексического анализатора, включая специально внесенные ошибки и случаи, не подлежащие распознаванию (запись чисел внутри строк и комментариев, цифры в идентификаторах и т.п.).

В выводах по работе необходимо привести оценку адекватности разработанного лексического анализатора, в т.ч. в тех случаях, когда невозможно однозначно определить границы лексем при помощи регулярной грамматики (т.е. задача должна решаться при помощи контекстно-свободных грамматик).

Варианты индивидуальных заданий

Вариант индивидуального задания выбирается по общим правилам:

1. Целочисленные константы: 8-я, 10-я и 16-я системы счисления; все возможные целочисленные типы (кроме `char`) с учетом модификаторов (суффиксов) в записи константы, тип по умолчанию – `int`.
2. Вещественные константы (с плавающей точкой): запись в виде десятичной дроби, показательной формы и их сочетания; все возможные типы с учетом модификаторов (суффиксов) в записи константы, тип по умолчанию – `double`.

Лабораторная работа № 3
Алгоритм рекурсивного спуска

Цель работы

Изучение LL-методов грамматического разбора для контекстно-свободных грамматик и их практическое освоение на примере метода рекурсивного спуска.

Содержание работы

Построение транслятора для языка, заданного контекстно-свободной грамматикой, с использованием метода рекурсивного спуска.

Задание

1. Записать грамматику, заданную вариантом индивидуального задания, включая полные множества правил, терминальных и нетерминальных символов.
2. Выполнить преобразование записанной грамматики к виду грамматики рекурсивного спуска (при необходимости).
3. Построить транслятор (интерпретатор) программ заданного языка, реализующий алгоритм рекурсивного спуска.

Методические рекомендации

Заданная грамматика G порождает язык L(G), каждая возможная цепочка которого представляет собой *один* оператор присваивания переменной (левый операнд) значения некоторого выражения (правый операнд), причем, при первом использовании в качестве левого операнда, переменная определяется, а при использовании ее в выражении правого операнда – должна быть определена ранее. Последовательность таких операторов, которые могут быть отделены друг от друга пустыми символами (в любом количестве, в т.ч. 0), представляет собой программу на языке L'. Таким образом, $L' = (L(G) \cup \Delta)^*$, где Δ – множество пустых (непечатаемых) символов, выполняющих роль разделителей (в данном случае – между операторами).

Тип данных, значениями которого оперирует программа на языке, порождаемом заданной грамматикой, определяется записью констант этого языка (разделы IV и VI грамматики).

Следует отметить, что правила грамматики, записанные в табл. 3.2, не предполагают наличия разделителей (пустых символов) между лексемами в одном операторе, в то время, как для большинства языков программирования это является нормой. Это сделано для упрощения записи правил грамматики. Например, запись правила `S→I=E;` из раздела I,а табл. 3.2 с учетом наличия пустых символов будет иметь вид $S \to \delta_1 I \delta_2 = \delta_3 E \delta_4 ;$, где $\delta_i \in \Delta^*$ и это так же, с формальной точки зрения, должно быть раскрыто при помощи правил грамматики. Очевидно, что ввод разделителей в явном виде излишне усложняет запись грамматики, поэтому их наличие должно быть учтено во время реализации интерпретатора. При выполнении лабораторной работы необходимо самостоятельно определить либо возможность, либо обязательное отсутствие таких символов, как пробел, табуляция и т.п., между терминальными и нетерминальными символами в правых частях правил разделов I и II грамматики, т.е. *между отдельными лексемами* внутри операторов.

Прежде, чем приступать к построению транслятора, необходимо убедиться, что грамматика, заданная в соответствии с вариантом индивидуального задания, является грамматикой рекурсивного спуска или привести ее к такому виду. Грамматикой рекурсивного спуска называется такая грамматика, в которой для любого нетерминального символа $A \in V^N$ существует либо единственное правило $A \to \alpha$, где

$\alpha \in V^*$, либо только правила вида $A \to a_1\beta_1 \mid a_2\beta_2 \mid \ldots \mid a_n\beta_n$, где $a_1,a_2,\ldots,a_n \in V^T$, $\beta_1,\beta_2,\ldots,\beta_n \in V^*$ причем $a_i \neq a_j$, при $i \neq j$, т.е. правые части этих правил начинаются с различных терминальных символов. Следовательно, по первому символу каждой цепочки, выводимой из некоторого нетерминального символа A, можно однозначно установить соответствующее правило, сравнивая этот символ с первыми символами правых частей правил, левые части которых представлены символом A.

Построение распознавателя для языков, заданных грамматикой рекурсивного спуска, выполняется по следующим правилам:

1. Для каждого нетерминального символа $A \in V^N$ строится своя процедура разбора, например ProcA, выполняющая распознавание всех цепочек, выводимых из символа A.
2. Первый символ входного потока, анализируемый каждой процедурой разбора (т.е. первый символ строки, выводимой из соответствующего нетерминального символа), считывается до ее вызова.
3. При завершении процедуры разбора, ею должен быть считан один символ входного потока, следующий за распознанной цепочкой (т.е. выводимой из соответствующего нетерминального символа).
4. Алгоритм каждой процедуры разбора строится в соответствии с правой частью выбранного правила для соответствующего нетерминального символа:
   - если очередной символ правой части правила является *терминальным*, то он должен быть равен текущему считанному символу входного потока (на первом шаге – производится выбор правила), иначе цепочка не принимается (выводится сообщение об ошибке); при успешном сравнении считывается следующий символ входного потока и происходит переход к анализу следующего символа правой части выбранного правила;
   - если очередной символ правой части правила является *нетерминальным*, то вызывается соответствующая ему процедура разбора (текущий считанный символ будет первым анализируемым символом в этой процедуре – правило 2, а после завершения вызванной процедуры текущий символ уже будет считан из входного потока – правило 3).

Каждая процедура, как правило, выполняет некоторые действия в соответствии с распознанной цепочкой и возвращает некоторое значение. Например, в случае интерпретации выражений, процедура распознавания константы «собирает» символы константы, вычисляет и возвращает ее значение, а процедура распознавания выражения с операцией «+» вызывает процедуры для операндов, суммирует возвращенные ими значения и возвращает полученный результат. Очевидно, что операндом может являться как константа, так и другое выражение, в т.ч. содержащее ту же операцию. Таким образом, возможна рекурсия (отсюда и название метода), но единственное ограничение – рекурсия не должна быть левой (как прямой, так и непрямой).

Расширить применение метода рекурсивного спуска можно за счет:

- приведения исходной грамматики к требуемому виду посредством выполнения левой факторизации, устранения левой рекурсии, преобразования к нормальной форме Грейбах и т.п.;
- построения конечных автоматов для распознавания лексем, записи правил в регулярных выражениях, привлечения эвристического анализа, модификации базового алгоритма процедуры разбора и др.

Контекстные зависимости, присущие заданному описанным образом языку (например, идентификаторы, которые определены в предшествующих операторах, зарезервированные слова и т.п.) реализуются посредством *семантического анализатора*. Семантический анализатор представляет собой некоторую структуру данных (таблицу идентификаторов), хранящую информацию о текущем контексте, и обслуживающих ее алгоритмов (поиск, извлечение, добавление и т.п.). Выбранный способ программной реализации таблицы идентификаторов не должен вносить существенных ограничений на

количество идентификаторов, используемых в программе, и их длину, если это не обусловлено исходной грамматикой.

Транслятор должен после интерпретации каждого оператора программы выводить его порядковый номер, имя переменной и значение, присвоенное ей в этом операторе, а после интерпретации всей программы – список всех переменных и их значений. В случае ошибки необходимо выводить развернутое сообщение о ее причинах. Ниже приведены примеры протоколов в трех случаях: успешная интерпретация, ошибка в написании идентификатора (имени переменной), обработка пустого файла:

```
Begin parsing.
Operator 1:   a = 12
Operator 2:   k34 = 0
Operator 3:   c = 1
Operator 4:   c = 18
Operator 5:   x0 = 502
Operator 6:   a = -3
Operator 7:   sum = 517
End parsing.
5 variables defined:
a = -3
k34 = 0
c = 18
x0 = 502
sum = 517
```

```
Begin parsing.
Operator 1:   a = 12
Operator 2:
Error 107: Identifier missing.
Abort parsing.
```

```
Begin parsing.
End parsing.
No variables defined.
```

Основная часть отчета по лабораторной работе должна содержать:

- запись исходной грамматики (по табл. 3.1 и 3.2) в нотации Бэкуса–Наура;
- описание языка, порождаемого заданной грамматикой (на естественном языке с примерами порождаемых конструкций);
- вывод о том, принадлежит или нет заданная грамматика классу грамматик рекурсивного спуска;
- описание преобразований грамматики (в случае необходимости) и полученную в результате грамматику рекурсивного спуска;
- описание алгоритмов процедур разбора (по одной для каждого раздела грамматики);
- протоколы работы лексического анализатора, включая как интерпретацию правильных программ языка, так и случаи идентификации специально внесенных ошибок.

Варианты индивидуальных заданий

В соответствии с номером варианта индивидуального задания (от 1 до 20) из табл. 3.1 выбираются варианты 6-ти разделов грамматики $G(V^T, V^N, P, S)$. Затем, в соответствии с вариантами разделов, из табл. 3.2 выписываются правила, образующие полное множество правил P грамматики. На основе полученного множества правил P строится алфавит грамматики $V = V^N \cup V^T$. Нетерминальный символ S является целевым (начальным) во всех полученных грамматиках.

Таблица 3.1

Соответствие вариантов разделов грамматики варианту индивидуального задания

Вариант задания	**Варианты разделов грамматики**						**Вариант задания**	**Варианты разделов грамматики**					
	I	**II**	**III**	**IV**	**V**	**VI**		**I**	**II**	**III**	**IV**	**V**	**VI**
1	а	б	г	а	б	б	11	б	а	а	а	б	г
2	б	б	г	а	а	б	12	в	г	г	в	б	б
3	а	б	в	г	б	б	13	а	а	в	а	б	г
4	б	б	в	г	а	б	14	б	а	в	а	б	а
5	в	б	б	г	а	в	15	г	а	в	а	б	а
6	б	д	а	а	а	б	16	а	в	б	б	б	а
7	г	г	б	в	б	а	17	г	в	б	б	а	а
8	г	а	а	а	а	г	18	а	д	б	б	б	а
9	г	г	г	в	б	б	19	б	в	б	б	б	а
10	в	г	б	в	б	а	20	в	в	б	б	а	а

Таблица 3.2

Содержание разделов грамматики по вариантам

Раздел грамматики	**Вариант раздела**	**Правила грамматики**	**Примечания**
I	а	`S→I=E;`	Операторы присваивания переменной с именем `I` значения выражения `E`. Переменная с именем `I` определяется, если она не была определена ранее
	б	`S→I:=E;`	
	в	`S→(I,E)`	
	г	`S→I(E)`	

II	а	`E→E+T|E-T|T`
		`T→T*M|T/M|M`
		`M→(E)|I|C`
			Выражения с традиционными арифметическими операциями +, –, *, / и круглыми скобками

	б	`E→E"|"T|T`
		`T→T&M|M`
		`M→~M|(E)|I|C`
			Выражения с поразрядными логическими операциями | (или), & (и), ~ (инверсия) и круглыми скобками

	в	`E→E+T|E-T|T`
		`T→M|F(E)`
		`M→I|C`
		`F→sin|cos|sqr|sqrt`
			Выражения с операциями суммирования, вычитания и вызова функции (синус, косинус, квадрат числа, квадратный корень)

	г	`E→-E|+(T)|*(T)|S|M`
		`T→E,T|E`
		`M→I|C`
			Выражения с унарным минусом и n-местными операциями суммирования и умножения в префиксной форме

	д	`E→T>T|T<T|T==T|T`
		`T→T+M|T-M|M`
		`M→!M|(E)|I|C`
			Выражения с логическими и арифметическими операциями, семантика которых соответствует языку C

III	а	`I→A|AA|AD|AAD`	Идентификаторы
	б	`I→AI|A`	
	в	`I→AK|A` `K→AK|DK|A|D`	
	г	`I→AK|A` `K→DK|D`	

IV	а	C→DC|D	Константы
	б	C→DC|D|.R R→DR|D	
	в	C→#R R→DR|D	
	г	C→D	
V	а	A→a|b|c|d|e|f|g|h|i|j|k|l|m| n|o|p|q|r|s|t|u|v|w|x|y|z	Буквы
	б	A→a|b|c|d|e|f|g|h|i|j|k|l|m| n|o|p|q|r|s|t|u|v|w|x|y|z|_	
VI	а	D→0|1|2|3|4|5|6|7|8|9	Десятичные цифры
	б	D→0|1	Двоичные цифры
	в	D→true|false	Логические значения
	г	D→0|1|2|3|4|5|6|7	Восьмеричные цифры

Лабораторная работа № 4
Генерация промежуточного кода

Цель работы

Практическое освоение методов построения трансляторов на примере метода трансляции в промежуточный код.

Содержание работы

Построение транслятора программ в промежуточный код.

Задание

На основе интерпретатора языка, построенного в лабораторной работе № 3, создать транслятор программы в промежуточный код, представляющий собой список триад.

Методические рекомендации

В компиляторах языков программирования для оптимизации программы и генерации машинного кода используется некоторый промежуточный код, который может быть записан в форме польской инверсной записи, тетрад, триад и др. Триада (тройка) – это совокупность операции и двух ее операндов, представляющая собой некоторую элементарную связку. Триады могут эффективно использоваться как для записи порядка вычисления операций в выражении (в том числе и операторов присваивания), так и для представления управляющих конструкций (условные и безусловные переходы, с использованием которых также строятся циклы), и являются разумным компромиссом между компактностью структуры данных и удобством анализа.

Список триад имеет регулярную структуру, за счет чего к нему легко применяются различные формальные методы анализа и алгоритмы преобразований (в т.ч. оптимизации кода). Каждая триада имеет вид @(op1, op2) или просто @ op1 op2, где @ – обозначение операции, op1 и op2 – первый (левый) и второй (правый) операнды. Например, выражение a + 5 будет представлено триадой +(a, 5) или + a 5.

В качестве обозначения операций могут выступать как символы, используемые для их записи, так и некоторые целочисленные значения или значения перечисления, поставленные в соответствие операциям. При этом следует учесть, что собственно присваивание тоже рассматривается как операция. Кроме того, с точки зрения реализации распознавателя и дальнейшей оптимизации промежуточного кода, имеет смысл рассматривать получение значения константы и обращение к переменной как элементарные операции, а также предусмотреть ситуации, когда триада не соответствует никакой операции (фиктивная триада).

Операндами триады могут быть: константное значение, переменная, результат вычисления другой триады, адрес перехода в командах передачи управления (в данной работе отсутствует). В случае унарных операций первый операнд является единственным, а второй – фиктивным. Операции с большей арностью (например, выражение (+ 1 2 3 4 5) языка LISP) сводятся к последовательности бинарных операций. В некоторых случаях это требует соблюдения дополнительных соглашений (например, вызовы процедур и функций с различным числом параметров), использования управляющих конструкций (например, тернарная операция ?: языка C) или применения других методов, поэтому подобные случаи в данной работе не рассматриваются.

В записи триад числовые константы записываются в общепринятой форме, операнды-переменные обозначаются их именами, а результат другой триады – ссылкой на нее (например, ее номером с предшествующим символом «^»). Допустим, дана следующая

программа, состоящая из двух операторов присваивания (нумерация операторов приведена условно):

```
1)  a = 1;
2)  b = 2 * (a + 5);
```

Запись этой программы в виде списка триад будет следующей:

```
1)  1:  =(a, 1)     // a = 1;
2)  2:  +(a, 5)     // a + 5
    3:  ×(2, ^2)    // 2 * (a + 5)
    4:  =(b, ^3)    // b = 2 * (a + 5);
```

Однако, с учетом приведенных выше замечаний об элементарных операциях, представляемых отдельными триадами, эта запись должна иметь такой вид:

```
1)  1:  V(a, ∅)     // a
    2:  C(1, ∅)     // 1
    3:  =(^1, ^2)   // a = 1;
2)  4:  V(b, ∅)     // b
    5:  C(2, ∅)     // 2
    6:  V(a, ∅)     // a
    7:  C(5, ∅)     // 5
    8:  +(^6, ^7)   // a + 5
    9:  ×(^5, ^8)   // 2 * (a + 5)
    10: =(^4, ^9)   // b = 2 * (a + 5);
```

Здесь как V и C обозначены операции обращения к переменной и загрузки константы соответственно, а ∅ – фиктивный операнд. Необходимость подобной «детальной» записи вызвана тем, что разным компьютерам свойственны различные сочетания инструкций и режимов адресации их операндов. Эти свойства компьютеров учитываются на этапах оптимизации и генерации машинного кода.

При построении транслятора программ в промежуточный код в качестве основы можно использовать интерпретатор, разработанный в лабораторной работе № 3. В этом случае необходимо модифицировать процедуры разбора для нетерминальных символов разделов I–IV заданной грамматики (в соответствии с тем же вариантом индивидуального задания). Основу грамматики составляют разделы I и II, правила которых определяют множество операций, их приоритеты и синтаксис их использования. Соответствующие процедуры должны вместо выполнения операции добавлять в список соответствующую триаду, а возвращаемым значением (результатом работы) процедуры будет являться номер этой триады, который будет использоваться в вышестоящей процедуре в качестве операнда в записи очередной триады и т.д. Следует отметить, что если возможны правила вида $A \to B$, то фактически никакая операция не выполняется, следовательно, новая триада в список не добавляется и процедура разбора для символа A просто возвращает то же значение, которое возвратила вызванная ею процедура для символа B. Аналогично, если в записи выражений для задания приоритетов операций используются скобки, то они влияют на порядок разбора, но операцией не являются.

Процедуры разделов III и IV осуществляют разбор лексем (имен переменных и записи констант) и реализуются, как правило, при помощи автоматной модели с использованием правил разделов V и VI. Эти процедуры образуют лексический анализатор (нижний уровень разбора) и не вызывают других процедур, поэтому они должны просто формировать и добавлять в список триады обращения к переменным и получения констант соответственно. Следует также обратить внимание на то, что в

выражении в правой части оператора присваивания могут присутствовать только те переменные, которые уже определены ранее, тогда как новый идентификатор в левой части приводит к определению переменной. Следовательно, процедуры для распознавания идентификаторов в левой и в правой частях оператора присваивания будут отличаться друг от друга по реализации, но, при этом, будут формировать одинаковые триады и возвращать одинаковые значения.

Транслятор должен формировать список триад (аналогично примерам выше, с нумерацией триад, но, конечно, без комментариев) по мере разбора входной программы и по окончании работы создавать файл протокола, содержащий этот список в виде текста. В случае ошибки в протоколе должно фиксироваться развернутое сообщение о ее причинах (аналогично предыдущей работе).

Основная часть отчета по лабораторной работе должна содержать:

– запись грамматики в нотации Бэкуса–Наура (получена в предыдущей работе);
– описание реализованного способа формирования и представления триад;
– протоколы работы лексического анализатора.

Лабораторная работа № 5
Оптимизация промежуточного кода

Цель работы

Изучение методов машинно-независимых и машинно-зависимых методов оптимизации кода и их практическое освоение на примере модуля оптимизации промежуточного кода транслятора.

Содержание работы

Построение модуля оптимизации промежуточного кода для транслятора программ.

Задание

Реализовать оптимизацию промежуточного кода (свертку) за счет исключения триад, связанных с константными вычислениями и обращениями к данным.

Методические рекомендации к заданию

Оптимизация программы при ее компиляции производится с целью улучшения ее характеристик – уменьшения объема используемой памяти (как для программы, так и для данных) или увеличения производительности. Обычно достижение двух этих критериев противоречит друг другу, а уменьшение объема используемой памяти для кода программы часто вынуждает использовать алгоритмы, требующие больш*и*х массивов данных. Тем не менее, есть возможность улучшения всех свойств программы за счет исключения «лишнего» кода.

Оптимизация может быть *машинно-независимой* и *машинно-зависимой*. В первом случае устраняется общая избыточность кода, во втором – используются особенности архитектуры конкретной вычислительной системы для *свертки* последовательностей инструкций в более компактные и/или быстрые.

В качестве машинно-независимой оптимизации необходимо осуществить свертку константных выражений до одной константы, таким образом, значение константного выражения будет вычислено во время трансляции. Свертку необходимо выполнять без учета возможных изменений значений переменных или их отсутствия в транслируемом фрагменте, т.е. выражения, подлежащие свертке, не должны содержать переменных. Таким образом, последовательность триад, соответствующая загрузке констант и вычислению значения константного выражения, будет свернута до одной триады загрузки константы.

Машинно-зависимая оптимизация должна заключаться в учете особенностей архитектуры компьютера, представленных в табл. 4.1.

Таблица 4.1

Допустимые режимы адресации целевой машины

Тип операции	**Группа операции**	**Варианты допустимых сочетаний режимов адресации**		**Примеры**	
		первый операнд	**второй операнд**	**триада**	**ассемблер**
Унарная	Все, кроме пересылки (загрузки)	Регистровый	–	–(^5, ∅)	`inc R`
		Непосредственный	–	–(8, ∅)	`cpl Data`
		Прямой	–	–(a, ∅)	`inc (Addr)`
	Пересылка (загрузка)	Непосредственный	–	C(8, ∅)	`mov R, Data`
		Прямой	–	V(b, ∅)	`mov R, (Addr)`
Бинарная	Все, кроме присваивания	Регистровый	Регистровый	+(^6, ^7)	`add R, R`
		Регистровый	Непосредственный	+(^6, 34)	`add R, Data`
	Присваивание	Прямой	Регистровый	=(a, ^4)	`mov (Addr), R`

Прямой режим адресации означает, что за кодом операции следует адрес, по которому производится обращение для считывания или записи данных – эти данные и являются собственно операндом. При *непосредственном* режиме адресации за кодом операции следуют *непосредственно* данные, т.е. значение операнда. Из таблицы 4.1 видно, что в данном компьютере эти режимы не совмещаются для двух операндов одной операции.

В реальных машинах приемником результата обычно является один из операндов – часто это регистр процессора или вершина стека для стековой машины. Таким образом, бинарная операция становится не трех- (операнд 1, операнд 2, результат), а двухадресной (операнд 1 и он же результат, операнд 2). В отличие от инструкций традиционных реальных компьютеров, в триадах не указывается приемник результата – результат просто будет востребован по ссылке на триаду. Поэтому в данном случае регистровый режим адресации может также означать и стековый – это зависит от типа конкретной машины (регистровая, стековая) и, следовательно, способа реализации передачи результата одной операции для использования его в качестве операнда другой операцией (результат будет помещен либо в регистр, либо на вершину стека).

Идентификаторы (имена) переменных существуют только в исходном коде программы и в ее записи в виде триад. В машинном коде (непосредственно в инструкциях процессора) им соответствуют адреса переменных. Поэтому триаде обращения к переменной соответствует, фактически, некоторая операция подготовки адреса (актуально для некоторых типов микропроцессоров), а случаю прямой адресации будет соответствовать использование имени переменной в качестве операнда триады.

Обобщая эту информацию, получаем, что свертка может выполняться следующим образом:

1. Если единственный операнд унарной операции или второй (правый) операнд бинарной операции, кроме присваивания, является ссылкой на триаду загрузки константы, то он заменяется самой константой, а соответствующая триада загрузки удаляется.
2. Если единственный операнд унарной операции является константой, то вычисляется соответствующая операция, а триада заменяется на триаду загрузки полученного значения.
3. Если первый (левый) операнд бинарной операции, кроме присваивания, является ссылкой на триаду загрузки константы и второй (правый) операнд – константой, то вычисляется соответствующая операция и триада заменяется на триаду загрузки полученного значения, а предшествующая триада загрузки константы удаляется.
4. Если первый (левый) операнд операции присваивания является ссылкой на триаду обращения к переменной, то он заменяется именем переменной, а соответствующая триада загрузки удаляется.

Каждая триада в списке должна быть проанализирована на возможность применения к ней каждого правила, но для адекватной оптимизации промежуточного кода необходима также четкая последовательность этих действий. Так как в качестве операндов могут фигурировать ссылки только на предшествующие триады, то анализировать список триад необходимо «сверху вниз» – последовательно, от меньших номеров триад к большим. Каждую триаду необходимо рассматривать на возможность применения к ней правил в приведенном выше порядке, так как применение правила 1 может повлечь возможность применения правила 2 и т.д., но не наоборот.

Также следует обратить внимание на то, что исключить какую-либо триаду из списка можно только тогда, когда на нее не останется ни одной ссылки. Учитывая, что в рамках данной работы не ставится задача оптимизации общих подвыражений, можно утверждать, что ссылка на каждую триаду является единственной. Таким образом, исключать триады, ставшие «ненужными», можно одновременно с заменой ссылки на нее значением.

При программной реализации рекомендуется не исключать триады из списка, так как это приведет к изменению нумерации, а заменять их пустыми и пропускать при

контрольном выводе или выводе в файл протокола. Это позволит сохранить нумерацию, что облегчает реализацию алгоритма, проверку его работоспособности и адекватности.

Работу алгоритма можно проиллюстрировать на следующем примере:

1)	a = 1;	1:	V(a, ∅)	// a
		2:	C(1, ∅)	// 1
		3:	=(^1, ^2)	// a = 1;
2)	b = (a + 2 * (5 + 7)) * 3;	4:	V(b, ∅)	// b
		5:	V(a, ∅)	// a
		6:	C(2, ∅)	// 2
		7:	C(5, ∅)	// 5
		8:	C(7, ∅)	// 7
		9:	+(^7, ^8)	// 5 + 7
		10:	×(^6, ^9)	// 2 * (5 + 7)
		11:	+(^5, ^10)	// a + 2 * (5 + 7)
		12:	C(3, ∅)	// 3
		13:	×(^11, ^12)	// (a + 2 * (5 + 7)) * 3
		14:	=(^4, ^13)	// b = (a + 2 * (5 + 7)) * 3;

В табл. 4.2 приведен список триад, соответствующий приведенным операторам, и показаны шаги, на которых выполняется свертка. В списке триад выделены те операнды и триады, которые подлежат свертке, а в нижней строке указан номер применяемого правила.

Таблица 4.1

Пример оптимизации промежуточного кода

	Шаг 1	**Шаг 2**	**Шаг 3**	**Шаг 4**	**Шаг 5**	**Шаг 6**	**Шаг 7**	**Шаг 8**	**Шаг 9**
1	**V(a, ∅)**								
2	C(1, ∅)	C(1, ∅)	C(1, ∅)	C(1, ∅)	C(1, ∅)	C(1, ∅)	C(1, ∅)	C(1, ∅)	C(1, ∅)
3	=(**^1**, ^2)	=(a, ^2)	=(a, ^2)	=(a, ^2)	=(a, ^2)	=(a, ^2)	=(a, ^2)	=(a, ^2)	=(a, ^2)
4	V(b, ∅)	V(b, ∅)	V(b, ∅)	V(b, ∅)	V(b, ∅)	V(b, ∅)	V(b, ∅)	**V(b, ∅)**	
5	V(a, ∅)	V(a, ∅)	V(a, ∅)	V(a, ∅)	V(a, ∅)	V(a, ∅)	V(a, ∅)	V(a, ∅)	V(a, ∅)
6	C(2, ∅)	C(2, ∅)	C(2, ∅)	C(2, ∅)	**C(2, ∅)**				
7	C(5, ∅)	C(5, ∅)	**C(5, ∅)**						
8	C(7, ∅)	**C(7, ∅)**							
9	+(^7, ^8)	+(^7, **^8**)	+(**^7**, 7)	**C(12, ∅)**					
10	×(^6, ^9)	×(^6, ^9)	×(^6, ^9)	×(^6, **^9**)	×(**^6**, 12)	**C(24, ∅)**			
11	+(^5, ^10)	+(^5, ^10)	+(^5, ^10)	+(^5, ^10)	+(^5, ^10)	+(^5, **^10**)	+(^5, 24)	+(^5, 24)	+(^5, 24)
12	C(3, ∅)	C(3, ∅)	C(3, ∅)	C(3, ∅)	C(3, ∅)	C(3, ∅)	**C(3, ∅)**		
13	×(^11,^12)	×(^11,^12)	×(^11,^12)	×(^11,^12)	×(^11,^12)	×(^11,^12)	×(^11,**^12**)	×(^11, 3)	×(^11, 3)
14	=(^4, ^13)	=(^4, ^13)	=(^4, ^13)	=(^4, ^13)	=(^4, ^13)	=(^4, ^13)	=(^4, ^13)	=(**^4**, ^13)	=(a, ^13)
	4	**1**	**3**	**1**	**3**	**1**	**1**	**4**	**–**

Программная реализация должна быть выполнена на основе транслятора, созданного в предыдущей лабораторной работе. Трансляция в промежуточный код и его оптимизация должны представлять собой два этапа, выполняемых строго последовательно, а передача информации между ними – посредством внутреннего представления списка триад (не через файл протокола). Оптимизированный промежуточный код желательно помещать в тот же файл протокола, вслед за исходным.

В основной части отчета необходимо отразить разработанный алгоритм оптимизации промежуточного кода и протоколы работы усовершенствованного транслятора. Выводы по работе должны содержать обоснование адекватности реализованного алгоритма.

Лабораторная работа № 6
Алгоритм простого предшествования

Цель работы

Изучение LR-методов грамматического разбора для контекстно-свободных грамматик и их практическое освоение на примере LR(1)-метода грамматик простого предшествования.

Содержание работы

Построение транслятора для языка, заданного контекстно-свободной грамматикой, с использованием метода простого предшествования.

Задание

1. Записать исходную грамматику, заданную вариантом индивидуального задания в лабораторной работе № 3.
2. Выполнить преобразование записанной грамматики к виду грамматики простого предшествования (при необходимости).
3. Построить транслятор программ заданного языка в промежуточный код, реализующий LR(1)-алгоритм простого предшествования.

Методические рекомендации

Прежде, чем приступать к построению транслятора, необходимо убедиться, что грамматика, заданная в соответствии с вариантом индивидуального задания (из лабораторной работы № 3), является грамматикой простого предшествования или привести ее к такому виду.

Грамматикой простого предшествования называется такая грамматика, которая:

1. Является приведенной контекстно-свободной грамматикой;
2. Не содержит двух правил с одинаковыми правыми частями;
3. Для $\forall X,Y \in V$ определено не более одного из трех отношений предшествования:
   - $X \lessdot Y$ – «предшествует основе» или просто «предшествует», если $(A \to \alpha X M \beta) \in P$ и $\exists M \Rightarrow^{+} Y\gamma$, где $A,M \in V^{N}$, $\alpha,\beta,\gamma \in V^{*}$, т.е. свертка X к A происходит позже, чем Y к M;
   - $X \doteq Y$ – «составляют основу», если $(A \to \alpha X Y \beta) \in P$, где $A \in V^{N}$, $\alpha,\beta \in V^{*}$, т.е. свертка X и Y происходит одновременно (непосредственно соседствуют в правой части правила);
   - $X \gtrdot Y$ – «следует за основой» или просто «следует», если $(A \to \alpha M N \beta) \in P$ и $\exists M \Rightarrow^{+} \gamma X$ и $\exists N \Rightarrow^{+} Y\delta$, где $A,M,N \in V^{N}$, $\alpha,\beta,\gamma,\delta \in V^{*}$, т.е. свертка X к M происходит раньше, чем Y к N.

Следовательно, для каждой пары символов XY, $X,Y \in V$ можно однозначно установить, принадлежат ли они одной связке или между ними находится граница связки, и выполнить свертку связки к нетерминальному символу по соответствующему правилу.

Отыскание отношений следует начинать с отношения «$\doteq$» выписыванием всех подходящих пар из правых частей правил. Затем отыскиваются отношения «$\lessdot$» и «$\gtrdot$» путем подстановок крайних символов правых частей правил вместо нетерминальных символов в уже найденные пары (последнего символа правой части правила вместо левого символа пары и первого – вместо правого) и выписыванием полученных пар, а также рекурсивно для вновь найденных пар.

Найденные отношения составляют функцию предшествования, которая фиксируется в виде квадратной матрицы предшествования. Строки и столбцы матрицы обозначены

всеми символами алфавита грамматики. Строки соответствуют левым символам пар, столбцы – правым, а ячейки на пересечении содержат обозначения отношений. При реализации транслятора матрица представляет собой квадратный двумерный массив, элементы которого кодируют найденное отношение либо отсутствие такового. На практике целесообразно группировать равнозначные, с точки зрения семантики, терминальные символы, например, буквы, используемые для записи идентификаторов, цифры, составляющие числовые константы, символы операций и т.п. Тогда отношения определяются для пары любых символов, принадлежащих группам, что влечет существенное сокращение размера матрицы предшествования.

Тогда процесс грамматического разбора будет заключаться в последовательном чтении входных символов и помещении их в стек вместе с обозначением отношения двух соседних символов до тех пор, пока не будет установлено отношение «$\doteq$» между символом на вершине стека и очередным считанным символом. В этом случае вся цепочка между «$\lessdot$» и «$\gtrdot$», соответствующая правой части ровно одного правила грамматики, извлекается и вместо входного (рекурсивно) аналогичным образом обрабатывается нетерминальный символ, к которому производится свертка. Если последовательность между «$\lessdot$» и «$\gtrdot$» не соответствует ни одному существующему правилу грамматики или на очередном шаге не установлено отношение предшествования для пары символов, то работа распознавателя завершается ошибкой (цепочка не принимается).

При реализации транслятора следует учесть, что правила грамматики, записанные в табл. 3.2, не предполагают наличия разделителей (пустых символов) между лексемами в одном операторе, в то время, как для большинства языков программирования это является нормой. Это сделано для упрощения записи правил грамматики. Например, запись правила `S→I=E;` из раздела I,а табл. 3.2 с учетом наличия пустых символов будет иметь вид $S \to \delta_1 I \delta_2 = \delta_3 E \delta_4$; , где $\delta_i \in \Delta^*$ и это так же, с формальной точки зрения, должно быть раскрыто при помощи правил грамматики. Очевидно, что ввод разделителей в явном виде излишне усложняет запись грамматики, поэтому их наличие должно быть учтено во время реализации транслятора. При выполнении лабораторной работы необходимо самостоятельно определить либо возможность, либо обязательное отсутствие таких символов, как пробел, табуляция и т.п., между терминальными и нетерминальными символами в правых частях правил разделов I и II грамматики, т.е. *между отдельными лексемами* внутри операторов.

Отчет должен содержать исходную грамматику, описание и результат ее преобразования, процесс и результат отыскания функции предшествования и полученную матрицу предшествования (аналогично примеру ниже), пример разбора цепочки языка, исходный код транслятора с комментариями, тестовые примеры и результат их выполнения (промежуточный код), выводы об эквивалентности построенного транслятора и из лабораторной работы № 4.

Пример построения и работы распознавателя для грамматики простого предшествования

1. Исходная грамматика дополнена новым целевым символом S' и правилом, включающим ограничитель ⊥, который считается терминальным символом:

$S' \to \perp S \perp$

1. $S \to a$
2. $S \to aT$
3. $S \to [S]$
4. $T \to b$
5. $T \to bT$

2. Найдены отношения предшествования для всех пар символов X и Y и построена матрица отношений (матрица предшествования):

		Y						
		S	T	⊥	a	b	[	]
X	S			≐				≐
	T			⋗				⋗
	⊥	≐			⋖		⋖	
	a		≐	⋗		⋖		⋗
	b		≐	⋗		⋖		⋗
	[	≐			⋖		⋖	
	]			⋗				⋗

3. На вход распознавателя поступает цепочка [[abb]], конец цепочки считаем маркированным ограничителем ⊥. Эта цепочка может быть выведена только следующим образом (вывод, конечно, должен быть правосторонним, но в данной грамматике это не играет роли в силу структуры ее правил):

$$S \overset{3}{\Rightarrow} [S] \overset{3}{\Rightarrow} [[S]] \overset{2}{\Rightarrow} [[aT]] \overset{5}{\Rightarrow} [[abT]] \overset{4}{\Rightarrow} [[abb]]$$

Таким образом, получена последовательность правил: 3 3 2 5 4 (в данном примере это желательно знать для проверки правильности работы распознавателя, но на практике это неизвестно и распознаватель должен дать ответ – правильная цепочка или нет).

4. Протокол работы распознавателя:

№ шага	Состояние стека	Отно-шение	Входная цепочка	Операция	№№ правил (обр. порядок)
0	⊥			Инициализация	–
1	⊥	⋖	[[abb]]⊥	Сдвиг	–
2	⊥⋖[	⋖	[abb]]⊥	Сдвиг	–
3	⊥⋖[⋖[	⋖	abb]]⊥	Сдвиг	–
4	⊥⋖[⋖[⋖a	⋖	bb]]⊥	Сдвиг	–
5	⊥⋖[⋖[⋖a⋖b	⋖	b]]⊥	Сдвиг	–
6	⊥⋖[⋖[⋖a⋖b	⋗	]]⊥	Свертка (к T)	**4**
7	⊥⋖[⋖[⋖a⋖b≐T	⋗	]]⊥	Свертка (к T)	**5** 4
8	⊥⋖[⋖[⋖a≐T	⋗	]]⊥	Свертка (к S)	**2** 5 4
9	⊥⋖[⋖[≐S	≐	]]⊥	Сдвиг	2 5 4
10	⊥⋖[⋖[≐S≐]	⋗	]⊥	Свертка (к S)	**3** 2 5 4
11	⊥⋖[≐S	≐	]⊥	Сдвиг	3 2 5 4
12	⊥⋖[≐S≐]	⋗	⊥	Свертка (к S)	**3** 3 2 5 4
13	⊥≐S	≐	⊥	Сдвиг	3 3 2 5 4
14	⊥≐S≐⊥			Стоп, цепочка **принята**	**3 3 2 5 4**

Примечание. Фактически, шаг 14 на практике не выполняется, т.к. на шаге 13 уже можно обнаружить, что считан ограничитель цепочки (например, достигнут конец файла), а стек содержит только целевой символ S.

Список рекомендуемой литературы

Основная литература

1. Молчанов, А. Ю. Системное программное обеспечение : учебник для студентов вузов / А. Ю. Молчанов. – СПб.: Питер, 2003. – 395 с.
2. Гордеев, А. В. Системное программное обеспечение : учебник для студентов вузов / А. В. Гордеев, А. Ю. Молчанов. – СПб.: Питер, 2002. – 736 с.
3. Опалева, Э. А. Языки программирования и методы трансляции / Э. А. Опалева, В. П. Самойленко. – СПб.: БХВ-Петербург, 2005. – 476 с.
4. Пратт, Т. Языки программирования: разработка и реализация / Т. Пратт, М. Зелковиц ; Под общей ред. А. Матросова. – СПб.: Питер, 2002. – 688 с.
5. Карпов, Ю. Г. Основы построения трансляторов: теория и технология программирования : учебное пособие для студентов вузов / Ю. Г. Карпов. – СПб.: БХВ-Петербург, 2005. – 270 с.

Дополнительная литература

6. Ахо, А. Теория синтаксического анализа, перевода и компиляции / А. Ахо, Дж. Ульман. – т. 1, 2. – М.: Мир, 1978.
7. Грис, Д. Конструирование компиляторов для цифровых вычислительных машин / Д. Грис. – М.: Мир, 1975.
8. Вайнгартен, Ф. Трансляция языков программирования / Ф. Вайнгартен. – М.: Мир, 1977. – 190 с.
9. Карпов, Ю. Г. Теория автоматов : учебник для вузов / Ю. Г. Карпов. – СПб.: Питер, 2002. – 208 с.
10. Фридл, Дж. Регулярные выражения: библиотека программиста / Дж. Фридл. – 2-е изд. – СПб.: Питер, 2003. – 464 с.
11. Новиков, Ф. А. Дискретная математика для программистов : учебник для вузов / Ф. А. Новиков. – СПб.: Питер, 2001. – 304 с.
12. Элджер, Дж. C++: библиотека программиста / Дж. Элджер. – СПб.: Питер, 2001. – 320 с.
13. Бентли, Д. Жемчужины программирования: библиотека программиста / Д. Бентли. – 2 изд. – СПб.: Питер, 2002. – 272 с.