## Лабораторный практикум по дисциплине «Теория формальных языков и компиляторов»

## Основные требования к отчетам по лабораторным работам

1. Отчеты оформляются в виде файлов формата Microsoft Word (файлы других форматов не принимаются), размер шрифта: от 12 до 14.
2. На титульном листе должны быть указаны **вариант задания** на курсовую работу и **имя системы правил**, которую надо проверять, в личном каталоге на сервере «ВебТрансБилдер».
3. Перечень разделов, которые должны присутствовать в отчете, указан в каждом задании на работы 1-8.
4. Все элементы отчета, в первую очередь – скриншоты, должны быть легко читаемы без изменения масштаба и поворотов страницы.
5. Тестовые программы в отчете должны быть приведены только в **текстовом** представлении (ни в коем случае не в виде скриншотов).

Несоблюдение этих требований может привести либо к снижению оценочного балла за работу сразу, либо к возврату отчета на доработку с последующим снижением балла.

## Лабораторная/практическая работа № 1

1. Название работы: «Лексика языков программирования. Регулярные выражения».
2. Цели работы: освоение основных навыков работы с учебным пакетом программ автоматизации разработки трансляторов ВебТрансБилдер, изучение и освоение пользовательского интерфейса пакета и форматов исходных данных/результатов работы, изучение метаязыка регулярных выражений и технологии разработки систем правил определения лексики языков программирования, изучение основных понятий теории конечных автоматов без памяти.
3. Основные теоретические сведения:

Под лексикой языка программирования понимается совокупность правил образования слов из символов. Такая совокупность может быть определена множеством различных способов, одним из которых, очень широко используемым, являются регулярные выражения.

Регулярные выражения

Регулярные выражения представляют собой формализм (метаязык), используемый для описания процесса порождения цепочек символов. Регулярные выражения широко используются и в других целях, в частности для поиска вхождений строк символов в другие строки.

Основой этого метаязыка являются первичные регулярные выражения. Первичным называется регулярное выражение, порождающее (описывающее) цепочки, состоящие ровно из одного символа. Это выражения следующих видов:

* [<*произвольный символ*>], например [*a*]. Такое регулярное выражение порождает единственную цепочку, состоящую только из этого символа (это обозначается так: [*a*] → *a*).
* [<*перечень символов*>], например [*aA*]. Такое выражение порождает несколько цепочек, каждая из которых содержит в точности один символ из указанного в квадратных скобках перечня ([*aA*] → *а* и [*aA*] → *А*);
* [<*диапазон символов*>], например [*0–9*]. Это выражение можно считать сокращенной формой записи выражения вида [*0123456789*]. Оно порождает ровно 10 различных цепочек, каждая из которых содержит единственную десятичную цифру.

В рассматриваемом здесь диалекте языка регулярных выражений используются метасимволы «[», «]» и «*–*», не принадлежащие определяемым цепочкам символов. Существуют и другие метасимволы, они будут определены позже.

Со второй и третьей формами записи первичных регулярных выражений (перечень символов и диапазон символов) связаны три умолчания, о которых всегда нужно помнить.

Во-первых, для указания диапазона символов необходимо точно знать таблицу кодирования символов, чтобы быть уверенным, что в диапазон не попадут «лишние» символы. Например, при использовании кодировки ASCII запись вида [*a–F*] нельзя использовать для определения старших шестнадцатеричных цифр. Правильной будет запись [*a–fA–F*] (следует обратить внимание на то, что в квадратных скобках допускается указание нескольких диапазонов и (или) перечней символов, например [*X–Z.!x–z;\*$*]).

Во-вторых, считается, что после символа с максимальным значением кода следует символ с минимальным значением (в системе кодирования ASCII после символа с кодом 255 идет символ с кодом 0).

В-третьих, метасимвол минуса нельзя определять «внутри» перечня. Если этот метасимвол нужно определить как символ алфавита описываемого языка, то в первичном выражении он должен быть записан либо сразу после метасимвола «[», либо непосредственно перед метасимволом «]». Так, например, регулярные выражения [–*+\*****/***] или [***+/****\*****–***] порождают каждое ровно четыре цепочки из одного символа:

– **+** *\** **/**

Однако выражение [***+–****\*****/***] в системе кодирования ASCII порождает ровно 256 цепочек длины 1, т. е. цепочки, содержащие все символы этого алфавита.

Нужно еще раз отметить, что допускается записывать перечни и диапазоны последовательно без каких-либо разделителей между метасимволами «[» и «]». Поэтому регулярные выражения [*abk–osx–z*] и [*abklmnosxyz*] являются эквивалентными, т. е. порождают одно и то же множество цепочек символов.

При форматировании текстов программ на экране или бумаге для удобного восприятия их человеком обычно используются такие символы, как табуляция, перевод строки и возврат каретки. Для них в метаязыке регулярных выражений используются общепринятые обозначения (заимствованные из С-подобных языков):

\t – символ табуляции,

\n – символ новой строки,

\r – символ перевода каретки.

Обратный слеш в качестве метасимвола применяется также для представления в словах формального языка таких символов, как \, [, ], ", являющихся одновременно метасимволами языка регулярных выражений:

\\ – один символ \

\[ – символ [

\] – символ ]

\" – символ "

Обратный слеш перед любыми другими символами игнорируется.

Простым называется произвольное регулярное выражение, заключенное в круглые скобки. Например, выражение ([*0–9*]) – простое выражение.

К первичным и простым (в некоторых диалектах – только к первичным) регулярным выражениям могут применяться следующие знаки операций (первые четыре из них называются квантификаторами), используемые для описания цепочек символов, длина которых больше единицы, и пустых цепочек (цепочек нулевой длины).

1. Квантификатор «Пусто или в точности одно» (знак операции ?).

Запись вида [–+]? следует понимать как возможно отсутствующий знак такой части численной константы, как порядок. Это выражение порождает либо пустую цепочку, либо один символ –, либо один символ +.

1. Квантификатор «Одно или несколько» (знак операции +).

Запись вида [*0–9*]+ является типичным определением целой десятичной константы без знака (порождает все возможные непустые цепочки произвольной длины, состоящие из десятичных цифр).

1. Квантификатор «Пусто, одно или несколько» (знак операции \*).

Запись вида [*0–9a–zA–Z*]\* является типичной частью определения идентификаторов и порождает произвольную, возможно пустую цепочку, состоящую из букв и (или) цифр. Полное определение идентификатора может выглядеть так: [*a–zA–Z*][*0–9a–zA–Z*]\*, и порождает цепочки, начинающиеся с любой латинской буквы, за которой в произвольном порядке может следовать сколько угодно латинских букв и (или) цифр.

1. Квантификатор «От *N* до *M*», записываемый либо {*N*,}, либо {,*M*}, либо {*N*,*M*}, где *N* и *M* – целые числа. Такой квантификатор определяет порождение цепочек символов, определяемых регулярным выражением, после которого он записан, длиной от *N* до *M* включительно. Если *N* опущено, то считается, что оно равно нулю. Если *M* опущено, то считается, что оно равно бесконечности. Отсутствие в таком квантификаторе обоих чисел считается недопустимым. Квантификаторы {,1} и {0,1} эквивалентны квантификатору ?, квантификатор {0,} – квантификатору \*, {1,} – квантификатору + и могут быть заменены на эти эквиваленты. Регулярное выражение вида R{*N*,*M*}, где *M* > *N*, всегда можно эквивалентно представить в виде

R (записано *N* раз) R? (записано *M*–*N* раз).

Например, последовательность шестнадцатеричных цифр, имеющая длину 2…4 и определенная как [*0–9a–fA–F*]{2,4}, в эквивалентном представлении выглядит так:

[*0–9a–fA–F*][*0–9a–fA–F*][*0–9a–fA–F*]?[*0–9a–fA–F*]?

К первичным выражениям, простым выражениям, и выражениям, построенным с использованием квантификаторов, могут применяться еще две операции: конкатенации и выбора.

**Операция конкатенации** (не имеющая знака операции)

Два последовательно записанных первичных или простых регулярных выражения понимаются как выражение, порождающее цепочку из двух символов. Первый символ порождается первым выражением, второй символ – вторым. Например, выражение [**/**][*\**] порождает цепочку символов **/***\**, понимаемую обычно как начало блочного (многострочного) комментария в С-подобных языках. Операция конкатенации не коммутативна (выражение [\*][/] порождает совсем другую цепочку символов \*/, отличную от цепочки, порождаемой выражением [/][\*]). Операция конкатенации может применяться к результату другой конкатенации: например, [>][>][=] – это определение знака операции «сдвинуть вправо и присвоить» в языках С/С++. Операция конкатенации может применяться также к регулярным выражениям произвольной сложности, порождающим цепочки, длина которых отлична от единицы.

Во многих диалектах метаязыка регулярных выражений допускаются выражения вида "<*произвольная строка символов*>, каждое из которых порождает цепочку, совпадающую с <*произвольная строка символов*>. Например, выражение *else* считается эквивалентным выражению [e][l][s][e].

**Операция выбора** (знак операции | )

Запись вида [*\**] | [**/**] порождает либо *\**, либо **/**. В некоторых случаях ее использование эквивалентно перечню [***\**/**] (или диапазону). Однако существуют такие группы слов, при определении которых невозможно ограничиться только первичными регулярными выражениями и приходится использовать конкатенацию, выбор и квантификаторы.

Пакет «ВебТрансБилдер» поддерживает все вышеперечисленные способы определения цепочек символов. Однако он не поддерживает некоторые виды выражений, используемые в ряде диалектов метаязыка регулярных выражений, например следующие:

– метасимвол «.» для обозначения произвольного символа определяемого языка. Вместо него в пакете «ВебТрансБилдер» используется пустая пара квадратных скобок [] (следует помнить, что пробел – это тоже символ, и поэтому выражение [] не эквивалентно выражению [ ], содержащему символ пробела);

– знак операции инверсии перечня или диапазона символов «^». Вместо него используются квантифицируемые выражения (или выражение []) в сочетании с выражениями, следующими за ними. Например, запись вида [/][/][]\*[\n] пакетом «ВебТрансБилдер» понимается и обрабатывается так: два символа «косая черта», за которыми следует произвольная цепочка, завершающаяся символом новой строки (это традиционное для С-подобных языков определение однострочного комментария). При использовании знака операции инверсии это выражение могло бы выглядеть так:

[/][/][^\n]\*[\n]

Метаязык регулярных выражений позволяет определять правила образования цепочек символов произвольной длины, являющихся словами формального языка. Некоторые языки (называющиеся словарными), например язык двоичных чисел, могут быть полностью определены на метаязыке регулярных выражений.

Для построения системы лексических правил языка программирования, слова которого образуют несколько групп, играющих разные роли в программах, используются именованные регулярные выражения, называемые регулярными определениями:

<наименование группы слов> : <регулярное выражение>

Совокупность таких правил называется системой регулярных определений, задающих способы порождения нескольких групп слов. Приведем пример системы регулярных определений для групп слов, из которых может состоять оператор присваивания в С-подобных языках (в этом примере не определяются все группы слов языка С):

|  |  |  |
| --- | --- | --- |
| Ident | : | [*a–zA–Z*][*0–9a–zA–Z*]\* |
| Const | : | [*0–9*]+([.][*0–9*]\*)? |
| Const | : | [*0–9*]\*[.][*0–9*]+ |
| WordForFormatting | : | [ \*r*\*n*\*t*]+ |
| SignOfOperation | : | [*–+\*/*] |
| Delimiter | : | [*;*] |
| AssignSign | : | [*=*] |

Далее эта система регулярных определений будет называться RESystem.

Во второй и третьей строках данной системы используется одно и то же имя группы слов для разных регулярных выражений. Этот способ определения допускается во многих диалектах языка регулярных определений, в том числе том, который реализуется пакетом «ВебТрансБилдер». Таким образом, операция выбора в некоторых случаях может быть заменена повторным использованием наименования группы слов в системе регулярных определений.

Система регулярных определений может быть преобразована в оптимальный конечный автомат без памяти, способный распознавать правильные слова всех определенных в ней групп (здесь описан только начальный этап этого преобразования).

Построение лексического акцептора для распознавания слов, заданных некоторой корректной системой регулярных определений, состоит в выполнении следующей последовательности действий (этапов):

* система регулярных определений преобразуется в недетерминированный неоптимальный и не полностью определенный конечный автомат без памяти;
* не полностью определенный недетерминированный неоптимальный конечный автомат преобразуется в оптимальный конечный автомат без памяти;
* управляющая таблица оптимального конечного автомата объединяется с программой, моделирующей поведение автомата. Если система определений содержит действия, то каждое из них встраивается в программную модель.

Подобная организация процесса построения лексического акцептора нужна для того, чтобы каждый из этих этапов можно было легко формализовать для компьютерной реализации.

Конечные автоматы без памяти

Конечный автомат без памяти есть совокупность

***K* = {*A*, *C*, *B*, *G*, *F*}**,

где ***A*** – алфавит входных символов; ***C*** – конечное множество состояний; ***B*** – начальное состояние; ***G*** – функция переходов, по текущему состоянию автомата и входному символу формирующая его состояние в следующий момент времени; ***F*** – конечное множество состояний останова (финальных состояний).

Применительно к программной реализации в трансляторах в качестве основы лексического анализатора конечным автоматом без памяти называется математическая модель устройства, которое:

* имеет один вход и не имеет ни одного выхода;
* в любой момент времени на входе имеет либо в точности один символ входного алфавита ***A***, либо пустую цепочку ε;
* функционирует в дискретном времени ***t*0**, ***t*1**, ***t*2**, ...;
* имеет определенный набор рабочих состояний ***C***, среди которых выделено особое начальное (стартовое) состояние ***С*0**;
* имеет некоторый набор состояний останова (финальных состояний) ***F***;
* при запуске, т. е. в момент времени ***t*0**, всегда оказывается в начальном состоянии ***С*0**, на входе в этот момент оказывается самый первый символ входной цепочки;
* в любой момент времени ***ti*** по текущему состоянию и символу, находящемуся на входе, определяет номер рабочего или финального состояния, в котором автомат окажется в момент времени ***ti+*1**;
* каждому финальному состоянию поставлена в соответствие группа правильных слов, возможно не единственная;
* если по каким-либо причинам номер следующего состояния не может быть определен, то автомат останавливается по ошибке (таких финальных состояний останова может быть несколько);
* при переходе в финальное состояние обнаружения правильного слова входной символ автомата не изменяется, т. е. этот переход фактически выполняется по пустой цепочке ε;
* при переходе в рабочее состояние или состояние останова по ошибке текущий входной символ заменяется следующим символом входной цепочки.

Следует помнить, что применительно к программной реализации автоматов и для решения задачи распознавания последовательности слов переход по пустой цепочке в финальное состояние эквивалентен следующей последовательности операций:

* чтению одного символа с входа;
* обнаружению того, что символ не принадлежит текущему слову;
* возврату символа на вход автомата.

Пусть, например, требуется решить задачу распознавания двоичных чисел, разделенных последовательностями пробелов. Простая система регулярных определений может описать слова этих двух групп:

BinaryNumber : [01] +

Space : [ ] +

Существует как минимум три способа определения конечного автомата без памяти, способного распознавать такие слова.

Конечный автомат без памяти может быть задан только функцией переходов, если соблюдаются определенные соглашения о способе нумерации состояний (начальным является состояние номер ***0***):

{текущее состояние; входной символ; следующее состояние}

Пример такого определения автомата:

*{****0***,ε*,****-1****} {****0***,*\d32*,***1****} {****0***,*0*,***2****} {****0***,*1*,***2****} {****1***,*\d32*,***1****} {****1***,ε,***-2****} {****2***,*0*,***2****} {****2***,*1*,***2****} {****2***,ε,***-3****}*

Алфавит входных символов может быть извлечен из функции переходов, это символы *0*, *1, пробел* (символ с десятичным кодом *\d32*). Любые другие символы, которые могут появиться на входе автомата, здесь обозначены пустой цепочкой ε, их обработка состоит в переходе автомата в финальные состояния*.*

Состояния ***0***, ***1*** и ***2*** являются рабочими, поскольку для них определены переходы в другие состояния по входным символам. Состояния ***-1***, ***-2*** и ***-3*** – финальные, так как из них не существует ни одного перехода. Состояние ***-1*** соответствует обнаружению конца цепочки символов, содержащей только правильные слова, т. е. двоичные числа и последовательности пробелов. Останов автомата в состоянии ***-2*** означает обнаружение последовательности пробелов, а в состоянии ***-3*** – обнаружение двоичного числа.

При программной реализации конечные автоматы без памяти обычно представляются либо графами состояний и переходов, либо управляющими таблицами.

Графом состояний и переходов (ГСП) конечного автомата без памяти (далее просто конечного автомата, КА) называется помеченный ориентированный граф, вершины которого сопоставлены состояниям, а дуги – переходам.

Разметка вершин обычно производится целыми числами, обозначающими номера состояний. Имеется особая начальная вершина (обычно с нулевым номером), в которую не входит ни одна дуга. Дуги графа могут быть помечены обозначением пустой цепочки ε, одиночным символом, перечнем или диапазоном символов.

Существуют особые финальные вершины, из которых не может выходить ни одна дуга. Имеется одна или несколько рабочих вершин, в которые могут входить дуги и из которых могут выходить дуги.

Пример списка состояний и переходов конечного автомата (графа, только в текстовом представлении),распознающего двоичные числа и последовательности пробелов в представлении, формируемом пакетом «ВебТрансБилдер»:

|  |  |  |
| --- | --- | --- |
| **0:** |  | *EOF*-> **-1** [\d32] -> **1** [01]-> **2** |
| **1:** |  | [*other*]-> **-2** [\d32] -> **1** |
| **2:** |  | [*other*]-> **-3** [01]-> **2** |

Разметка дуг графа производится с помощью указания одного из символов входного алфавита того языка, для распознавания слов которого построен данный автомат, либо обозначения пустой цепочки символов ε, которому здесь соответствует [*other*]. Это обозначение надо понимать так: любой символ, не принадлежащий разметке какой-либо дуги, выходящей из данного состояния. Поэтому в разных состояниях [*other*] обозначают разные множества символов. Для состояния **1** это не пробел, а для состояния **2** – любой символ, кроме двоичных цифр. Если дуга помечена символом входного алфавита, то при переходе по этой дуге автомат заменяет данный входной символ следующим символом из входной цепочки. При переходе по дуге, помеченной обозначением пустой цепочки ε, символ на входе автомата не изменяется.

Еще одним способом определения конечного автомата без памяти является табличный способ. Автомат задается прямоугольной таблицей, строки которой обычно соответствуют состояниям, а столбцы – входным символам. Номера состояний и входные символы показаны в заголовках строк и столбцов, однако следует помнить, что заголовки строк и столбцов не являются элементами таблицы. В клетках таблицы указываются номера состояний перехода (из состояния, в строке которого находится клетка, по символу, указанному в заголовке столбца). Пример формируемой пакетом «ВебТрансБилдер» управляющей таблицы (УТ) автомата, предназначенного для акцепта двоичных чисел, показан в виде табл. 1.

Таблица 1

|  |  |  |  |
| --- | --- | --- | --- |
|  | **0** | **1** | **2** |
| [ ► ] | **–1** | **–2** | **–3** |
| [ \d32 ] | **1** | **1** | **–3** |
| [ 01 ] | **2** | **–2** | **2** |

Так называемая рабочая зона управляющей таблицы содержит только клетки с переходами (верхняя строка таблицы не входит в рабочую зону). Если в рабочей зоне управляющей таблицы нет пустых клеток, то автомат называется полностью определенным. В противном случае автомат называется не полностью определенным.

Историей работы конечного автомата без памяти для данной входной цепочки называется упорядоченная (по моменту дискретного времени, т. е. номеру шага) совокупность троек значений *{****t***,***a***,***c****}*, где ***t*** – момент дискретного времени; ***a*** – текущий входной символ; ***c*** – текущее состояние. Для любой конечной по количеству символов входной цепочки автомат, запущенный в начальном состоянии, завершит свою работу за конечное число шагов, т. е. реализует конечную историю работы.

Историю работы автомата удобно представлять в виде таблицы, столбцы которой содержат значения троек. В качестве момента времени ***ti*** указывается порядковый номер ***i***такта работы автомата.

Пусть дана входная цепочка, содержащая шесть символов: «*101 10».* История работы автомата, заданного табл. 1, по разбору этой цепочки будет выглядеть так, как показано в табл. 2. Четвертый символ цепочки (символ пробела) в истории представлен двоичным кодом (*\d32*), а конец цепочки, т. е. отсутствие символов для обработки, – специальным обозначением ►, соответствующим понятию EndOfFile.

Таблица 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Такт | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| Символ | *1* | *0* | *1* | *\d32* |  | *\d32* | 1 |  | 1 | 0 | ► |  | ► |  |
| Состояние | ***0*** | ***2*** | ***2*** | ***2*** | ***-3*** | ***0*** | ***1*** | ***-2*** | ***0*** | ***2*** | ***2*** | ***-3*** | ***0*** | ***-1*** |

Остановы автомата в состоянии -***3*** свидетельствуют об обнаружении двоичных чисел, в состоянии ***-2*** – об обнаружении последовательности пробелов, а в состоянии ***-1*** – об обнаружении конца входной цепочки.

Конечный автомат, заданный вышеприведенными графом или таблицей, для любой конкретной входной цепочки при каждом запуске будет реализовывать одну и ту же историю работы. Такие автоматы называются детерминированными.

Конечный автомат без памяти является детерминированным, если для обработки любой входной цепочки символов при любом запуске он реализует одну и ту же последовательность переходов из состояния в состояние.

Существуют и недетерминированные автоматы, которые могут отрабатывать разные последовательности переходов из состояния в состояние при различных запусках для одной и той же входной цепочки символов.

Недетерминированностью первого рода называется наличие перехода из некоторого состояния по пустой цепочке ε не в финальное, а в какое-либо рабочее состояние. При формальном следовании правилам работы автомата это приводит к возможности перехода при разных запусках из такого состояния либо по текущему входному символу со сменой его на следующий, либо по ε с сохранением текущего символа на входе.

Недетерминированностью второго рода называется наличие нескольких переходов из некоторого состояния по одному и тому же символу в разные состояния, что также приводит к возможности реализации разных последовательностей переходов из состояния в состояние при разных запусках.

Конечный автомат может содержать недостижимые состояния (в которые не существует никакого пути из начального состояния), тупиковые состояния (из которых не существует ни одного пути в какое-либо финальное состояние) и пары эквивалентных состояний (такие, что все переходы из них по одинаковым символам ведут в одни и те же состояния).

Автомат называется оптимальным, если любое уменьшение количества строк или столбцов его управляющей таблицы приводит к изменению его функционала (способности попадать в нужное финальное состояние при обработке всех правильных слов всех заданных групп), и неоптимальным – в противном случае. Неоптимальный конечный автомат может быть эквивалентно без потери функционала преобразован в оптимальный.

Преобразование системы регулярных определений в недетерминированный неоптимальный конечный автомат

Для такого преобразования конечный автомат без памяти вначале представляется в виде графа состояний и переходов (ГСП). Построение конечного автомата начинается с образования начальной вершины с номером ***0***, общей для всех регулярных определений системы (рис. 11).

Для каждого именованного регулярного выражения РВ i образуются:

1. промежуточная вершина ***pi***;
2. условная дуга перехода из начальной вершины графа в промежуточную вершину, помеченная всем регулярным определением РВ*i* (условность отражена прерывистостью дуги);
3. финальная вершина ***f i***, соответствующая акцепту слов данной группы;
4. дуга перехода из промежуточной вершины ***pi*** в финальную ***fi***, помеченная обозначением пустой цепочки.

**Финальная вершина длясловльная вер**

**Финальная вершина ершинаая вершина д**

**Финальная вершина дляܡсловльная ве*n***

**ε**

**ε**

РВ2

РВn

***0***

***p*1**

***p*2**

***pn***

***f*1**

***f*2**

***fn***

·

·

·

РВ1

**ε**

*Рис. 11*.Начальный вид графа состояний и переходов

Если для некоторой группы слов в системе определено несколько одинаково поименованных регулярных выражений, то для нее образуется единственная финальная вершина и строится несколько путей в эту вершину из начальной. Пусть для группы 2 задано два регулярных определения РВ21 и РВ22 (слова группы Const в вышеприведенной системе регулярных определений). В подобном случае граф автомата будет выглядеть следующим образом (рис. 12).

**Финальная вершина для слов группы 1**

**Финальная вершина для слов группы 2**

**Финальная вершина для слов группы n**

**ε**

**ε**

**ε**

РВ 1

РВ 21

РВ*n*

***0***

***p*1**

***pn***

***f*1**

***f*2**

***fn***

***p*21**

***p*22**

РВ 22

**ε**

**·**

**·**

**·**

*Рис. 12.* Начальный вид ГСП при множественном определении групп слов

Отметим, что из практических соображений любая система регулярных определений перед преобразованием явно или неявно должна быть расширена определением слова «EndOfFile», образующего отдельную специальную группу. Этим словом заканчивается любая правильная программа на любом языке программирования. Регулярное выражение, определяющее эту группу слов, обычно автоматически и неявно добавляется преобразователем (и пакетом «ВебТрансБилдер») в заданную систему регулярных определений.

Далее условная дуга каждого фрагмента графа, имеющего вид, показанный на рис. 13, преобразуется согласно табл. 3 в другой фрагмент, содержащий те же самые начальную и конечную вершины, но вместо одной условной дуги перехода появляется либо безусловная дуга, либо новые вершины и дуги.

РВ*i*

*Начальная вершина фрагмента*

*Конечная вершина фрагмента*

*Рис. 13.* Преобразуемый фрагмент ГСП

Этот процесс продолжается до тех пор, пока все дуги графа не окажутся помечены одиночным символом, или перечнем символов, или диапазоном символов, или обозначением пустой цепочки.

Приведенные в табл. 3 способы преобразования строго соответствуют очевидным правилам поведения автомата для распознавания символьных цепочек, определяемых каждым из способов образования сложного регулярного выражения из более простых.

В том случае если регулярное выражение, помечающее условную дугу перехода, является первичным, условная дуга просто заменяется на дугу безусловного перехода с соответствующей разметкой. В противном случае согласно правилам образования регулярных выражений любое РВ*i* можно представить как одно из выражений строк 2–6 первого столбца табл. 3 и заменить условную дугу перехода исходного фрагмента на новый фрагмент из третьего столбца таблицы с сохранением начальной и конечной вершин. При этом новые условные дуги переходов будут помечены составными частями исходного регулярного выражения, т. е. более простыми подвыражениями. Следовательно, для любого регулярного выражения, имеющего конечную длину, за конечное число шагов может быть получен граф состояний и переходов, не содержащий условных дуг переходов.

Таблица 3

**Способы преобразования фрагментов ГСП**

|  |  |  |  |
| --- | --- | --- | --- |
| Вид выражения | Начальная вершина | Новый фрагмент | Конечная  вершина |
| 1. Первичное:  [*x*] или [*xyz*] или [*x–z*] … |  | *x* или *xyz* или *x–z* … |  |
| 2. РВ1 РВ2 | РВ 2  РВ 1 |  |  |
| 3. РВ1 | РВ2 | РВ 2  **ε**  РВ 1  **ε** |  |  |
| 4. РВ? | **ε**  РВ  **ε** |  |  |
| 5. РВ+ | РВ  **ε**  РВ |  |  |
| 6. РВ\* | РВ  **ε**  РВ  **ε** |  |  |

Приведенные в табл. 3 способы преобразования строго соответствуют очевидным правилам поведения автомата для распознавания символьных цепочек, определяемых каждым из способов образования сложного регулярного выражения из более простых.

В том случае если регулярное выражение, помечающее условную дугу перехода, является первичным, условная дуга просто заменяется на дугу безусловного перехода с соответствующей разметкой. В противном случае согласно правилам образования регулярных выражений любое РВ*i* можно представить как одно из выражений строк 2–6 первого столбца табл. 3 и заменить условную дугу перехода исходного фрагмента на новый фрагмент из третьего столбца таблицы с сохранением начальной и конечной вершин. При этом новые условные дуги переходов будут помечены составными частями исходного регулярного выражения, т. е. более простыми подвыражениями. Следовательно, для любого регулярного выражения, имеющего конечную длину, за конечное число шагов может быть получен граф состояний и переходов, не содержащий условных дуг переходов.

Пронумеруем в произвольном порядке все рабочие вершины этого графа числами ***1***, ***2***, .., ***m***, затем все финальные вершины числами ***m***+ ***1***, ***m***+ ***2***, ..., ***n*** и преобразуем граф состояний и переходов автомата в управляющую таблицу.

Построенный автомат, вероятнее всего, будет недетерминированным и неоптимальным. В качестве иллюстрации можно привести конечный автомат без памяти, построенный описанным выше способом по системе определений RESystem. Граф состояний и переходов этого автомата показан на рис. 14.

Особое внимание следует обратить на тот фрагмент графа состояний и переходов, который обеспечивает распознавание констант. Для этой группы слов в системе задано два различных регулярных определения:

|  |  |  |
| --- | --- | --- |
| Const | : | [*0–9*]+([.][*0–9*]\*)? |
| Const | : | [*0–9*]\*[.][*0–9*]+ |

Соответственно этим определениям в графе автомата (см. рис. 14) образовано два фрагмента с общими вершинами, начальной и финальной. Первое из выражений является конкатенацией двух более простых выражений [*0–9*]+ и ([.][*0–9*]\*)?, поэтому в процессе преобразования условная дуга, помеченная этим выражением, была заменена двумя условными дугами, помеченными этими подвыражениями, и образована промежуточная вершина номер *5*.

Затем для каждого из подвыражений по таблице соответствия условные дуги были заменены на нужные фрагменты. В результате образовались вершины ***4*** и ***6*** с соответствующими дугами. Продолжение этого процесса привело к образованию представленного графа (на рис. 14 не показана финальная вершина номер -***1***, соответствующая акцепту слова «EndOfFile»).

**Акцепт:  
Formatting**

*0-9*

*0-9*

***10***

*0-9a-zA-Z*

*0-9a-zA-Z*

*a-zA-Z*

*0-9*

**ε**

*0-9*

*0-9*

•

***11***

*0-9*

***12***

***13***

•

*0-9*

**Акцепт:  
Const**

**Акцепт:  
Ident**

***14***

***15***

***16***

***17***

└┘\*r*\*n*\*t*

**Акцепт:  
OpSign**

**Акцепт:  
Delimiter**

**Акцепт:  
AsSign**

└┘\*r*\*n*\*t*

**– +** \***/**

**;**

***=***

**ε**

***-2***

***-4***

**ε**

**ε**

**ε**

**ε**

**ε**

**ε**

**ε**

**ε**

**ε**

**ε**

**ε**

**ε**

**ε**

*0-9*

**ε**

***-5***

***-6***

***-7***

***-3***

*Рис. 14*. Недетерминированный неоптимальный автомат, построенный   
по системе определений RESystem

Недетерминированный неоптимальный конечный автомат без памяти далее преобразуется в детерминированный оптимальный. Алгоритмы такого преобразования и алгоритмы функционирования его результата изучаются при выполнении работы № 2.

1. Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Пример1):
   1. Изучить интерфейс пакета ВебТрансБилдер: запуск, регистрация, состав элементов основного окна, команды меню. Используя справку ВебТрансБилдера (команда меню «Помочь»), изучить структуру таблицы системы редактируемых правил основного окна, приемы и способы формирования/редактирования ее содержимого.
   2. Освоить:
      * настройку и сохранение личных параметров (пункт меню «Настроить»);
      * открытие имеющейся в базе данных системы правил (пункт меню «Открыть»);
      * редактирование правил (щелчок левой кнопкой мыши по заполненной строке таблицы «Видимые-редактируемые правила»);
      * автодобавление правила (щелчок левой кнопкой мыши по последней (пустой) строке таблицы «Видимые-редактируемые правила»);
      * операции сортировки таблицы правил (щелчок левой кнопкой мыши по клетке «Левая часть» заголовка таблицы);
      * скрытие/отображение действий (щелчок левой кнопкой мыши по клетке «Правая часть» заголовка таблицы);
      * добавления пустых строк, удаления, вырезания и вставки правил (щелчок правой кнопкой мыши по правилу в таблице);
      * сохранение системы правил (пункт меню «Сохранить как»).
   3. Изучить технологию разработки систем регулярных определений пакета ВебТрансБилдер, ориентируясь на свой вариант задания на курсовую работу. Освоить использование простых регулярных выражений (один символ, перечень символов, диапазон символов) и квантификаторов «?», «\*» и «+», операций группировки «( …)» и конкатенации (не имеющей знака операции).
   4. Разработать и сохранить систему правил для всех групп слов языка(или, как минимум, для идентификаторов, констант, знаков операций, пробельных последовательностей), заданного в курсовой работе.
   5. Построить вручную недетерминированный граф состояний и переходов по разработанной системе правил как результат выполнения первого этапа алгоритма преобразования системы регулярных определений в конечный автомат (см. п. 2.4.2.1 в [1]). Граф необязательно рисовать, его можно включить в отчет в форме списка вершин и выходящих из вершин маркированных дуг наподобие того, как это делает ВебТрансБилдер.
   6. Освоить построение транслятора как совокупности лексического анализатора (сканера) и синтаксического анализатора (парсера, пока - заглушки) средствами пакета ВебТрансБилдер (пункт меню «Построить») и просмотра визуального представления его элементов: графа состояний и переходов и управляющей таблицы (пункты меню «Показать/Сканер, управляемый таблицей» и «Показать/Сканер, управляемый графом»).
   7. Построить простейший транслятор, освоить запуск транслятора при использовании инструментального языка JavaScript (пункт меню «Запустить»), освоить выгрузку и сохранение кода построенного транслятора при использовании других инструментальных языков (пункт меню «Скачать»). Изучить код построенного транслятора (пункт меню «Показать/Код транслятора»), найти в нем вставленное расширение (var ignoreLastWord;) и разобраться в том, как в сканере (лексическом анализаторе) используется определяемая в нем переменная.
   8. Проверить работоспособность построенного транслятора на примере нескольких последовательностей правильных слов заданного языка (тех слов, которые определены разработанной и сохраненной системой правил), убедиться, что транслятор не воспринимает неправильные слова.
   9. Подготовить, сдать и защитить отчет к лабораторной работе.
2. Требования к содержанию отчета.

Отчет должен содержать:

* + - цель работы;
    - перечень групп слов учебного языка, заданного на курсовую работу;
    - примеры правильных слов заданного языка;
    - результаты разработки фрагмента системы правил языка, заданного на курсовую работу, описание разработанных регулярных выражений;
    - граф состояний и переходов недетерминированного конечного автомата (результат выполнения пункта 4.5);
    - граф состояний и переходов сканера, построенного ВебТрансБилдером по разработанному фрагменту, описание алгоритма работы автомата, управляемого графом;
    - управляющая таблица сканера, построенного ВебТрансБилдером по разработанному фрагменту, описание алгоритма работы автомата, управляемого таблицей;
    - результаты тестирования транслятора (пока – только лексического анализатора, т.е. сканера), построенного ВебТрансБилдером по разработанной системе правил;
    - выводы и заключение

1. Контрольные вопросы.
   1. Опишите состав и назначение компонентов учебного пакета автоматизации проектирования трансляторов ВебТрансБилдер.
   2. Перечислите основные функции пакета ВебТрансБилдер.
   3. С помощью каких пунктов (и подпунктов) меню осуществляются основные операции пакета ВебТрансБилдер?
   4. Как можно создать новое правило?
   5. Где пишутся действия в лексических правилах?
   6. Что такое регулярное выражение?
   7. Что такое квантификатор?
   8. Какие квантификаторы можно использовать в пакете ВебТрансБилдер?
   9. Что такое шаблон построения транслятора?
   10. Какие знаки операций можно использовать в регулярных выражениях?
   11. Как в регулярном выражении определить диапазон символов?
   12. Может ли внутри одной пары квадратных скобках быть записано несколько диапазонов символов?
   13. Как можно изменить личные настройки в пакете ВебТрансБилдер?
   14. Может ли быть создано несколько одноименных регулярных определений?
   15. Как можно задать лексическое правило для слова, содержащего метасимволы языка регулярных выражений, такие как '[', ']', '\', …?
   16. Каковы приоритеты знаков операций в языке регулярных выражений пакета ВебТрансБилдер?
   17. Для чего предназначен пункт меню «Показать»?
   18. Что такое действие в лексическом правиле?
   19. Как с помощью действий можно заблокировать возврат правильно распознанного слова из лексического анализатора?
   20. На какие части разделено основное окно клиентской части пакета ВебТрансБилдер?

## Лабораторная/практическая работа № 2

1. Название работы: «Лексика языков программирования. Конечные автоматы без памяти для обнаружения слов в тексте программы».
2. Цели работы: изучение конечных автоматов (КА) без памяти, способов определения КА в трансляторах – графового и табличного, методов построения недетерминированного КА по системе регулярных выражений, методов эквивалентных преобразований недетерминированных КА в оптимальные полностью определенные КА – лексические акцепторы.
3. Основные теоретические сведения:

Оптимальность конечных автоматов без памяти

Оптимальным в множестве эквивалентных друг другу автоматов называется полностью определенный детерминированный автомат, управляющая таблица которого имеет минимальное количество клеток в рабочей зоне.

Оптимальный автомат удовлетворяет следующим критериям.

1. Множества символов, помечающие столбцы управляющей таблицы, попарно не пересекаются.
2. Автомат не содержит недетерминированностей первого рода.
3. Автомат не содержит недетерминированностей второго рода.
4. Автомат не имеет тупиковых состояний.
5. Автомат не имеет недостижимых состояний.
6. Все рабочие состояния попарно не являются эквивалентными.
7. В рабочей зоне управляющей таблицы нет одинаковых столбцов.
8. В рабочей зоне управляющей таблицы нет пустых клеток.

Преобразование недетерминированного неоптимального автомата в детерминированный оптимальный автомат можно осуществить путем последовательной проверки всех этих критериев. Если автомат не удовлетворяет проверяемому критерию, то следует выполнить эквивалентное преобразование автомата, устраняющее нарушение этого критерия.

После того как будут проверены все критерии и выполнены все требуемые эквивалентные преобразования, автомат станет детерминированным оптимальным.

Устранение пересечений множеств символов разметки столбцов управляющей таблицы (критерий 1)

Каждый столбец управляющей таблицы, имеющий в заголовке множество из *k* > 1 символов, следует заменить на *k* столбцов, поместив в их заголовки в точности по одному символу из множества разметки исходного столбца.

При этом в таблице могут возникнуть столбцы, имеющие одинаковые заголовки. Следует объединить все одинаково помеченные столбцы. При этом могут возникнуть в явном виде недетерминированности второго рода.

Устранение недетерминированностей (критерии 2 и 3)

Основная идея этого преобразования состоит в том, чтобы поставить в соответствие каждому подмножеству состояний (вершин в графе состояний и переходов) исходного автомата, в которых он может оказаться «одновременно» в результате недетерминированных переходов, в точности одно состояние (вершину графа) эквивалентного ему детерминированного автомата.

Если исходный автомат имеет *n* состояний, то множество всех непустых подмножеств его состояний включает ровно 2*n*– 1 подмножеств. Начальному состоянию (вершине *s*0) исходного автомата соответствует подмножество {*s*0}, содержащее единственную начальную вершину и являющееся, по сути, начальной вершиной результата преобразования.

Остальные состояния детерминированного автомата и переходы между состояниями формируются путем выполнения алгоритма, в котором используются отмеченные и неотмеченные множества вершин исходного автомата.

**Шаг 1.** Выполнение преобразования начинается с формирования начального подмножества состояний, в которое в этот момент включается единственная вершина с номером 0 исходного ГСП. Это подмножество включается в набор неотмеченных подмножеств.

**Шаг 2.** Основной цикл: организуется перебор и обработка неотмеченных подмножеств. Если ни одного такого подмножества нет, то выполняется выход из основного цикла и переход к шагу 3. Каждое подмножество вершин после обработки помечается, т. е. исключается из перечня неотмеченных. Обработка неотмеченного подмножества заключается в следующих действиях.

**Шаг 2.1.** Построение ε –замыкания текущего подмножества вершин: в текущее подмножество включаются все те рабочие (не финальные) вершины, в которые есть дуги переходов, помеченные пустой цепочкой εиз какой бы то ни было вершины текущего подмножества. Процесс замыкания осуществляется путем перебора всех переходов по пустой цепочке εиз вершин текущего подмножества до тех пор, пока оно не перестанет изменяться или не будет исчерпан перечень таких переходов

**Шаг 2.2.** Поиск сформированного подмножества среди всех ранее обработанных (отмеченных) подмножеств вершин. Если в точности такого подмножества вершин еще не было образовано, то выполняется добавление его к перечню отмеченных подмножеств.

**Шаг 2.3.** Определение набора символов, помечающих все дуги переходов из вершин текущего подмножества (в том числе символа ε, помечающего дуги переходов в финальные состояния исходного автомата).

**Нет в УП**

**Шаг 2.4.** Внутренний цикл: перебор всех символов набора. Для каждого из символов построенного на шаге 2.3 набора (обозначим такой выбранный символ как *x*) выполняется формирование нового неотмеченного подмножества вершин.

**Шаг 2.5.** Замыкание. В полученное подмножество включаются вершины, в которые ведут дуги переходов из состояний текущего подмножества, помеченные выбранным символом. Каждое сформированное подмножество вершин добавляется к перечню неотмеченных подмножеств, но только в том случае, если его там еще нет. Формируется дуга перехода из текущего подмножества вершин во вновь сформированное подмножество, помеченная символом *x*.

**Шаг 3.** Завершение. Сопоставление в точности одной вершины нового графа каждому образованному в результате выполнения шагов 1 и 2 подмножеству вершин исходного графа. Если в рассматриваемом множестве есть хотя бы одна финальная вершина исходного автомата, то и поставленная ему в соответствие вершина нового графа считается финальной.

Ассоциированная с финальной вершиной исходного графа группа слов теперь считается ассоциированной с финальной вершиной результата преобразования. Наличие нескольких финальных вершин исходного графа в одном подмножестве может свидетельствовать о внутренней противоречивости исходной системы регулярных определений лексики языка. Действительно, останов полученного конечного автомата в таком финальном состоянии не позволяет получить исчерпывающей информации о том, слово какой группы им обнаружено. В большинстве языков программирования лексические правила образования служебных слов языка совпадают с правилами образования идентификаторов. В подобных случаях следует тщательно проанализировать систему регулярных определений и, возможно, модифицировать ее с целью ликвидации обнаруженной неоднозначности.

Другим возможным выходом из сложившейся ситуации является написание дополнительного действия, связанного с неоднозначно распознаваемыми группами слов, и предназначенного для выполнения дополнительного анализа текста обнаруженного слова, а также принятия решения об отнесении его к той или иной группе.

Автомат, полученный путем выполнения этого алгоритма над ГСП исходного автомата, не будет содержать недетерминированностей ни первого, ни второго рода по следующим причинам:

–в результате выполнения ε**-**замыкания каждого подмножества (шаг 2.1) ликвидируются все недетерминированности первого рода;

– за счет формирования единого подмножества вершин с использованием всех дуг переходов по символу *x* из текущего подмножества на шаге 2.4 ликвидируются недетерминированности второго рода.

Удаление тупиковых состояний (критерий 4)

Найти все тупиковые состояния, если они есть. Сначала необходимо определить списки достижимых финальных состояний для каждого рабочего состояния. При представлении конечного автомата в виде управляющей таблицы это делается следующим образом.

**Шаг 1.** С каждой строкой ***i*** управляющей таблицы сопоставляется список ***Si*** (в начальный момент он пустой) достижимых финальных состояний, в которые есть переходы из данной строки.

**Шаг 2.** Просматриваются все строки таблицы начиная с нулевой.

**Шаг 3.** В каждой строке (пусть ее номер ***n***) просматриваются все клетки.

**Шаг 4.** Если клетка не пуста и содержит номер финального состояния, то этот номер добавляется к списку ***Sn***, если его нет в этом списке.

**Шаг 5.** Если клетка не пуста и содержит номер рабочего состояния ***k*** (***k*** не равно ***n***), то все элементы списка ***Sk*** финальных состояний строки ***k*** добавляются к списку ***Sn*** финальных состояний строки ***n*** (естественно, если их нет в списке ***Sn***).

**Шаг 6.** Если была просмотрена последняя строка рабочей зоны и в процессе просмотра изменился хотя бы один список ***Si***, то нужно вернуться к шагу 2, иначе процесс завершается.

Все состояния, для которых списки достижимых финальных состояний пусты, являются тупиковыми, и их следует удалить. Для этого необходимо:

1) удалить из всех клеток таблицы все переходы в выявленные тупиковые состояния;

2) удалить строки тупиковых состояний из управляющей таблицы.

Удаление недостижимых состояний (критерий 5)

Найти все недостижимые состояния, если они есть. Для этого выполним следующие шаги (предполагается, что автомат задан управляющей таблицей).

**Шаг 1.** Сопоставим с каждой строкой рабочей зоны управляющей таблицы (кроме нулевой строки) отметку «недостижима», а с нулевой строкой – отметку «достижима».

**Шаг 2.** Просмотрим все «достижимые» строки начиная с нулевой.

**Шаг 3.** В каждой строке просмотрим все клетки. Если в клетке находится номер рабочего состояния ***k***, то значение отметки строки ***k*** изменим на значение «достижима».

**Шаг 4.** Если просмотрена последняя строка таблицы и хотя бы для одной строки отметка изменилась со значения «недостижима» на значение «достижима», то возвращаемся к шагу *2*, иначе процесс завершается.

Все строки, отметка которых имеет значение «недостижима», следует удалить из таблицы.

Слияние эквивалентных состояний (критерий 6)

Для обнаружения таких состояний необходимо просмотреть все возможные пары строк рабочей зоны управляющей таблицы.

Если найдется пара таких строк ***i*** и ***j***, содержимое которых либо полностью совпадает, либо различается только в клетках одного и того же столбца и только переходами внутри этой пары ({***i***→***i*** и ***j***→***j***}, или {***i***→***j*** и ***j***→***i***}, или {***i***→***j*** и ***j***→***j***}, или {***i***→***i*** и ***j***→***i***}), то нужно выполнить следующее:

1) удалить любую из них (например, строку ***j***) из таблицы;

2) во всех клетках остальных строк таблицы заменить переходы в удаленное состояние ***j*** переходами в оставленное состояние ***i***.

Слияние одинаковых столбцов (критерий 7)

Просмотреть попарно все столбцы управляющей таблицы. Если найдется пара одинаковых столбцов, помеченных множествами символов *M*1 и *M*2, то выполнить следующие действия:

1) заголовок одного из столбцов заменить объединением множеств символов *M*1 и *M*2;

2) удалить другой столбец;

3) продолжить просмотр.

Такое преобразование не может привести к нарушению ни одного из критериев. В графе состояний и переходов это преобразование состоит в замене параллельных дуг одной дугой, помеченной объединением разметок заменяемых дуг.

При использовании управляющей таблицы, столбцы которой помечены множествами символов, моделирующая поведение конечного автомата программа обязана выполнять преобразование кода каждого входного символа в индекс столбца таблицы. Такое преобразование обычно производится с использованием вспомогательной одномерной таблицы, называемой транслитератором. Транслитератор – это просто одномерный целочисленный массив, каждый элемент которого содержит индекс столбца управляющей таблицы для символа, код которого рассматривается как индекс элемента.

Формирование полностью определенного автомата (критерий 8)

В результате последовательного выполнения всех этих преобразований будет получена управляющая таблица автомата, содержащая столбец, помеченный обозначением пустой цепочки. Некоторые клетки этой таблицы могут быть пусты. Каждая непустая клетка содержит единственный номер состояния перехода.

Строки управляющей таблицы могут оказаться пронумерованными не по порядку, поскольку одни состояния могли быть добавлены, а другие – удалены. Нужно перенумеровать их последовательно, начиная с нуля и одновременно приводя в соответствие все переходы на те состояния, номера которых изменяются.

Для преобразования этого автомата в полностью определенный конечный автомат без памяти необходимо точно сформулировать смысл понятия пустой цепочки символов, положенный в основу способа формирования графа состояний и переходов по системе регулярных определений.

Пустую цепочку символов можно (а в автоматной модели для не полностью определенных автоматов – нужно) понимать как последовательность, состоящую из трех действий:

* чтения входного символа;
* анализа символа (проверка наличия перехода в рабочее состояние из данного состояния по данному символу);
* возврата символа на вход, если такого перехода нет.

Теперь ясно, что если в некоторой строке управляющей таблицы клетка в столбце, помеченном обозначением пустой цепочки символов, содержит номер состояния перехода, то этот номер следует занести во все пустые клетки данной строки. Тем самым как раз и будут реализованы переходы по пустой цепочке символов.

Если же такая клетка пуста, то во все пустые клетки данной строки необходимо занести номер состояния останова по обнаружению лексической ошибки. В зависимости от целей, которые преследуются при построении автомата, можно либо предусматривать единственное состояние останова по ошибке, либо множество таких состояний. В последнем случае номер состояния, в котором остановился автомат, может использоваться для формирования развернутых диагностических сообщений.

Наконец, обозначение пустой цепочки в заголовке столбца необходимо заменить обозначением конца файла (EOF), которое не следует путать со словом «EndOfFile».

1. Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample2):
   1. Используя пакет «ВебТрансБилдер», освоить:
      * создание лексических правил на языке регулярных выражений (РВ);
      * построение сложных регулярных выражений с использованием способа определения любого символа в виде «[]», квантификаторов вида {<число>,<число>} и операций выбора «|» языка РВ.
   2. Разработать (доработать разработанную при выполнении работы № 1) систему правил для всех групп слов языка, определенного заданием на курсовую работу (в том числе комментариев, которые заданием не определены, но в программах на любом языке необходимы). Построить по этой системе:
      * сканер, управляемый графом состояний и переходов;
      * сканер, управляемый таблично.
   3. Изучить структуру программных модулей, построенных пакетом «ВебТрансБилдер». Используя операцию меню «Показать / Код транслятора», найти и изучить в этих кодах функции лексического акцепта для графового и табличного способов реализации КА. Сравнить реализации конечных автоматов, управляемых различными способами. Изучить способ реализации вызова действий, определенных в лексических правилах, и алгоритм работы формирователя лексем.
   4. Проверить функционирование конечных автоматов, построенных пакетом «ВебТрансБилдер»:
      * подготовить тестовый пример программы на языке, заданном на курсовую работу (пример должен содержать слова абсолютно всех групп языка);
      * запустить каждый автомат на выполнение, протрассировать с использованием отладчика браузера (для открытия отладчика нажать функциональную клавишу F12) вручную работу лексического акцептора и в графовой, и в табличной реализации. Для этого расставить точки останова в лексическом акцепторе и лексическом анализаторе (функции getLexem, имеющиеся и в акцепторе, и в анализаторе) и проследить процессы лексического акцепта и анализа тестовой программы (программа должна содержать несколько слов из групп «идентификаторы», «константы» и «пробелы»);
      * убедиться в работоспособности автоматов, в противном случае доработать систему РВ и добиться правильности функционирования лексического акцептора.
   5. Построить вручную в виде двух таблиц из трех строк каждая (номер шага, текущий входной символ, текущее состояние автомата) две истории работы табличного и графового автоматов как результаты обработки ими двух различных операторов присваивания языка, заданного на курсовую работу (например, x\_Yz <= 2.37; a\_B <= 34;). Сравнить последовательности обнаруженных слов в построенных историях работы с теми, которые формируются при тестировании сканера, и объяснить расхождения, если они имеются.
   6. Разработать полное описание лексики учебного языка, заданного на курсовое проектирование. Описание лексики должно охватывать все группы слов языка. Для каждой группы должны быть сформулировано следующее:
      * назначение (или способы использования) слов этой группы в программах на учебном языке;
      * правила формирования слов языка из символов алфавита (или перечень допустимых слов);
      * характеристики слов, если они имеют значение для синтаксического и семантического анализа (например, приоритеты знаков операций, диапазоны значений констант);
      * примеры правильных слов.
2. Подготовить отчет, который должен содержать следующие разделы:
   * + цель работы;
     + доработанную систему правил, определяющую полностью все группы слов языка, заданного на курсовую работу;
     + управляющую таблицу и граф состояний и переходов сканеров, построенных пакетом «ВебТрансБилдер» по этой системе, а также краткое описание алгоритмов функционирования этих автоматов;
     + историю работы сканера, управляемого таблицей, построенную путем ручной имитации работы автомата при лексическом анализе простого оператора присваивания языка, заданного на курсовую работу;
     + историю работы сканера, управляемого графом состояний и переходов, построенную путем ручной имитации работы автомата при лексическом анализе оператора присваивания (не точно такого же, какой использовался в предыдущем пункте);
     + описание лексики учебного языка, определенного заданием на курсовую работу, включающее все группы слов;
     + выводы и заключение;
     + приложение: тестовая программа на языке, заданном на курсовую работу, содержащая все элементы языка (функции, блоки операторов и все управляющие операторы: условные, операторы цикла, переключатели, операторы присваивания и все виды констант).
3. Контрольные вопросы.
   1. Что такое конечный автомат без памяти?
   2. Какие существуют способы задания конечных автоматов без памяти?
   3. Что такое первичное регулярное выражение?
   4. Как можно задать перечень и диапазон символов в регулярном выражении?
   5. Что такое квантификаторы, какие квантификаторы можно использовать в системах правил для пакета ВебТрансБилдер?
   6. Что такое эквивалентность конечных автоматов без памяти?
   7. Что такое тупиковые, недостижимые и эквивалентные состояния конечного автомата без памяти?
   8. Что такое оптимальность конечного автомата?
   9. Перечислите основные этапы процесса преобразования системы регулярных выражений в конечный автомат без памяти.
   10. Что такое недетерминированность конечного автомата без памяти? Перечислите виды недетерминированности.
   11. Что такое полностью и неполностью определенные конечные автоматы без памяти?
   12. Опишите существо процесса эпсилон-замыкания множества вершин графа состояний и переходов при ликвидации недетерминированностей.
   13. Какие способы преобразования графа состояний и переходов соответствуют операциям языка регулярных выражений?
   14. Как осуществляется поиск и удаление тупиковых состояний конечного автомата без памяти?
   15. Как осуществляется поиск и удаление недостижимых состояний конечного автомата без памяти?
   16. Как осуществляется поиск и слияние эквивалентных состояний конечного автомата без памяти?
   17. Как осуществляется преобразование не полностью определенного конечного автомата без памяти в полностью определенный?
   18. Что такое рабочая зона управляющей таблицы конечного автомата без памяти?
   19. Что такое история работы управляющей таблицы конечного автомата без памяти?
   20. Как в метаязыке регулярных выражений определяются символы, не имеющие графического изображения?

## Лабораторная/практическая работа № 3

1. Название работы: «Синтаксис языков программирования. Формальные грамматики».
2. Цели работы: изучение основных понятий метаязыка формальных грамматик, свойств грамматик и нетерминальных символов, рекурсивности и однозначности грамматик, недостижимости, бесплодности, аннулируемости и рекурсивности нетерминальных символов, отношений предшествования и последования между символами, приобретение навыков разработки формальных грамматик.
3. Основные теоретические сведения:

Формальные грамматики

Формальной грамматикой ***G*** называется совокупность

***G =* {*At*** , ***An*** ,*S*, ***P*}**,

состоящая:

* из алфавита терминальных символов ***At***;
* алфавита нетерминальных символов ***An***;
* начального нетерминального символа *S*;
* системы правил подстановки ***P***.

Алфавит терминальных символов ***At*** есть конечное множество всех слов языка, порождаемого данной грамматикой. Понятие «терминальный» в этом случае обозначает неразложимость, элементарность таких символов с точки зрения синтаксических правил.

Например, любой идентификатор при синтаксическом анализе считается простейшим символом, даже если он является цепочкой из нескольких литер. Более того, под одиночным терминальным символом, как правило, понимается вся группа слов, таких как идентификаторы или константы. В самом деле, с точки зрения синтаксиса (как совокупности правил образования предложений из слов) совершенно безразлично, какой именно идентификатор находится в левой части оператора присваивания или какие именно константы содержатся в выражении.

Наряду со словосочетанием «терминальный символ» будет использоваться слово «терминал».

В дальнейшем терминальные символы обозначаются либо одиночными литерами, представляющими собой отдельные слова в языках программирования (+, \*, =, ;, {, }, …), либо терминами, называющими слово или группу слов (идентификатор, константа, id, const, …), либо просто малыми буквами латинского алфавита, обозначающими произвольный терминал (a, b, c, …). Например:

***At***={идентификатор, константа, *+*, *–*, *\**, /, =}

Алфавит нетерминальных символов ***An*** есть конечное множество названий синтаксических конструкций, таких как <предложение>, <выражение>, <список аргументов>, <условный оператор>, <тело функции>. Нетерминальные символы используются только в метаязыке, на котором описывается язык программирования, никакой нетерминальный символ не может появиться в тексте правильной программы. Наряду со словосочетанием «нетерминальный символ» в дальнейшем будет использоваться просто «нетерминал». Нетерминальные символы принято обозначать либо словосочетаниями в угловых скобках, как показано в начале этого абзаца, либо словами, начинающимися с прописной буквы и содержащими буквы, цифры и символы подчеркивания (например, Function, ArgList, Const\_16), либо просто большими буквами латинского алфавита в курсивном начертании: *A*, *B*, *C*, ..., *Z.*

Начальный нетерминальный символ *S* является одним из нетерминальных символов. Обычно так обозначается наиболее общая синтаксическая конструкция, например <правильная программа>.

Символы (как терминальные, так и нетерминальные) могут образовывать цепочки, которые будут обозначаться малыми буквами греческого алфавита α, β, γ, ..., ω. Особое значение будет играть пустая цепочка символов ε.

Система правил подстановки ***P*** (иногда называемая системой порождающих правил, или продукций) есть конечное множество пар цепочек вида α : β, причем цепочка α (левая часть правила) должна содержать хотя бы один нетерминальный символ.

Каждая такая пара цепочек называется правилом подстановки (порождающим правилом, продукцией) и определяет возможный способ замены левой части правила на его правую часть. Правила подстановки обычно нумеруются, причем в левой части правила с номером 1 должен находиться одиночный начальный нетерминал грамматики.

В качестве примера приведем фрагмент грамматики С-подобного языка (пока будем считать этот фрагмент полноценной грамматикой ***G*1**):

*S : S + T*

*S : S – T*

*S : T*

*T : ident*

*T : const*

Лексические правила для терминалов *ident* и *const* пока не будем считать частью этой грамматики. Эта грамматика порождает (определяет как правильные) арифметические выражения, содержащие только идентификаторы, константы и знаки операций сложения и вычитания.

Дерево грамматического разбора

Пусть дана грамматика ***G*** с системой порождающих правил ***P***.

Цепочка ψ непосредственно выводится из цепочки σ (σ → ψ), если:

* σ = ω α δ, где ω и δ – произвольные, возможно пустые цепочки;
* ψ = ω β δ, где ω и δ – те же самые цепочки;
* в системе порождающих правил ***P*** есть правило α : β.

Подстановка цепочки β вместо α в цепочке σ = ω α δ порождает ω β δ, т. е. цепочку ψ. Обратная подстановка не подразумевается, т. е. из возможности непосредственного вывода σ → ψ не следует возможность непосредственного вывода ψ → σ.

Например, в грамматике ***G*1**цепочка *S + const* непосредственно выводится из цепочки *S + T* путем подстановки правой части правила *T : const* вместо *T*.

Цепочка ψ выводится из цепочки σ обозначается (σ ⇒ ψ), если существует последовательность непосредственных выводов

σ → ψ1 → ψ2 → ψ3 ... ψn → ψ

Например, в грамматике ***G*1**цепочка *x* – 1 выводится из начального нетерминала *S* путем выполнения такой последовательности непосредственных выводов:

*S* → *S – T* → *T – T* → *ident – T* → *x – T* → *x – const* → *x – 1*

Правильным предложением языка ***L***, определяемого грамматикой ***G***, называется цепочка, состоящая ***только*** из терминальных символов и ***выводимая*** из начального нетерминала *S*. Любая цепочка терминальных символов, для которой не существует вывода из начального нетерминала *S* грамматики ***G***, является неправильным предложением языка ***L***. Цепочка, содержащая символы, не входящие в алфавит терминальных символов ***At*** (в частности, нетерминальные символы), вообще не является предложением языка ***L***.

Таким образом, грамматика ***G*** разбивает бесконечное множество всех возможных цепочек, содержащих только терминальные символы (слова языка), на два непересекающихся подмножества:

* подмножество правильных предложений, которое собственно и является языком ***L***, порождаемым грамматикой ***G***;
* подмножество неправильных предложений.

Вывод правильного предложения можно представить в графической форме, сопоставив символам цепочек узлы, а применению правил – дуги графа. Такая форма обычно называется деревом грамматического разбора.

Основная задача синтаксического акцепта – проверка правильности последовательности слов транслируемой программы – формулируется так: предложение языка (т. е. последовательность терминалов порождающей язык грамматики) является правильным, если может быть установлен факт существования дерева его грамматического разбора, и неправильным в противном случае.

Само дерево разбора при этом можно и не строить, достаточно только выяснить, возможно ли это. Сложность задачи восстановления дерева разбора предложений языка (или выяснения факта его существования) существенно зависит от свойств грамматики, определяющей этот язык.

Любая грамматика ***G*** определяет единственный язык ***L***. Однако существуют языки, для определения которых можно построить более чем одну грамматику (различающимися считаются грамматики, для которых деревья грамматического разбора хотя бы одного правильного предложения данного языка имеют разную структуру). Таков, например, язык скобочных арифметических выражений со знаками операций различного приоритета, две разные грамматики которого показаны в табл. 4. Эти две грамматики различаются не только количеством правил и обозначениями нетерминалов.

Таблица 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Грамматика ***G a*1** | | | Грамматика ***G a*2** | |
| 1 | *S* : *S + T* | | 1 | *S* : *U R* |
| 2 | *S* : *T* | | 2 | *R* : *+ S* |
| 3 | *T* : *T \* V* | | 3 | *R* : |
| 4 | *T* : *V* | | 4 | *U* : *V W* |
| 5 | *V* : *( S )* | | 5 | *W* : *\* U* |
| 6 | | *V* : *ident* | 6 | *W* : |
| 7 | | *V* : *const* | 7 | *V* : *( S )* |
|  | |  | 8 | *V* : *ident* |
|  | |  | 9 | *V* : *const* |

Пусть дано предложение *(a + b) \* с*, где *a*, *b* и *с* – идентификаторы или константы. Построим выводы этого предложения из начальных нетерминалов грамматик ***Ga*1** и ***Ga*2**, заменяя на каждом шаге самый левый нетерминал на правую часть одного из правил для этого нетерминала.

Для ***Ga*1**:

*S → T → T\*V → V\*V → (S)\*V → (S+T)\*V → (T+T)\*V → (V+T)\*V →   
(a+T)\*V → (a+V)\*V → (a+b)\*V → (a+b)\*c*

Для ***Ga*2**:

*S → UR → VWR → (S)WR → (UR)WR → (VWR)WR → (aWR)WR → (aR)WR → (a+S)WR → (a+UR)WR → (a+VWR)WR → (a+bWR)WR → (a+bR)WR → (a+b)WR → (a+b)\*UR → (a+b)\*VWR → (a+b)\*cWR → (a+b)\*cR → (a+b)\*c*

Даже не преобразуя эти два вывода в графически представленные деревья грамматического разбора, легко можно заметить, что эти деревья существенно отличаются друг от друга. Таким образом, действительно существуют эквивалентные, но различающиеся по своим свойствам грамматики, определяющие один и тот же язык.

Свойства грамматик оказывают большое влияние как на принципиальную возможность использования данной грамматики для синтаксического анализа, так и на применимость средств автоматизации построения синтаксических анализаторов.

Все грамматики можно разделить на четыре основных класса по Хомскому, впервые сформулировавшему приведенные ниже признаки деления на иерархически вложенную систему классов: общие, контекстно-зависимые, контекстно-свободные и регулярные.

Для целей построения синтаксических анализаторов трансляторов наилучшим образом подходят контекстно-свободные грамматики, у которых левая часть каждого правила представляет собой в точности один нетерминал, а на правые части не накладывается никаких ограничений. Они позволяют определять синтаксис формальных языков, в том числе языков программирования, т. е. совокупность правил построения предложений языка из его слов.

Независимость от контекста означает возможность подстановки правой части любого из правил для замены нетерминала из левой части правила в любой цепочке, содержащей этот нетерминал.

Свойства контекстно-свободных грамматик

Свойства любой грамматики полностью определяются ее системой порождающих правил и могут служить основанием для разбиения всего класса контекстно-свободных грамматик на более узкие подклассы. Для некоторых подклассов контекстно-свободных грамматик известны эффективные алгоритмы автоматизации проектирования трансляторов для порождаемых ими языков.

Наиболее важными свойствами грамматик в целом являются рекурсивность и однозначность.

***Рекурсивность.*** Нетерминальный символ *X* называется рекурсивным, если из него могут быть выведены цепочки, содержащие сам этот символ *X*, т. е.

*X* ⇒ μ *X* η,

где μ *X* η – произвольные цепочки символов.

Существует много вариантов свойства рекурсивности нетерминалов: прямая, косвенная, левая, правая и общая. Эти варианты могут сочетаться произвольным образом. Грамматика называется рекурсивной, если рекурсивен хотя бы один нетерминальный символ, и нерекурсивной в противном случае.

Нерекурсивные грамматики с конечным количеством порождающих правил определяют так называемые конечные языки, в которых количество правильных предложений ограниченно. Такие языки и грамматики не представляют практического интереса.

***Однозначность.*** Грамматика называется однозначной, если любое правильное предложение порождаемого ею языка имеет единственное дерево грамматического разбора, и неоднозначной в противном случае. Приведенные выше грамматики ***Ga*1**и ***Ga*2** являются однозначными, но для того же самого языка можно привести пример неоднозначной грамматики, например, такой, как показано в табл. 5.

Выявление свойства однозначности некоторой грамматики является алгоритмически неразрешимой проблемой. Не существует универсального алгоритма, способного для любой заданной контекстно-свободной грамматики определить, однозначна ли она. Известно, что методы построения синтаксических анализаторов путем преобразования формальной грамматики в конечный автомат не приводят к желаемому результату для неоднозначных грамматик. Другими словами, неоднозначные грамматики непригодны для преобразования в конечный автомат (со стековой памятью), выполняющий функции синтаксического анализатора. Однако и многие однозначные грамматики также не допускают подобного преобразования, но по другим причинам.

Таблица 5

|  |  |
| --- | --- |
| Грамматика ***Ga*3** | |
| 1 | *S* : *S + S* |
| 2 | *S* : *T* |
| 3 | *T* : *T \* T* |
| 4 | *T* : *V* |
| 5 | *V* : *( S )* |
| 6 | *V* : *ident* |
| 7 | *V* : *const* |

Кроме общих свойств грамматик для решения задач синтаксического анализа важны некоторые свойства нетерминальных символов и отношения между символами грамматики.

Свойства символов грамматики

***Аннулируемость.*** Нетерминальный символ называется аннулируемым, если из него может быть выведена пустая цепочка символов. В противном случае нетерминал является неаннулируемым. В литературе по формальным языкам и грамматикам такие символы обычно называются аннулирующими и неаннулирующими.

Это свойство нетерминальных символов может иметь важное значение для свойств грамматики в целом и ее применимости в том или ином методе синтаксического анализа.

***Недостижимость.*** Символ (терминальный или нетерминальный) называется недостижимым, если он не появляется ни в одной цепочке символов, выводимой из начального нетерминала грамматики.

***Бесплодность.*** Нетерминальный символ называется бесплодным, если из него не может быть выведена цепочка, состоящая только из терминалов.

Недостижимые и бесплодные нетерминалы могут появиться в грамматике либо в результате ошибок при разработке ее системы порождающих правил, либо в результате эквивалентных преобразований грамматики с целью изменения ее свойств или свойств ее символов. Все правила, содержащие такие символы, следует удалить из грамматики (а сами символы – из ее алфавитов). Алгоритмы вычисления свойств символов грамматики следует изучить по [1].

Кроме свойств отдельных символов большое значение для целей синтаксического анализа имеют некоторые отношения между символами, определяемые всей совокупностью правил грамматики.

Важнейшими отношениями являются отношение предшествования и отношение следования, они вычисляются в виде множеств, сопоставляемых с символами грамматики и содержащих опять-таки символы этой грамматики.

Множества предшественников символов грамматики

Предшественником некоторого символа *X* называется символ, с которого начинается цепочка, выводимая из *X*. Считается, что любой символ является предшественником самого себя (т. е. допускается вывод длины 0).

Термин «предшественник» в русском языке имеет несколько другой смысл: нечто, появляющееся **до** того, чему оно предшествует. Тем не менее в литературе по формальным языкам и грамматикам (и в настоящем учебном пособии) этот термин используется именно в том смысле, который определен в предыдущем абзаце.

Множества предшественников цепочек символов

При построении синтаксических анализаторов часто приходится оперировать не отдельными символами, а цепочками символов.

В множество предшественников цепочки β входят те и только те терминальные символы, с которых начинаются цепочки, выводимые из β.

Если известны множества предшественников одиночных символов грамматики, то можно легко сформулировать способ определения множеств предшественников произвольных цепочек символов.

Пусть цепочка имеет вид β = *ABCD*..., где *A*, *B*, *C*, *D*, ... – произвольные (не обязательно нетерминальные) символы грамматики. Пусть также известны множества предшественников всех символов грамматики.

Тогда множество предшественников цепочки β можно вычислить по следующей формуле:



Здесь  – пустое множество; под неаннулируемым символом понимается либо терминал, либо неаннулируемый нетерминал.

Множества последователей символов грамматики

Символ *Y* называется последователем символа *X*, если хотя бы в одной цепочке ω, выводимой из начального нетерминала грамматики, символ *Y* непосредственно следует за *X*:

*S* ⇒ ω = ... *X Y* ...

Это определение понятия последования не является конструктивным. Для того чтобы можно было понять, что такое последователи, и построить процедуру определения множеств последователей для символов грамматики, нужно точно сформулировать условия возникновения отношения последования.

Символ *Y* является непосредственным последователем символа *X*, если выполняется либо условие 1, либо условие 2.

Условие 1. В грамматике есть правило вида *N* : ϕ *X Y* ψ , где ϕ и ψ – произвольные, возможно пустые цепочки.

Условие 2. В грамматике есть правило вида *N* : ϕ *X* σ *Y* ψ, где ϕ и ψ – произвольные, возможно пустые цепочки; σ – цепочка, состоящая только из аннулируемых нетерминалов.

Непосредственными последователями, к сожалению, полные множества последователей символов грамматики не исчерпываются.

Символ *Y* является последователем символа *X*, если *Y* является последователем (не важно, непосредственным или нет) некоторого символа *Z* и выполняется условие 3.

Условие 3. В грамматике есть правило вида *Z* : χ *X* σ, где χ – произвольная, возможно пустая цепочка; σ – цепочка, либо состоящая только из аннулируемых нетерминалов, либо пустая.

Наконец, символ *Y* является последователем символа *X*, если *Y* есть последователь некоторого символа *Z* (не важно, непосредственный или нет)и выполняется условие 4.

Условие 4. В грамматике есть совокупность правил следующего вида:

|  |  |  |
| --- | --- | --- |
| *Z* | : | χ1 *Z*1σ1 |
| *Z*1 | : | χ2 *Z*2σ2 |
| *...* |  |  |
| *Zn–*1 | : | χ*n Zn* σ*n* |
| *Zn* | : | χ0 *X* σ0 |

Здесь χ0, χ1, ..., χ*n* – произвольные цепочки, а σ0, σ1, ..., σ*n* – цепочки, состоящие только из аннулируемых нетерминалов или пустые.

Условия 1 и 2 определяют отношение «символ *Y* есть непосредственный последователь символа *X*», а условия 3 и 4 – отношение «символ *X* есть последний (замыкающий) символ в цепочке, выводимой из символа *Z*». Искомое отношение «символ *Y* есть последователь символа *X*» является произведением этих двух отношений.

Хотя для решения задачи синтаксического анализа интерес представляют только множества последователей нетерминальных символов, для их вычисления приходится определять множества последователей всех (терминальных и нетерминальных) символов грамматики.

Множества предшественников и последователей символов грамматики будут самым существенным образом использоваться в процессе преобразования формальных грамматик в синтаксические акцепторы, т. е. при выполнении всех остальных работ.

1. Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample3):
   1. Разработать описание синтаксиса языка, заданного на курсовую работу. Описание синтаксиса должно включать в себя следующее:
      * структуру файла программы (состав модулей / секций файла);
      * структуру и назначение каждой секции / модуля;
      * формат и точный способ выполнения всех операторов (присваивания, цикла и др.) в виде последовательности операций, сложность которых не выше сложения / умножения / сравнения /… двух значений, а также операций условного или безусловного перехода.
   2. Используя систему правил Samples / Sample3 как основу, выполнить следующее:
      * освоить и изучить ввод и редактирование синтаксических правил, содержащих слова-строки, группирование, операции выбора и квантификаторы;
      * проанализировать соответствие фактических правил видимым / редактируемым правилам в процессе редактирования системы правил; понять, как выявляется начальный нетерминал грамматики;
      * разобраться в причинах возникновения недоступных и бесплодных нетерминалов, отображаемых серым цветом и отсутствующих в системе фактических правил после завершения ввода правил, которые содержат такие символы;
      * разобраться в том, каким образом пакет «ВебТрансБилдер» разделяет правила на лексические и синтаксические; описать принципы этого разделения.
   3. Изучить по [2–5] теоретические определения и алгоритмы вычисления отношений предшествования и последования для символов грамматики.
   4. Освоить просмотр свойств грамматик и их символов в пакете «ВебТрансБилдер». Необходимо достичь понимания того, почему те или иные символы грамматики имеют свой конкретный набор свойств (пункты меню «Показать / Правила грамматики», «Показать / Множества предшественников» и «Показать / Множества последователей»).
   5. Используя полученные навыки работы с грамматиками и программным обеспечением, начать поэтапную разработку грамматики языка для заданного варианта курсовой работы. Реализовать правила, определяющие как минимум выражения со всеми знаками операций языка, заданного на курсовую работу, оператор присваивания и не менее одного управляющего оператора.
   6. Выполнить построение транслятора по разработанной грамматике и шаблону табличного восходящего анализатора для языка JavaScript. Проверить работоспособность построенного транслятора на фрагментах тестовых программ, написанных на заданном языке; добиться того, чтобы выражения и выполняемые операторы воспринимались транслятором как правильные.
2. Подготовить отчет, который должен содержать следующие разделы:
   * + цель работы;
     + описание синтаксиса языка как результат выполнения п. 1 практической части (и как раздел расчетно-пояснительной записки к курсовой работе);
     + результаты разработки и тестирования грамматики (п. 5 и 6 практической части) для языка, заданного на курсовую работу;
     + матричное представление отношений предшествования и последования для символов этой грамматики, сформированное пакетом «ВебТрансБилдер»; краткое описание этих отношений;
     + результаты анализа алгоритмов и способов преобразования видимой системы правил в фактическую, выявления или формирования начального нетерминала, разделения правил на лексические и синтаксические, выявления недостижимых и бесплодных нетерминалов. Этот анализ должен быть проведен для результата выполнения п. 6 практической части;
     + выводы и заключение.
3. Контрольные вопросы.
   1. Что такое терминальные и нетерминальные символы формальных грамматик?
   2. Что такое рекурсия? Перечислите виды рекурсии.
   3. Что такое контекстно-свободные грамматики?
   4. Вычислите множество предшественников цепочки символов   
      *W R* для грамматики ***Ga2***.
   5. Является ли однозначной грамматика:
   6. S : S "|" S
   7. S : S S
   8. S : "(" S ")"
   9. S : ident
   10. Что такое аннулируемый нетерминал?
   11. Что такое стековая память? Какие операции обычно определяются над стековой памятью?
   12. Что такое множество последователей символа грамматики?
   13. Чем понятие вывода отличается от понятия непосредственного вывода?
   14. Что такое недетерминированность конечного автомата без памяти? Какие бывают виды недетерминированности?
   15. Устраните левую рекурсию в такой грамматике:
4. *S* : *"(" L ")"*
5. *S* : *ident*
6. *L* : *L "," S*
7. *L* : *S*
   1. Что такое дерево грамматического разбора?
   2. Какие нетерминальные символы называются бесплодными?
   3. Что такое рекурсивный цикл в грамматике?
   4. Какие грамматики называются эквивалентными?
   5. Могут ли быть эквивалентными два конечных автомата без памяти, имеющие разное количество финальных состояний?
   6. Вычислите множества последователей для грамматики из вопроса 6.5.
   7. Что такое контекстно-зависимая грамматика?
   8. Что такое отношение предшествования символов?
   9. Можно ли удалить из грамматики пряморекурсивный нетерминал?
   10. Постройте дерево разбора предложения (ident, (ident, ident)) в грамматике из вопроса 6.11.
   11. Что такое факторизация?
   12. Сформулируйте физический смысл перехода по пустой цепочке в финальное состояние конечного автомата без памяти.
   13. Можно ли и, если можно, то как преобразовать косвенную рекурсию в прямую?
   14. Определите алфавиты терминальных и нетерминальных символов для грамматики из вопроса 6.11.
   15. Какие конечные автоматы без памяти называются эквивалентными?
   16. Перечислите условия, которые определяют отношение последования символов грамматики.
   17. Вычислите вручную множества предшественников и последователей и все свойства символов для фрагмента грамматики, определяющий синтаксис условного оператора языка С:
8. *Operator : …*
9. *Operator : "if" "(" LogicalExpression ")" Operator Rest*
10. *Rest : "else" Operator*
11. *Rest :*
    1. Является ли грамматика вопроса 6.29 однозначной?
    2. Как соотносятся между собой языки и грамматики?
    3. Могут ли быть эквивалентными два конечных автомата без памяти, имеющие различное количество рабочих состояний?
    4. Как вычисляется множество предшественников цепочки символов?
    5. Почему из системы порождающих правил грамматики не может быть удален ее начальный нетерминал?
    6. Как связаны между собой понятия вывода и дерева грамматического разбора?

## Лабораторная/практическая работа № 4

1. Название работы: «Синтаксис языков программирования. Нисходящий синтаксический анализ.
2. Цели работы: изучение основных идей и понятий нисходящих методов синтаксического анализа, выявление свойств формальных грамматик, необходимых для реализации нисходящего восстановления дерева грамматического разбора, приобретение навыков построения процедурной и различных автоматных реализаций нисходящего анализа, исследование поведения нисходящих синтаксических акцепторов.
3. Основные теоретические сведения:

Нисходящие методы синтаксического анализа

Нисходящими называются такие методы синтаксического анализа, при которых восстановление дерева грамматического разбора выполняется сверху от начального нетерминала контекстно-свободной грамматики вниз к анализируемому предложению (входной цепочке терминалов).

Каждый шаг процесса восстановления дерева состоит в применении одного непосредственного вывода, т. е. в замене единственного нетерминального символа правой частью какого-либо правила для этого нетерминала. Главное прагматическое требование к этому процессу – необходимость организации безоткатного однонаправленного движения вниз по дереву.

Следовательно, выбор правила на каждом шаге должен осуществляться таким образом, чтобы гарантировать:

* + - 1. восстановление дерева для любого правильного предложения;
      2. обнаружение невозможности восстановить дерево для любого неправильного предложения.

Оказывается, что дать такие гарантии можно отнюдь не для любой грамматики и, более того, не для любого языка.

Существуют языки, для которых никакая порождающая грамматика не позволяет организовать безвозвратное нисходящее восстановление дерева грамматического разбора (в чистом виде, т. е. без применения специальных мер, модифицирующих свойства грамматики). Существуют и такие языки, для которых одни порождающие грамматики пригодны для детерминированного нисходящего восстановления дерева, а другие нет. Языки программирования обычно относятся именно к этой группе.

Основная идея группы нисходящих методов восстановления дерева грамматического разбора

Эта идея очень проста (при условии, что грамматика относится к так называемому классу **LL(1)**) и состоит в применении следующего алгоритма.

* + - * 1. Создается корень дерева (он же – самый верхний уровень дерева) в виде узла, содержащего начальный нетерминал грамматики. Из анализируемого предложения считывается (считывание слова – это вызов лексического анализатора) первое слово, которое становится текущим терминалом. Далее нисходящий синтаксический анализатор постоянно циклически работает только с двумя символами: один берется из текущего узла нижнего уровня восстанавливаемого дерева, другим является текущий терминал. В каждом повторении цикла реализуется либо шаг А2, либо шаг А3.
        2. Если текущий узел нижнего уровня дерева содержит нетерминал, то просматриваются те и только те правила грамматики, которые содержат этот нетерминал в левой части. Возможны по меньшей мере три разных результата этого просмотра.
  1. Нет ни одного правила, из правой части которого можно вывести остаток предложения, начинающийся с текущего терминала. Это означает, что входная цепочка символов в целом не может быть выведена из начального нетерминала грамматики, т. е. не является правильным предложением языка, порождаемого грамматикой. В этом случае процесс синтаксического анализа нужно остановить по ошибке.
  2. Если из правой части просматриваемого правила можно вывести цепочку символов, начинающуюся с текущего терминала, то это правило можно применить для замены узла последовательностью узлов, формируемых из цепочки символов правой части. Если такое правило в точности одно, то текущий уровень дерева разбора преобразуется в следующий путем замены текущего узла на цепочку узлов, формируемых из символов правой части этого правила. Текущим узлом становится либо первый узел этой цепочки, если правая часть примененного правила не пуста, либо узел, следующий за обработанным текущим узлом, если правая часть правила пуста. Текущий терминал не изменяется, выполняется возврат к шагу А2.
  3. Если найдено несколько правил, из правой части каждого из которых можно вывести цепочку символов, начинающуюся с текущего терминала из предложения, то обнаружена невозможность обеспечить безоткатное однонаправленное движение сверху вниз для восстановления дерева. Процесс синтаксического анализа нужно остановить. Эта грамматика не годится для нисходящего синтаксического анализа по следующим причинам:

– во-первых, существенно усложняется процесс формирования дерева и необходимые для этого структуры данных из-за необходимости обеспечивать возвраты для перебора пригодных правил;

– во-вторых, даже если усложнить алгоритм и реализовать возвраты, то резко, до неприемлемых на практике значений порядка *N* в кубе, возрастает количество циклов алгоритма для восстановления дерева (здесь *N* – количество слов в анализируемой программе).

* + - * 1. Если текущий узел нижнего уровня дерева содержит терминал, то возможны только три следующих варианта.

А3.1. Этот терминал не совпадает с текущим терминалом из предложения. В таком случае входная цепочка символов в целом не может быть выведена из начального нетерминала грамматики, т. е. не является правильным предложением языка, порождаемого грамматикой. Процесс синтаксического анализа нужно остановить по ошибке.

А3.2. Этот терминал совпадает с текущим терминалом из предложения и не является терминалом ► (конец файла). Нужно перейти к следующему узлу в нижнем уровне дерева (сделать следующий текущим), прочитать следующий терминал из предложения путем вызова лексического анализатора и установить полученное от него слово в качестве нового текущего терминала. Затем должен быть выполнен возврат к шагу А2 для продолжения восстановления дерева.

А3.3. Этот терминал совпадает с текущим терминалом из предложения и является терминалом ► (конец файла). Восстановление дерева завершено, анализируемое предложение является правильным.

Пригодность грамматики для реализации нисходящего разбора. LL(1)-грамматики

Из описания существа процесса нисходящего синтаксического анализа следует, что грамматика должна быть предварительно проанализирована на принадлежность к так называемому классу **LL(1)** для исключения возможности возникновения ситуаций вида А2.3. Одновременно нужно определить способ проверки того, может ли из правой части правила быть выведена цепочка терминалов, которая начинается с имеющегося на шаге А2 текущего терминала.

Свяжем с каждым правилом грамматики множество терминалов, называемое множеством выбора и вычисляемое следующим образом.

Пусть есть правило *N* : α*j*.

В зависимости от того, какова цепочка символов α*j*, составляющая правую часть правила, множество его выбора вычисляется так:

1. *M*выб (*N* : α*j*) = *M*пр (α*j*), если α*j* содержит хотя бы один терминал или неаннулируемый нетерминал;
2. *M*выб (*N* : α*j*) = *M*посл (*N*), если α*j* есть пустая цепочка;
3. *M*выб (*N* : α*j*) = *M* пр (α*j*) **∪** *M*посл (*N*), если α*j* не пуста, но состоит только из аннулируемых нетерминалов.

Здесь *M* пр (α*j*) – множество предшественников цепочки α*j*; *M*посл (*N*) – множество последователей нетерминала *N*.

**LL(1)**-грамматикой называется такая контекстно-свободная грамматика, у которой множества выбора правил с одинаковым нетерминалом в левой части попарно не пересекаются.

Любая такая грамматика может быть использована для организации нисходящего детерминированного восстановления дерева грамматического разбора предложений порождаемого ею языка. Другими словами, на основе любой **LL(1)**-грамматики может быть построен детерминированный нисходящий синтаксический акцептор, проверяющий правильность предложений языка.

Вычислим множества выбора для правил известных нам грамматик ***Ga*1** и ***Ga*2**, используя свойства символов и множества предшественников и последователей, изучавшиеся в предыдущей работе, и проверим, относятся ли эти грамматики к классу **LL(1)**.

Вначале рассмотрим свойства грамматики ***Ga*2**(табл. 6, правая часть). Заметим, что в ее системе порождающих правил для каждого нетерминала либо есть единственное правило, либо множества выбора нескольких правил, содержащих один и тот же нетерминал в левой части, не пересекаются (правила 2 и 3 для *R*, правила 5 и 6 для *W*, правила 7, 8 и 9 для *V*). Поэтому при восстановлении дерева разбора в любом цикле на шаге А2 по любому текущему входному символу можно выбрать только одно правило для замены нетерминала – то, множество выбора которого содержит текущий входной терминал.

Таблица 6

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Грамматика ***G a*1** | | | | Грамматика ***G a*2** | | | |
| № | Правило | Способ | Множество | № | Правило | Способ | Множество |
| 0 | *Z* : *S* ► | *M* пр(*S* ► ) | *(, ident, const* | 0 | *Z* : *S* ► | *M* пр(*S* ►) | *(, ident, const* |
| 1 | *S* : *S + T* | *M* пр(*S + T*) | *(, ident, const* | 1 | *S* : *U R* | *M* пр(*U R*) | *(, ident, const* |
| 2 | *S* : *T* | *M* пр(*T*) | *(, ident, const* | 2 | *R* : *+ S* | *M* пр(+ *S*) | + |
| 3 | *T* : *T \* V* | *M* пр(*T \* V*) | *(, ident, const* | 3 | *R* : | *M* посл (*R*) | *(,* ► |
| 4 | *T* : *V* | *M* пр(*V*) | *(, ident, const* | 4 | *U* : *V W* | *M* пр(*V W*) | *(, ident, const* |
| 5 | *V* : *( S )* | *M* пр(*( S )*) | *(* | 5 | *W* : *\* U* | *M* пр(+ *S*) | \* |
| 6 | *V* : *ident* | *M* пр(*i*) | *ident* | 6 | *W* : | *M* посл (*W*) | *+, (,* ► |
| 7 | *V* : *const* | *M* пр(*c*) | *const* | 7 | *V* : *( S )* | *M* пр( *(S)* ) | *(* |
|  |  |  |  | 8 | *V* : *ident* | *M* пр(*ident*) | *ident* |
|  |  |  |  | 9 | *V* : *const* | *M* пр(*const*) | *const* |

Например, если текущий узел нижнего уровня восстанавливаемого дерева содержит нетерминал *W*, а текущим терминалом из входной цепочки является знак операции сложения + (или круглая закрывающая скобка *)*, или конец файла ►), то заменять узел *W* следует на правую часть правила 6, т. е. на пустую цепочку. Если же текущий терминал из входной цепочки есть знак операции умножения \*, то вместо узла с *W* необходимо подставить цепочку из двух узлов *\** и *U*, т. е. правую часть правила 5. Если же текущий терминал – это идентификатор *ident*, константа *const* или открывающая скобка *(*, то никакое правило не может быть использовано для замены нетерминала *W*. Это свидетельствует о том, что входная цепочка не может быть выведена из начального нетерминала этой грамматики и, следовательно, не является правильным предложением. Например, при попытке восстановления дерева разбора цепочки *( x+ y ) z* ► возникнет именно эта ситуация. Очередной уровень дерева разбора, выведенный из нетерминала *S*, будет выглядеть так:

*S* ⇒ *( x+ y ) W R*

Текущий узел выделен рамкой. Текущим терминалом является идентификатор *z*. Поскольку идентификатор не входит ни в одно множество выбора правил, имеющих нетерминал *W* в левой части, то не существует способа вывода остатка входной цепочки, имеющего вид *z* ►, из цепочки узлов *W R*, а следовательно, и дерева грамматического разбора всей входной цепочки *( x + y ) z* ► из начального нетерминала *S*.

Грамматика ***Ga*2** принадлежит к классу **LL(1)**-грамматик.

Обратимся теперь к грамматике ***Ga*1**(табл. 6, левая часть). Множества выбора правил 1 и 2 для нетерминала *S* одинаковы (а следовательно, пересекаются). Одинаковы также множества выбора правил 3 и 4 для нетерминала *T*. Поэтому попытка восстановления дерева разбора правильной цепочки (например, *( x + y ) \* z* ►) может завести нас в тупик вследствие неверного принятия решения уже на первом шаге, если вместо подстановки по правилу 2(*S* → *T*) будет выбрана подстановка по правилу 1 (*S* → *S + T*).

И та и другая подстановка в данный момент формально применимы, поскольку множества выбора и правила 1, и правила 2 содержат терминал *(*. Если будет выбрано правило 1, то впоследствии никаким способом уже нельзя будет вывести цепочку *( x + y ) \* z* ► из цепочки *S + T*.

Возможность принятия неверного решения при использовании грамматики ***Ga*1** возникает каждый раз, когда на текущем уровне восстанавливаемого дерева самым левым оказывается либо нетерминал *S*, либо нетерминал *T*.

Грамматика ***Ga*1** не относится к классу **LL(1)**-грамматик. Это является следствием леворекурсивности нетерминалов *S* и *T*. Любая левая рекурсия   
(в том числе косвенная) влечет за собой пересечение множеств выбора правил, имеющих в левой части леворекурсивный нетерминал. Поясним это на простом примере прямой левой рекурсии. Пусть в системе порождающих правил грамматики есть такие правила для нетерминала *N*:

*N* : *N* β

*N* : γ.

Заметим, что первое правило – причина левой рекурсии, а из правой части второго правила должна быть выводима цепочка терминалов (если это не так или второго правила просто нет, то нетерминал *N* бесплоден и должен быть удален из грамматики со всеми своими правилами).

Каково бы ни было множество выбора второго правила, оно целиком содержится и в множестве выбора первого правила, поскольку

*M*выб (*N* β) = *M*пр (*N* β),

а множество предшественников цепочки *N* β по определению включает в себя множество предшественников нетерминала *N*, которое, в свою очередь, содержит множество предшественников цепочки γ. Следовательно, множества выбора этих правил пересекаются и оба содержат множество предшественников цепочки символов γ.

Итак, любая леворекурсивная грамматика не принадлежит классу  
**LL(1)**-грамматик. Левую рекурсию всегда можно преобразовать в правую или общую. Однако это преобразование не гарантирует перехода грамматики в класс **LL(1)**, потому что не только свойство левой рекурсии может быть причиной непригодности грамматики для построения нисходящего синтаксического акцептора. В общем случае задача нахождения **LL(1)**-грамматики, эквивалентной заданной грамматике, алгоритмически неразрешима.

Алгоритм нисходящего восстановления дерева грамматического разбора, сформулированный выше, в принципе может быть использован с любой **LL(1)**-грамматикой, но применение его на практике потребует уточнения ряда деталей, в том числе способа поиска правила, способа представления и хранения узлов дерева и др. Такая детализация может привести к радикальному изменению внешнего вида алгоритма при сохранении его сути.

Реализация общего алгоритма для конкретной грамматики обычно сводится к построению специального алгоритма, определяемого совокупностью порождающих правил, или к преобразованию грамматики в управляющую таблицу конечного автомата. Методы построения специальных алгоритмов или управляющих таблиц по грамматике легко формализуются. Следовательно, если заданная грамматика принадлежит классу **LL(1)**-грамматик, то построение нисходящего синтаксического акцептора предложений порождаемого ею языка может быть автоматизировано.

Существует несколько вариантов реализации общего алгоритма и методов соответствующего преобразования грамматики. Пакет «ВебТрансБилдер» для каждого инструментального языка содержит набор шаблонов преобразования грамматики в код программы транслятора. Для построения нисходящих синтаксических анализаторов (парсеров) существуют шаблоны, формирующие следующие их виды:

* парсер как нисходящий стековый автомат с одним состоянием;
* парсер как нисходящий стековый автомат с несколькими состояниями;
* парсер как совокупность функций, рекурсивно вызывающих друг друга.

Построение парсера как нисходящего стекового автомата   
с одним состоянием

Поведение нисходящего стекового автомата с одним состоянием определяется управляющей таблицей, столбцы которой соответствуют входным символам, строки – символам, которые могут находиться в стеке, а в клетках содержится последовательность операций над стеком, входной цепочкой символов и состоянием автомата.

Обычные для стековой памяти операции будут обозначаться так:

! *X* – занесение символа (или цепочки символов) *X* в стек (аналог операции push), при этом первый символ цепочки окажется в стеке под всеми остальными, а последний символ окажется на самом верху стека;

^ – снятие одного символа с вершины стека (аналог операции pop). Заметим, что попытка выполнения этой операции при пустом стеке должна приводить к останову автомата по обнаружению ошибки во входной цепочке.

Над входным потоком определена единственная операция, которую будем обозначать так:

> – чтение следующего символа из входной цепочки.

Для управления состоянием автомата используется единственный знак операции Stop, предназначенный для останова по успешному окончанию восстановления дерева разбора. Для обозначения операции останова по обнаружению ошибок во входной цепочке используется обычное соглашение: соответствующая клетка управляющей таблицы пуста.

Далее описана процедура преобразования системы порождающих правил грамматики в управляющую таблицу автомата, реализованная в шаблонах пакета.

**Шаг 1.** Построить заготовку таблицы, имеющую ровно столько столбцов, сколько символов есть в терминальном алфавите грамматики (включая псевдотерминал ►), и столько строк, сколько символов есть в нетерминальном алфавите (ниже будет показано, что в процессе преобразования возможно появление дополнительных строк). Озаглавить столбцы терминалами грамматики (порядок следования столбцов не имеет значения), строки – нетерминалами (тоже в произвольном порядке).

**Шаг 2.** В силу того, что автомат предназначен для разбора цепочки, выводимой из правой части специального добавочного правила грамматики *Z* : *S* ►, перед запуском в его стеке должна оказаться правая часть этого правила, причем нижним символом в стеке должен быть псевдотерминал ►, а верхним соответственно начальный нетерминал грамматики. Поскольку рано или поздно псевдотерминал ► может стать верхним символом в стеке, к таблице добавляется еще одна строка, озаглавленная этим символом.

**Шаг 3.** Для каждой строки таблицы, начиная с первой, в ее клетках формируются знаки операций следующим образом.

**Шаг 3.1.** Если строка озаглавлена нетерминальным символом (пусть это будет символ *N*), то последовательно в произвольном порядке перебираются все правила грамматики, имеющие этот нетерминал в левой части.

**Шаг 3.1.1.** Если очередное правило имеет вид *N* : *M* α, где *M* – нетерминальный символ, а α – цепочка символов *s*1 *s*2 ... *sk* (возможно пустая), то во все клетки данной строки, находящиеся на пересечении со столбцами, помеченными терминалами из множества выбора данного правила, заносится такая последовательность знаков операций:

^ ! *sk* ...  *s*2  *s*1 *M*

Если среди символов *s*1 *s*2 ... *sk* встречаются терминалы, то к таблице добавляются новые строки, озаглавленные этими терминалами, но только при условии, что таких строк в ней еще нет.

**Шаг 3.1.2.**Если очередное правило имеет вид *N* : *t* α, где *t* – терминальный символ, а α – возможно пустая цепочка символов *s*1 *s*2 ... *sk*, то в клетку, находящуюся на пересечении со столбцом, помеченным терминалом *t* (очевидно, что множество выбора данного правила содержит единственный символ *t*), заносится такая последовательность знаков операций:

^ ! *sk* ...  *s*2  *s*1 >

Если среди символов *s*1 *s*2 ... *sk* встречаются терминалы, то к таблице добавляются новые строки, озаглавленные этими терминалами, но только при условии, что таких строк в ней еще нет.

Эта последовательность знаков операций завершается чтением следующего входного символа вместо записи первого символа правой части правила в стек. Причина очевидна: терминал, с которого начинается цепочка правой части правила, совпадает с текущим входным символом. Если его заносить в стек, то только для того, чтобы удалить на следующем такте.

**Шаг 3.1.3.** Если очередное правило имеет вид *N* : ε, то во все клетки данной строки, находящиеся на пересечении со столбцами, помеченными терминалами из множества выбора данного правила, заносится единственный знак операции ^. Очевидно, что удаление символа *N* с вершины стека без выполнения каких-либо других действий соответствует применению этого правила.

**Шаг 3.2.** Если текущая строка озаглавлена терминалом *t*, то в клетку, находящуюся на пересечении с одноименным столбцом (т. е. также озаглавленным терминалом *t*), заносится последовательность знаков операций ^ >. Терминал *t* мог быть занесен в стек при выполнении последовательности операций, сформированных согласно шагам 3.1.1 и 3.1.2. Если он появился на вершине стека, то входным символом обязан быть именно этот терминал, иначе входная цепочка неверна. Последовательность знаков операций ^ > обеспечивает переход к следующим символам из стека и из входной цепочки.

**Шаг 3.3***.* Наконец, если текущая строка озаглавлена псевдотерминалом ►, то в клетку, находящуюся на пересечении с одноименным столбцом, заносится знак операции Stop.

Алгоритм работы программной модели автомата очень прост и здесь не описывается.

Построение парсера как нисходящего стекового автомата   
с несколькими состояниями

Функционирование конечного автомата со стековой памятью и несколькими состояниями также определяется управляющей таблицей, но имеющей совершенно другую структуру. Предполагается, что автомат при запуске оказывается в особом начальном состоянии, а на каждом такте по входному символу и текущему состоянию он определяет и выполняет операции над входным потоком символов, стековой памятью и собственным состоянием.

Для выявления характера этих операций и структуры управляющей таблицы рассмотрим еще раз, но несколько с другой точки зрения, существо процесса нисходящего синтаксического акцепта.

Процесс нисходящего восстановления дерева грамматического разбора можно интерпретировать как управляемое входной цепочкой движение по порождающим правилам грамматики. Для такого рассмотрения удобно считать, что каждое правило завершается обозначением пустой цепочки ε. В этом случае обработка правил с пустой правой частью ничем не будет отличаться от обработки остальных правил. Управляющая таблица автомата при этом будет обладать некоторой избыточностью, впоследствии легко удаляемой.

Начиная с нетерминала *S* в правой части добавочного правила *Z*:*S*►, движение осуществляется следующим образом.

1. По правым частям правил посимвольно слева направо.

Обработка любого нетерминала состоит в переключении на первое правило для этого нетерминала. Более точно, на состояние, соответствующее нетерминалу из левой части первого правила с сохранением в стеке точки возврата в текущую правую часть правила. До переключения осуществляется проверка принадлежности текущего входного символа к множеству предшественников данного нетерминала (т. е. к объединению множеств выбора всех правил, в левой части которых находится этот нетерминал).

Обработка терминального символа состоит в проверке его совпадения с текущим входным символом и при положительном результате проверки завершается чтением следующего терминала из входной цепочки. Отрицательный результат проверки приводит к останову автомата по обнаружению ошибки.

Обработка пустой цепочки ε, завершающей каждое правило, состоит в возврате по номеру состояния, снимаемого с вершины стека. Возврат в состояние, соответствующее псевдотерминалу ►, рассматривается как успешное окончание процесса восстановления дерева при условии, что текущим входным символом является признак конца входной цепочки ►. Если же в этот момент текущим входным символом является любой другой терминал, то выполняется останов по ошибке.

1. По левым частям правил сверху вниз.

При этом движении используются только правила, имеющие в левой части один и тот же нетерминал. Для каждого правила прежде всего проверяется, содержит ли его множество выбора текущий входной символ.

При отрицательном результате проверки осуществляется переход к левой части следующего правила, тем самым обеспечивается поиск подходящего правила для замены нетерминала.

При положительном результате проверки выполняется переключение на обработку первого символа из правой части данного правила, т. е. подстановка правой части вместо нетерминала из левой части.

Если такого правила нет вообще (ни одно из множеств выбора правил для данного нетерминала не содержит текущего входного символа), то восстановить дерево невозможно и следует остановиться по обнаружении ошибки во входном предложении.

Таким образом, каждому символу каждого правила грамматики (в том числе нетерминалам, находящимся в левых частях правил, и обозначениям пустой цепочки, замыкающим каждое правило) должно быть поставлено в соответствие в точности одно состояние автомата. С каждым состоянием должно быть связано множество выбора и два адреса перехода (один используется при положительном результате проверки принадлежности текущего входного символа множеству выбора, второй – при отрицательном). Под адресом перехода понимается номер состояния. Ниже показано, что если соблюдать определенные правила нумерации состояний и ввести операцию управления остановом по ошибке, то можно обойтись только одним адресом перехода.

С каждым состоянием должны быть также связаны операции управления стековой памятью (занесение адреса возврата, снятие адреса с вершины стека и переключение в состояние возврата) и операция управления чтением следующего входного символа. Все операции управления могут задаваться булевыми значениями *true* / *false*, которые далее называются флажками. Обозначения для флажков управления операциями:

* + флажок **a** управляет чтением следующего входного символа;
  + флажок **s** управляет занесением адреса точки возврата (вычисляемого как номер текущего состояния плюс единица) в стек;
  + флажок **r** обеспечивает переключение автомата в состояние, номер которого снимается с вершины стека возвратов;
  + флажок **e** запрещает останов по ошибке в случае, когда состояние соответствует нетерминалу из левой части и есть еще хотя бы одно правило для такого нетерминала.

Таким образом, каждая клетка управляющей таблицы автомата должна содержать следующие поля:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Номер состояния | Флажки | | | | Адрес перехода | Множество выбора состояния | Действие |
| **a** | **s** | **r** | **e** |

При практических применениях автоматной реализации рекурсивного спуска в состав клетки управляющей таблицы обычно включается дополнительное поле, которое указывает на действие, сопровождающее синтаксический акцепт (например, для нейтрализации ошибок) или относящееся к задачам семантического анализа и формирования объектного кода.

Для построения управляющей таблицы автомата по заданной **LL(1)**-грамматике (в качестве иллюстрации используется грамматика ***Ga*2**,к каждой правой части правил которой дописано обозначение пустой цепочки ε) необходимо выполнить следующую процедуру.

**Шаг 1.** Определение и нумерация множества состояний. Для этого всем символам системы порождающих правил грамматики, исключая символ *Z* в левой части добавочного правила, но включая обозначения пустых цепочек, присваивается номер так, чтобы выполнялось следующее:

* символ *S* в добавочном правиле *Z* : *S*►получил номер 0;

символы, следующие друг за другом в правых частях правил, имели последовательно возрастающие номера. При соблюдении этого требования адрес возврата, помещаемый в стек при обработке нетерминального символа в правой части правила, вычисляется как номер текущего состояния плюс единица;

* одинаковые нетерминалы в левых частях правил имели последовательно возрастающие номера; при соблюдении этого требования легко обеспечивается перебор правил при обработке нетерминалов из левых частей правил .

Таблица 7

|  |  |
| --- | --- |
| Грамматика ***Ga2*** | |
| 0 | *Z* : *S*0 ▶ 1 |
| 1 | *S*2 : *U*11 *R*12 ε***13*** |
| 2 | *R*3 : + 14 *S*15 ε16 |
| 3 | *R*4 : **ε**17 |
| 4 | *U*5 : *V*18 *W*19 ε20 |
| 5 | *W*6 : \*21 *U*22 ε23 |
| 6 | *W*7 : ε24 |
| 7 | *V*8 : *(* 25 *S26**)*27 ε28 |
| 8 | *V*9 : *i*29 ε30 |
| 9 | *V*10 : *c*31 ε32 |

В табл. 7 приведены результаты выполнения шага 1 для модифицированной грамматики ***Ga*2**.

**Шаг 2.** Формирование множества выбора для каждого состояния управляющей таблицы.

Способ образования множества выбора состояния зависит от того, какому символу (терминалу, нетерминалу или пустой цепочке) и из какой части правила оно поставлено в соответствие.

Если состояние соответствует нетерминалу *N* из левой части правила *N* : α, то его множество выбора есть множество выбора данного правила:

– множество предшественников цепочки α, если она содержит хотя бы один терминал или неаннулируемый нетерминал;

– множество последователей *N*, если цепочка α пуста;

– объединение этих двух множеств, если цепочка α не пуста, но состоит только из аннулируемых нетерминалов).

Если состояние соответствует нетерминальному символу из правой части правила, то его множество выбора есть объединение множеств выбора всех правил грамматики для этого нетерминала.

Если состояние соответствует терминальному символу (такие символы могут появляться только в правых частях правил), то его множество выбора содержит только этот терминальный символ.

Для состояний, соответствующих обозначениям пустой цепочки, множества выбора есть множество последователей нетерминала из левой части данного правила.

**Шаг 3.** Формирование значений флажков управления операциями.

Флажок **a** устанавливается (имеет значение *true*) только в состояниях, соответствующих терминальным символам (которые, естественно, могут находиться только в правых частях правил).

Флажок **s** устанавливается в состояниях, соответствующих нетерминальным символам, которые находятся в правых частях правил.

Флажок **r** устанавливается в состояниях, соответствующих обозначениям пустой цепочки символов в конце правой части каждого правила.

Флажок **e** устанавливается в состояниях, соответствующих нетерминальным символам, которые находятся в левой части правил, за исключением последнего правила для каждого нетерминала.

**Шаг 4.** Образование адреса перехода.

В клетках состояний, соответствующих нетерминалам из левых частей правил, адрес перехода должен быть равен номеру состояния, соответствующего первому символу правой части данного правила.

В клетках состояний, соответствующих символам из правых частей правил, адрес перехода формируется только в том случае, если для этого состояния не установлен флажок **r** (когда флажок **r** установлен, переход осуществляется по адресу, снимаемому со стека возвратов). Если флажок в данном состоянии **r** установлен, то в поле адреса перехода будем заносить значение «0». Особое значение адреса перехода (Stop) формируется для состояния 1. Переход по этому адресу означает останов автомата по окончании восстановления дерева разбора правильного предложения при условии, что стек пуст. В противном случае (если стек не пуст) операция Stop означает останов по ошибке.

Для состояний, соответствующих терминальным символам, в поле адреса перехода заносится номер состояния, соответствующего следующему символу правила (при используемом способе нумерации состояний он вычисляется как номер текущего состояния плюс единица).

Для состояний, соответствующих нетерминальным символам в правых частях правил, в поле адреса перехода заносится номер состояния, приписанного первому такому (одноименному) нетерминалу, но находящемуся в левой части правил.

В табл. 8. приведены результаты применения этой процедуры преобразования грамматики в управляющую таблицу автомата для грамматики ***Ga*2**(в полях флажков управления значению *true* сопоставлена единица, а значению *false* – пустая клетка).

Таблица 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | | Флажки | | | | Переход | Множество выбора | Действие |
|  | **a** | **s** | **r** | **e** |  |
| 0 |  | 1 |  |  | 2 | *( i c* |  |
| 1 |  |  |  |  | Stop | ► |  |
| 2 |  |  |  |  | 11 | *( i c* |  |
| 3 |  |  |  | 1 | 14 | *+* |  |
| 4 |  |  |  |  | 17 | *)* ► |  |
| 5 |  |  |  |  | 18 | *( i c* |  |
| 6 |  |  |  | 1 | 21 | *\** |  |
| 7 |  |  |  |  | 24 | *+ )* ► |  |
| 8 |  |  |  | 1 | 25 | *(* |  |
| 9 |  |  |  | 1 | 29 | *i* |  |
| 10 |  |  |  |  | 31 | *c* |  |
| 11 |  | 1 |  |  | 5 | *( i c* |  |
| 12 |  | 1 |  |  | 3 | *+ )* ► |  |
| 13 |  |  | 1 |  | 0 | *i c\* +( )* ► |  |
| 14 | 1 |  |  |  | 15 | *+* |  |
| 15 |  | 1 |  |  | 2 | *( i c* |  |
| 16 |  |  | 1 |  | 0 | *i c\* +( )* ► |  |
| 17 |  |  | 1 |  | 0 | *i c\* +( )* ► |  |
| 18 |  | 1 |  |  | 8 | *( i c* |  |
| 19 |  | 1 |  |  | 6 | *\* + )* ► |  |
| 20 |  |  | 1 |  | 0 | *i c\* +( )* ► |  |
| 21 | 1 |  |  |  | 22 | *\** |  |
| 22 |  | 1 |  |  | 5 | *( i c* |  |
| 23 |  |  | 1 |  | 0 | *i c\* +( )* ► |  |
| 24 |  |  | 1 |  | 0 | *i c\* +( )* ► |  |
| 25 | 1 |  |  |  | 26 | *(* |  |
| 26 |  | 1 |  |  | 2 | *( i c* |  |
| 27 | 1 |  |  |  | 28 | *)* |  |
| 28 |  |  | 1 |  | 0 | *i c\* +( )* ► |  |
| 29 | 1 |  |  |  | 30 | *i* |  |
| 30 |  |  | 1 |  | 0 | *i c\* +( )* ► |  |
| 31 | 1 |  |  |  | 32 | *c* |  |
| 32 |  |  | 1 |  | 0 | *i c\* +( )* ► |  |

Этот автомат имеет определенную избыточность. Добавление обозначений пустой цепочки в конец правой части правил 1, 2, 4, 5, 7, 8 и 9 привело к образованию в управляющей таблице состояний, зарезервированных для возможного включения действий в грамматику. Эти состояния с номерами 13, 16, 20, 23, 28, 30 и 32 являются избыточными при решении задачи чистого синтаксического акцепта, т. е. без учета задач нейтрализации ошибок, семантического анализа и генерации кода.

Если для этих состояний при расширении синтаксического акцептора до анализатора так и не будут определены действия, то они легко могут быть удалены из управляющей таблицы.

Программная модель автомата с несколькими состояниями и стековой памятью должна реализовывать следующий алгоритм.

**Шаг 1.** Запуск и инициализация. Очистить стек, прочитать первый символ входной цепочки, установить в качестве текущего состояние 0 и перейти к шагу 2.

**Шаг 2.** Проверить, принадлежит ли очередной символ множеству выбора текущего состояния. Если да, то перейти к шагу 3, иначе – к шагу 6.

**Шаг 3.** Если в клетке текущего состояния установлен флажок **a**, то прочитать следующий символ входной цепочки.

**Шаг 4.** Если в клетке текущего состояния установлен флажок **s**, то поместить в стек номер текущего состояния, увеличенный на единицу.

**Шаг 5.** Определение номера следующего состояния. Для этого прежде всего проверяется значение флажка **r** текущего состояния.

**Шаг 5.1.** Если флажок **r** установлен, то выполнить следующие шаги.

**Шаг 5.1.1.** Если стек не пуст, то снять с вершины стека номер состояния, установить его в качестве текущего и перейти к шагу 2;

**Шаг 5.1.2.** Если стек пуст, то перейти к шагу 7.

**Шаг 5.2.** Если флажок **r** не установлен, то выполнить следующие шаги.

**Шаг 5.2.1.** Если текущим является состояние 1, то возможны два варианта.

**Шаг 5.2.1.1.** Если стек пуст, то перейти к шагу 8.

**Шаг 5.2.1.2.** Если стек не пуст, то перейти к шагу 7.

**Шаг 5.2.2.** Если текущим является любое другое состояние, то взять номер состояния из поля адреса перехода клетки текущего состояния. Установить в качестве текущего состояние с этим номером и вернуться к шагу 2.

**Шаг 6.** Если в клетке текущего состояния установлен флажок **e**, то установить в качестве текущего следующее состояние (его номер вычисляется как номер текущего состояния плюс единица) и вернуться к шагу 2, иначе перейти к шагу 7.

**Шаг 7.** Останов по ошибке.

**Шаг 8.** Останов по окончании разбора правильного предложения.

Построение парсера как совокупности функций   
нисходящего рекурсивного восстановления дерева разбора

Пакет «ВебТрансБилдер» предоставляет возможность преобразования **LL(1)**-грамматики в программный код, содержащий совокупность функций нисходящего рекурсивного восстановления дерева разбора.

Для каждого нетерминала грамматики создается функция, которая:

поочередно проверяет принадлежность текущего терминала из предложения множеству выбора каждого правила;

1. при положительном результате проверки «реализует» правую часть правила, двигаясь по ее символам слева направо, следующим образом:
   * вызывает соответствующие функции парсера (возможно и сама себя), если очередной символ – это нетерминал;
   * сравнивает символ из правила с текущим терминалом, если это терминал, и при этом:

* вызывает лексический анализатор для чтения следующего терминала из предложения при совпадении символов;
* возвращает значение *false* (ложь) при несовпадении;
  + возвращает значение *true* (истина), если был обработан последний символ правой части правила (или правая часть пуста);

1. возвращает значение *false* (ложь), если не было найдено ни одного подходящего правила.

Детально способ преобразования **LL(1)**-грамматики в программный код описан в [1].

1. Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample4):
   1. Используя пакет «ВебТрансБилдер», выполнить следующее:

– расширить грамматику заданного на курсовую работу языка, разработанную при выполнении работы № 3, до полной грамматики языка (или как минимум до грамматики блока операторов с реализацией правил для всех заданных операторов языка согласно варианту курсовой работы);

– изучить и освоить проверку принадлежности грамматики к классу **LL(1)** (пункт меню «Показать / Множества выбора правил»). Изучить, что такое множества выбора правил и как они формируются, а также их использование для преобразования грамматики в нисходящий синтаксический анализатор;

– добиться того, чтобы разработанная грамматика стала принадлежать классу **LL(1)**. При необходимости освоить для этого технологию удаления терминальных символов из множеств выбора правил с использованием токена «~», как это сделано в примере Sample3;

– построить конечный автомат со стековой памятью и несколькими состояниями, найти в коде построенного автомата управляющую таблицу, описать для отчета ее структуру;

– построить конечный автомат со стековой памятью и одним состоянием, управляемый входным символом и символом, снятым с вершины стека. Найти в коде построенного автомата управляющую таблицу, описать для отчета ее структуру;

– построить программную реализацию нисходящего рекурсивного спуска. Найти в коде функцию, построенную для нетерминала, определяющего оператор присваивания, и описать ее алгоритм для отчета.

* 1. Выполнить трассировку процессов нисходящего синтаксического акцепта для всех построенных синтаксических анализаторов, включая для этого флажок показа истории работы при запуске транслятора на тестирование. Изучить по историям работы поведение всех построенных синтаксических акцепторов при разборе как правильных предложений, так и предложений с намеренно внесенными синтаксическими ошибками.
  2. Проанализировать и сравнить между собой все полученные тексты программ и результаты выполнения п. 2. Оценить степень пригодности изученных вариантов реализации нисходящих синтаксических акцепторов для выполнения курсовой работы, отразить результаты этой оценки в заключении к отчету.

1. Подготовить отчет, который должен содержать следующие разделы:
   * + цель работы;
     + систему правил **LL(1)**-грамматики для заданного языка;
     + описание этой грамматики;
     + фрагменты текста процедурной реализации и управляющих таблиц автоматных реализаций нисходящего синтаксического акцептора, построенных по этой грамматике, для разбора оператора присваивания, описания структур данных и алгоритмов работы соответствующих автоматов;
     + фрагменты историй работы процедурной и автоматных реализаций нисходящего синтаксического акцептора для правильного и ошибочного тестовых примеров; описание этих фрагментов для разбора оператора присваивания;
     + выводы и заключение.
2. Контрольные вопросы.
   1. Что такое множество выбора правила грамматики?
   2. Постройте историю работы нисходящего синтаксического акцептора с одним состоянием при разборе цепочки: *(((x)))*
   3. Напишите **LL(1)**-грамматику для условного оператора С-подобного языка, имеющего и полную и сокращенную форму.
   4. Что такое s-грамматика?
   5. Какие операции способен выполнять автомат нисходящего восстановления дерева грамматического разбора с одним состоянием?
   6. Что такое **LL(1)**-грамматики?
   7. Что такое детерминированное (безвозвратное) восстановление дерева грамматического разбора?
   8. Почему леворекурсивная грамматика не может быть преобразована в нисходящий синтаксический акцептор?
   9. Опишите технологию разработки процедурной реализации рекурсивного спуска.
   10. Как вычисляются множества выбора правил грамматики?
   11. Перечислите поля клетки нисходящего автомата с несколькими состояниями.
   12. Как формируются значения флагов в управляющей таблице нисходящего синтаксического акцептора с несколькими состояниями?
   13. Разработайте **LL(1)**-грамматику для оператора переключателя (switch) С-подобного языка.
   14. В каком порядке нумеруются символы грамматики при преобразовании в управляющую таблицу автомата с несколькими состояниями?
   15. Чем управляется нисходящий автомат с одним состоянием?
   16. Каково назначение каждого управляющего флажка управляющей таблицы автомата с несколькими состояниями?
   17. Как формируются значения управляющих флагов нисходящего автомата с несколькими состояниями?
   18. Опишите последовательность действий при преобразовании грамматики в процедурную реализацию нисходящего синтаксического акцептора.
   19. Как формируются множества выбора состояний нисходящего синтаксического акцептора с несколькими состояниями?
   20. Опишите процесс функционирования нисходящего рекурсивного спуска.

## Лабораторная/практическая работа № 5

1. Название работы: «Синтаксис языков программирования. Восходящий синтаксический анализ, автоматная реализация».
2. Цели работы: изучение основных идей и понятий восходящего синтаксического анализа, свойств формальных грамматик, определяющих принадлежность грамматики к одному из классов **LR**, получение навыков построения автоматной реализации восходящего анализатора, исследование поведения восходящих синтаксических акцепторов.
3. Основные теоретические сведения:

Восходящие методы синтаксического анализа

Восходящими называются такие методы синтаксического анализа, при которых цепочка терминальных символов входного предложения шаг за шагом «сворачивается» в цепочки, содержащие и нетерминальные символы и являющиеся очередными уровнями дерева грамматического разбора. Этот процесс продолжается до тех пор, пока всё правильное предложение не окажется свернутым в цепочку, содержащую единственный нетерминальный символ – начальный нетерминал грамматики, либо до тех пор, пока не будет обнаружена невозможность восстановления дерева для неправильных предложений, т. е. ошибка.

Как и для группы нисходящих методов, главным прагматическим требованием к организации процесса сворачивания цепочек является детерминированность (однонаправленность) движения по дереву снизу вверх. Отметим, что, как и при нисходящем анализе, не любая грамматика обладает такими свойствами, чтобы по ее правилам можно было осуществлять детерминированное восходящее восстановление дерева разбора.

Структура автомата для восходящего восстановления дерева грамматического разбора

Восходящий синтаксический акцептор представляет собой конечный автомат со стековой памятью (рис. 15), управляемый входным символом и текущим состоянием.



*Рис. 15.* Восходящий парсер как автомат с несколькими состояниями   
и стеком состояний

Перед запуском автомата номер его начального состояния (состояния с номером 0) должен быть занесен в стек. Далее на каждом шаге работы номер состояния, находящийся на вершине стека, используется в качестве текущего и определяет выбор строки управляющей таблицы. Из входного предложения считывается первый терминал, этот символ находится на входе автомата и определяет колонку управляющей таблицы. На каждом шаге работы из клетки, находящейся на пересечении строки текущего состояния и столбца текущего символа, выбирается и выполняется знак операции, в результате чего может измениться содержимое стека (и соответственно текущее состояние) и символ на входе автомата. Этот процесс продолжается до тех пор, пока автомат не остановится по завершению восстановления дерева разбора правильного предложения или по обнаружению ошибки.

Знаки операций, которые могут находиться в клетках управляющей таблицы, перечислены ниже.

**S***n* – сдвиг (**Shift**) по входному предложению, т. е. чтение следующего терминала из анализируемого предложения путем вызова лексического анализатора, сопровождаемое занесением номера состояния *n* на вершину стека. Автомат переключается в состояние *n*, на его входе появляется прочитанный терминал.

**G***n* – переход (**Go**) в новое состояние. Номер состояния *n* заносится на вершину стека, автомат переключается в состояние *n*. Эта операция выполняется всегда после выполнения операции свертки, которая «образовала» нетерминальный символ и поместила его на вход автомата перед текущим терминалом. Текущий терминал не изменяется.

**R***k*,N – свертка (**Reduce**) правой части правила для нетерминала N, имеющей длину *k* символов. С вершины стека сбрасывается в никуда *k* номеров состояний (эта последовательность номеров образовалась в стеке в результате выполнения *k* предыдущих во времени операций **Shift** и **Go**, соответствующих символам правой части этого правила). Автомат переключается в состояние, номер которого остался на вершине стека (и который был там до выполнения *k* операций **Shift** и **Go**, соответствующих символам правой части этого правила). Формируется нетерминал N, временно, до выполнения операции **Go**, «вытесняющий» текущий терминал со входа автомата. В дальнейшем для удобства практической реализации вместо обозначения **R***k*,N будет использоваться обозначение **R***k*,*n*, где малое *n* обозначает номер столбца управляющей таблицы, помеченного нетерминалом N.

**Stop** – останов по завершению восстановления дерева разбора правильного предложения (пакет «ВебТрансБилдер» вместо этого обозначения формирует знак операции **S**-1; отрицательное значение в обозначении операции **Shift** трактуется как операция **Stop**).

Отсутствие любой из этих операций в клетке означает операцию останова по обнаружению ошибки.

Для преобразования данной контекстно-свободной грамматики в управляющую таблицу такого автомата может быть использовано несколько разных по сложности алгоритмов. В процессе преобразования может быть выяснено, применим ли данный алгоритм к данной грамматике, т. е. будет ли построенный автомат стопроцентно работоспособен. При получении положительного ответа на этот вопрос можно просто использовать построенный автомат в качестве парсера. При обнаружении отрицательного результата можно попытаться построить автомат, применяя более сложный алгоритм, либо попытаться модифицировать грамматику или ее свойства для повторного применения уже использовавшегося алгоритма. Алгоритмы преобразования используют то или иное понятие конфигурации. Начнем с простейшего.

Понятие конфигурации. Связь между конфигурациями, состояниями и операциями восходящего парсера

Для иллюстрации процессов, протекающих при построении восходящего парсера, будем использовать грамматику языка арифметических выражений ***Ga*1** с добавленным специальным правилом вида *Z* : *S* ► (табл. 9).

Под конфигурацией понимается правило грамматики, в котором в произвольной точке правой части помещен дополнительный метасимвол – маркер. Маркером отмечается возможное разбиение правой части правила в процессе восходящего синтаксического анализа на две цепочки: прочитанную / обработанную (находящуюся слева от маркера) и еще не обработанную, находящуюся справа от маркера. Ясно, что если правая часть правила содержит ровно *k* символов (*k* = 0, 1, 2, …), то из этого правила может быть образовано в точности *k*+ 1 различных конфигураций. Например, из правила *S* : *S + T* можно образовать следующие конфигурации (в качестве маркера используется символ ▼):

Таблица 9

|  |  |
| --- | --- |
| Грамматика ***Ga*1** | |
| 0 | *Z* : *S* EOF |
| 1 | *S* : *S + T* |
| 2 | *S* : *T* |
| 3 | *T* : *T \* V* |
| 4 | *T* : *V* |
| 5 | *V* : *( S )* |
| 6 | *V* : *ident* |
| 7 | *V* : *const* |

*S* : ▼ *S + T*

*S* : *S* ▼ *+ T*

*S* : *S +* ▼ *T*

*S* : *S + T* ▼

Для любой контекстно-свободной грамматики с конечным количеством правил множество всех возможных различных конфигураций также конечно.

Между конфигурациями, состояниями восходящего автомата и операциями, которые должны выполняться в этих состояниях, существуют определенные связи (соотношения), на основании которых может быть определен метод преобразования грамматики в управляющую таблицу.

Связи между конфигурациями и операциями

Если в некоторой конфигурации *N* : α ▼*t* β маркер находится перед терминалом, то в состоянии номер *m*, которому соответствует эта конфигурация, должна выполняться операция **Shift**, осуществляющая переключение автомата в такое состояние, которому соответствует конфигурация *N*: α *t*▼ β с маркером, размещенным после этого терминала. Знак этой операции должен находиться в той клетке управляющей таблицы, которая расположена на пересечении строки состояния *m* со столбцом, помеченным терминальным символом *t* (или номером, приписанным этому терминалу).

Если маркер в конфигурации *N* : α ▼*M* β находится перед нетерминалом, то в соответствующем ей состоянии *m* должна выполняться операция **Go**, осуществляющая переключение автомата в такое состояние, которому соответствует конфигурация с маркером, размещенным после этого нетерминала, т. е. *N*: α *M*▼ β. Знак этой операции должен находиться в той клетке управляющей таблицы, которая расположена на пересечении строки состояния *m* со столбцом, помеченным нетерминальным символом *M* (или его номером).

Наконец, если маркер в конфигурации находится после последнего символа некоторого правила, то в соответствующем этой конфигурации состоянии должна выполняться операция свертки **R***k*,*n*, удаляющая из стека *k* номеров состояний (ровно столько символов в правой части этого правила), переключающая автомат в состояние, номер которого оказался на вершине стека после этого удаления, и готовящая номер столбца *n* управляющей таблицы для последующей операции **Go**.

Связи между конфигурациями и состояниями автомата

Допустим, что каким-либо образом некоторому состоянию поставлена в соответствие определенная конфигурация. Если эта конфигурация имеет вид

*N* : α ▼ *M* β ,

где *M* – нетерминальный символ, и в грамматике есть следующие правила:

*M* : γ1

*M* : γ2

…

*M* : γ*n*,

то все конфигурации вида *M* : ▼ γ1 , *M* : ▼ γ2 , ... *M* : ▼ γ*n* также должны быть поставлены в соответствие этому состоянию автомата.

Действительно, если в некоторой цепочке терминалов, выводимой из начального нетерминала грамматики, обнаружена подцепочка, выводимая из *N*, и начинающаяся с цепочки, выводимой из α, то автомат в этот момент находится в состоянии, отмеченном маркером в конфигурации *N* : α ▼ *M* β.

Поскольку входная цепочка правильная, ее продолжение (а автомат в этот момент находится перед первым символом этого продолжения) обязано выводиться из цепочки *M* β. Следовательно, оно выводится из одной из цепочек вида γ*i*β.

В силу того, что автомат должен восстановить этот пока еще неизвестный вывод, данному его состоянию должны быть сопоставлены все конфигурации, образуемые путем помещения маркера перед первыми символами порождающих правил для нетерминала *M*. Это позволит по символам, перед которыми находятся маркеры во вновь образованных конфигурациях, определить для данного состояния знаки операций в управляющей таблице, необходимые для восстановления нужного вывода.

Конфигурации вида *M* : ▼ γ*i* называются дочерними по отношению к исходной (родительской) конфигурации вида *N* : α ▼ *M* β.

Если в какой-либо из вновь образованных конфигураций вида *M* : ▼ γ*i* маркер находится перед нетерминалом *L* (цепочка γ*i* начинается с этого нетерминала), то данному состоянию автомата должны быть поставлены в соответствие и все конфигурации вида *L* : ▼ λ*j*(дочерние по отношению к *M* : ▼ γ*i*).

Таким образом, состоянию автомата может соответствовать не одна конфигурация, а некоторое подмножество полного множества конфигураций заданной грамматики.

Пусть дано подмножество – одна или несколько конфигураций, сопоставленных состоянию автомата. Применение вышеописанного способа добавления к подмножеству конфигураций, дочерних по отношению к одной или нескольким родительским вплоть до момента, когда это подмножество перестанет изменяться (расширяться), называется его замыканием. При выполнении замыкания новые дочерние конфигурации добавляются к подмножеству только в том случае, если их там еще нет.

Пусть известно замыкание подмножества конфигураций, соответствующих некоторому состоянию автомата. Разобьем его на более мелкие подмножества, содержащие конфигурации, в которых маркер находится перед одним и тем же символом грамматики. В особое подмножество выделим конфигурации, в которых маркер находится после последнего символа правила.

Каждая из конфигураций последнего подмножества (если оно не пусто), имеющая вид *X* : χ ▼, указывает на необходимость выполнения автоматом, оказавшимся в данном состоянии, операции свертки вида **R***k*,*nx*. Здесь *k* – длина цепочки χ, а *nx* – номер колонки управляющей таблицы автомата, поставленной в соответствие нетерминальному символу *X*.

Наличие нескольких различных конфигураций в этом подмножестве является причиной возникновения конфликтов типа «свертка/свертка».

Вернемся к таким конфигурациям данного состояния, в которых маркер находится перед одним и тем же символом *w*. Очевидно, что если автомат в процессе работы оказался в данном состоянии и его текущим входным символом является *w*, то должна быть выполнена операция **Shift**, если *w* – терминал, или операция **Go**, если *w* – нетерминал. Выполнение такой операции переключает автомат в некоторое другое состояние С, номер которого заносится на вершину стека.

Применительно к подмножеству конфигураций, в которых маркер находится перед символом *w*, такую операцию следует трактовать как перенос маркера через этот символ. Полученное таким образом подмножество конфигураций должно быть сопоставлено с тем состоянием С, в которое переключается автомат. Состав этого подмножества может впоследствии измениться при его замыкании. Назовем подмножество конфигураций в его начальном составе (в момент образования путем переноса маркера через *w*) базовым (или базой) по отношению к состоянию С.

Таким образом, если известно подмножество конфигураций какого-либо состояния автомата, то могут быть определены базовые подмножества конфигураций, а после выполнения их замыкания – и полные множества конфигураций некоторых других состояний, а именно тех, в которые автомат может переключиться из исходного.

В силу того что база начального состояния автомата всегда известна – ею является конфигурация *Z*: ▼ *S* ►, на основе рассмотренных связей между состояниями, операциями и конфигурациями может быть построена процедура преобразования грамматики в управляющую таблицу восходящего синтаксического акцептора, выполняемая в два этапа.

Преобразование грамматики в управляющую таблицу   
восходящего синтаксического акцептора

***Этап 1.*** Формирование подмножеств конфигураций, соответствующих состояниям автомата. Определение набора состояний восходящего акцептора.

Подмножества конфигураций, соответствующих состояниям автомата, будем формировать в табличном представлении.

Начальное состояние таблицы подмножеств конфигураций (далее – таблицы конфигураций) для любой грамматики (с точностью до наименования начального нетерминала грамматики) определяется добавочным правилом и выглядит так, как показано в табл. 10.

Таблица 10

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Состояние | Образовано | | База | Конфигурация | Символ | Отм. |
| Из | Через |
| 0 |  |  | Да | *Z : ▼ S* ► | *S* |  |

Поясним назначение колонок таблицы конфигураций.

*Состояние* – содержит номер, присвоенный состоянию автомата, которое определяется подмножеством конфигураций.

*Образовано* – показывает, из подмножества конфигураций какого состояния образовано данное состояние путем переноса маркера и через какой символ (эти сведения понадобятся при формировании знаков операций).

*База* – просто отметка «Да», если конфигурация, записанная в строке, является базовой для данного состояния.

*Конфигурация* – собственно маркированное правило.

*Символ* – вспомогательная колонка, содержащая тот символ грамматики, перед которым в данной конфигурации находится маркер.

*Отм.* – также вспомогательная колонка. Ее назначение станет ясно из описания процедуры.

Начальное состояние таблицы соответствует моменту, когда образована база начального нового состояния.

**Шаг 1.** Замыкание. Просматриваются все конфигурации нового состояния автомата, и если в какой-либо из них маркер находится перед нетерминалом, то каждое правило грамматики, имеющее этот нетерминал в левой части, преобразуется в конфигурацию путем помещения маркера перед первым символом правой части. Для каждой вновь образованной конфигурации просматриваются все строки таблицы данного состояния. Если вновь образованной конфигурации нет ни в одной такой строке, то формируется новая строка. Новая конфигурация заносится в соответствующую клетку этой строки, а в клетку колонки, обозначенной *Символ*, заносится первый символ правой части правила (если правило имеет вид *N* : ε, то эта клетка остается пустой), все остальные клетки новой строки оставляются пустыми. Этот шаг выполняется для данного состояния до тех пор, пока не перестанут добавляться новые строки.

Построенный фрагмент таблицы конфигураций после выполнения этого шага для нулевого состояния выглядит так, как показано в табл. 11.

Таблица 11

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Состояние | Образовано | | База | Конфигурация | Символ | Отм. |
| Из | Через |
| 0 |  |  | Да | *Z* : ▼ *S* ► | *S* |  |
|  |  | *S* : ▼ *S + T* | *S* |  |
|  |  | *S* : ▼ *T* | *T* |  |
|  |  | *T* : ▼ *T \* V* | *T* |  |
|  |  | *T* : ▼ *V* | *V* |  |
|  |  | *V* : ▼ *( S )* | *(* |  |
|  |  | *V* : ▼ *i* | *i* |  |
|  |  | *V* : ▼ *c* | *c* |  |

**Шаг 2.** Образование новой базы. Просматривается колонка *Отм*. с целью найти такую строку таблицы, которая еще не помечена и в колонке *Символ* содержит какой-либо символ грамматики, отличный от псевдотерминала ► (конец файла). Если таких строк нет, то этап построения таблицы конфигураций завершается.

Если такая строка найдена, то зафиксируем номер состояния (далее оно называется исходным), а затем найдем и отметим те строки таблицы, которые принадлежат данному состоянию и содержат тот же самый символ в соответствующей колонке. Выберем конфигурации из соответствующей колонки этих строк и перенесем в каждой из них маркер через один символ. Тем самым будет образовано новое базовое подмножество конфигураций.

**Шаг 3.** Проверка наличия состояния с таким набором базовых конфигураций, который совпадает с вновь образованной базой. Просматривается таблица конфигураций и база каждого уже существующего состояния (в том числе исходного) сравнивается с новой базой, сформированной на предыдущем шаге. Возможно два случая.

**Шаг 3.1.** Состояние, имеющее в точности такую же базу, уже существует (подчеркнем, что для этого новая и уже существующая база должны совпадать полностью). В этом случае добавим в колонки *Из* и *Через* этого состояния номер исходного состояния и символ, через который был перенесен маркер для образования базы. Эти данные пригодятся на втором этапе при преобразовании таблицы конфигураций в управляющую таблицу. Возвращаемся к шагу 2.

**Шаг 3.2.** Не найдено состояния, имеющего в точности такую же базу, как и проверяемая (вновь образованная). В этом случае создается новое состояние.

В таблицу добавляется столько строк, сколько конфигураций содержится в проверяемой базе. Клетки во вновь добавленных строках заполняются в соответствии с назначением колонок. После завершения формирования нового состояния осуществляется возврат к шагу 1.

Описание процедуры завершено. Результат ее применения к грамматике ***Ga*1** приведен в табл. 12.

Заметим, что в колонке *Отм.* указан порядковый номер применения второго шага процедуры. Шаги с первого по девятый приводили к образованию таких подмножеств базовых конфигураций, которых к моменту выполнения каждого шага еще не было в таблице. Поэтому были сформированы состояния с номерами 1–9.

На десятом применении второго шага в результате переноса маркера через символ *T* были получены конфигурации *S*: *T*▼ и *T*: *T*▼*\* V* , в точности совпадающие с базой состояния номер 2. Согласно третьему шагу процедуры новое состояние не создано; в колонке *Из* состояния 2 отмечено, что его база получена из конфигураций как состояния 0, так и состояния 4.

Начиная с одиннадцатого новые состояния были созданы только на 15, 20 и 24-м применении второго шага процедуры. Все остальные шаги приводили к образованию уже существующих базовых подмножеств конфигураций.

***Этап 2.*** Преобразование таблицы конфигураций в управляющую таблицу восходящего парсера.

В результате построения таблицы конфигураций (табл. 12) было выяснено, что автомат, который реализует восходящее восстановление дерева грамматического разбора предложений языка, порождаемого грамматикой ***Ga*1**, должен иметь ровно 13 состояний (с номерами 0…12). Общее количество символов (как терминальных, так и нетерминальных) вместе с псевдотерминалом ► равно 10.

Таблица 12

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Состояние* | *Образовано* | | *База* | *Конфигурация* | *Символ* | *Отм.* |
| *Из* | *Через* |
| 0 |  |  | Да | *Z* : ▼ *S* ► | *S* | 1 |
|  |  | *S* : ▼ *S + T* | *S* | 1 |
|  |  | *S* : ▼ *T* | *T* | 2 |
|  |  | *T* : ▼ *T \* V* | *T* | 2 |
|  |  | *T* : ▼ *V* | *V* | 3 |
|  |  | *V* : ▼ *( S )* | *(* | 4 |
|  |  | *V* : ▼ *i* | *i* | 5 |
|  |  | *V* : ▼ *c* | *c* | 6 |
| 1 | 0 | *S* | Да | *Z* : *S* ▼► |  |  |
| *S* | Да | *S* : *S* ▼ *+ T* | *+* | 7 |
| 2 | 0, 4 | *T* | Да | *S* : *T* ▼ |  |  |
| *T* | Да | *T* : *T* ▼ *\* V* | *\** | 8 |
| 3 | 0, 4, 7 | *V* | Да | *T* : *V* ▼ |  |  |
| 4 | 0, 4, 7, 8 | *(* | Да | *V* : *(*▼ *S )* | *S* | 9 |
|  |  | *S* : ▼ *S + T* | *S* | 9 |
|  |  | *S* : ▼ *T* | *T* | 10 |
|  |  | *T* : ▼ *T \* V* | *T* | 10 |
|  |  | *T* : ▼ *V* | *V* | 11 |
|  |  | *V* : ▼ *( S )* | *(* | 12 |
|  |  | *V* : ▼ *i* | *i* | 13 |
|  |  | *V* : ▼ *c* | *c* | 14 |
| 5 | 0, 4, 7, 8 | *i* | Да | *V* : *i* ▼ |  |  |
| 6 | 0, 4, 7, 8 | *i* | Да | *V* : *c* ▼ |  |  |
| 7 | 1, 9 | *+* | Да | *S* : *S +* ▼ *T* | *T* | 15 |
|  |  | *T* : ▼ *T \* V* | *T* | 15 |
|  |  | *T* : ▼ *V* | *V* | 16 |
| *Из* | *Через* |  |  |  |  |
|  |  |  |  | *V* : ▼ *( S )* | *(* | 17 |
|  |  | *V* : ▼ *i* | *i* | 18 |
|  |  | *V* : ▼ *c* | *c* | 19 |
| 8 | 2, 10 | *\** | Да | *T* : *T \** ▼ *V* | *V* | 20 |
|  |  | *V* : ▼ *( S )* | *(* | 21 |
|  |  | *V* : ▼ *i* | *i* | 22 |
|  |  | *V* : ▼ *c* | *c* | 23 |
| 9 | 4 | *S* | Да | *V* : *( S* ▼ *)* | *)* | 24 |
| *S* | Да | *S* : *S* ▼*+ T* | *+* | 25 |
| 10 | 7 | *T* | Да | *S* : *S + T* ▼ |  |  |
| *T* | Да | *T* : *T* ▼*\* V* | *\** | 26 |
| 11 | 8 | *V* | Да | *T* : *T \* V* ▼ |  |  |
| 12 | 9 | *)* | Да | *V* : *( S )* ▼ |  |  |

Таблица конфигураций преобразуется в управляющую таблицу путем выполнения следующей процедуры.

**Шаг 1.** Строится пустая заготовка управляющей таблицы автомата, содержащая 13 строк и 10 столбцов (не считая заголовочных). Строки и столбцы нумеруются начиная с нуля. Нетерминальные символы грамматики в произвольном порядке помещаются в заголовки столбцов начиная с нулевого. Затем в произвольном порядке формируются заголовки столбцов, соответствующих терминальным символам. Последняя колонка ставится в соответствие псевдотерминальному символу конца входной цепочки ►.

**Шаг 2.** Занесение знаков операций **Stop**, **Shift** и **Go**. Знак операции **Stop** заносится в клетку строки состояния 1, колонка которой озаглавлена псевдотерминалом ►. Это соответствует моменту окончания восстановления дерева разбора согласно конфигурации *Z* : *S* ▼►.

Просматриваются колонки *Из* и *Через* таблицы конфигураций (табл. 12). Для каждой непустой пары значений из этих колонок формируется знак операции **Shift**, если символ в колонке *Через* является терминальным, и знак операции **Go** – в противном случае. Номер состояния, взятый из первой колонки текущей строки таблицы конфигураций, является параметром знака операции. Построенный таким образом знак операции заносится в клетку управляющей таблицы, находящуюся на пересечении строки с номером, взятым из колонки *Из*, и столбца, озаглавленного символом из колонки *Через*.

Например, при просмотре первой строки таблицы конфигураций, соответствующей состоянию номер 1, будет образован один знак операции **G**1, который должен быть занесен в клетку строки номер 0 того столбца управляющей таблицы, который помечен символом *S*. При обработке первой строки состояния 3 таблицы конфигураций будут образованы три одинаковых знака операции **S**3, которые должны быть помещены в клетки столбца, помеченного нетерминалом *V*, находящиеся на его пересечении со строками номер 0, 4 и 7.

**Шаг 3.** Занесение знаков операций **Reduce**. Просматривается содержимое колонки *Конфигурация* в поисках такой конфигурации, в которой маркер находится после последнего символа правой части правила. Формируется знак операции **R***k*,*n*, где *k* – количество символов в правой части правила; *n* – номер столбца управляющей таблицы, помеченного нетерминалом из левой части этого правила.

Выполняется попытка занесения сформированного знака операции во все клетки той строки управляющей таблицы, которая имеет номер, взятый из первой колонки таблицы конфигураций. Знак операции свертки не заносится в клетки, столбцы которых помечены нетерминалами. Если клетка уже содержит знак операции **Shift** или **Reduce**, то фиксируется конфликт типа «сдвиг/свертка» или «свертка/свертка» соответственно. Вопрос о том, что делать при возникновении конфликтов, будет рассматриваться ниже.

Применение этой процедуры к ранее построенной нами таблице конфигураций (см. табл. 12) позволит получить управляющую таблицу автомата (табл. 13).

В качестве примера выполнения шага 3 рассмотрим состояние номер 2, в наборе конфигураций которого имеется конфигурация *S* : *T*▼. Формируется знак операции **R**1,0, поскольку правая часть правила содержит один символ *T*, а нетерминалу *S* из левой части поставлен в соответствие столбец управляющей таблицы с номером 0. При занесении этого знака в строку номер 2 управляющей таблицы будет обнаружен конфликт типа «сдвиг/свертка» в клетке столбца, помеченного терминалом \*. Возникновение конфликта легко объясняется наличием в наборе конфигураций состояния номер 2 конфигурации *T* : *T*▼*\* V*.

Таблица 13

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| *S* | | *T* | *V* | *+* | *\** | *(* | *)* | *i* | *c* | ► |
| 0 | **G**1 | | **G**2 | **G**3 |  |  | **S**4 |  | **S**5 | **S**6 |  |
| 1 |  | |  |  | **S**7 |  |  |  |  |  | **Stop** |
| 2 |  | |  |  | **R**1,0 | **S**8 | **R**1,0 | **R**1,0 | **R**1,0 | **R**1,0 | **R**1,0 |
| 3 |  | |  |  | **R**1,1 | **R**1,1 | **R**1,1 | **R**1,1 | **R**1,1 | **R**1,1 | **R**1,1 |
| 4 | **G**9 | | **G**2 | **G**3 |  |  | **S**4 |  | **S**5 | **S**6 |  |
| 5 |  | |  |  | **R**1,2 | **R**1,2 | **R**1,2 | **R**1,2 | **R**1,2 | **R**1,2 | **R**1,2 |
| 6 |  | |  |  | **R**1,2 | **R**1,2 | **R**1,2 | **R**1,2 | **R**1,2 | **R**1,2 | **R**1,2 |
| 7 |  | | **G**10 | **G**3 |  |  | **S**4 |  | **S**5 | **S**6 |  |
| 8 |  | |  | **G**11 |  |  | **S**4 |  | **S**5 | **S**6 |  |
| 9 |  | |  |  | **S**7 |  |  | **S**12 |  |  |  |
| 10 |  |  | |  | **R**3,0 | **S**8 | **R**3,0 | **R**3,0 | **R**3,0 | **R**3,0 | **R**3,0 |
| 11 |  |  | |  | **R**3,1 | **R**3,1 | **R**3,1 | **R**3,1 | **R**3,1 | **R**3,1 | **R**3,1 |
| 12 |  |  | |  | **R**3,2 | **R**3,2 | **R**3,2 | **R**3,2 | **R**3,2 | **R**3,2 | **R**3,2 |

В табл. 13 серым фоном отмечены две клетки, в которых при занесении знаков операций были зафиксированы конфликты типа «сдвиг/свертка».

Возникновение конфликтов, казалось бы, свидетельствует о недетерминированности конечного автомата, т. е. о невозможности восстановления дерева разбора некоторых входных цепочек без возвратов к пересмотру ранее принятых решений. Знаки операций сдвига и свертки не могут быть помещены в одну клетку управляющей таблицы в силу противоположности смысла действий, выполняемых этими операциями. То же самое относится и к потенциально возможным конфликтам типа «свертка/свертка».

В одной клетке управляющей таблицы может находиться только один знак операции. Поэтому необходимо выяснить, может ли каждый конфликт быть разрешен в пользу одного из знаков операций. Только тогда можно считать завершенным процесс преобразования грамматики в восходящий синтаксический акцептор. Для разрешения конфликтов, очевидно, необходимо исследовать свойства исходной грамматики и порождаемого ею языка.

Однако в действительности для многих грамматик, в том числе для рассматриваемой грамматики ***Ga*1**, возникновение конфликтов обусловлено не их свойствами и даже не свойствами порождаемого языка, а лишь несовершенством способа занесения знаков операций свертки в управляющую таблицу, сформулированного в шаге 3 процедуры.

Предотвращение конфликтов путем использования множеств последователей нетерминальных символов

Недетерминированность автомата, обнаруженная при попытке занесения знака операции свертки **R**1,0 (или **R**3,0) в клетку состояния номер 2 (или 10) (см. табл. 13), находящуюся на пересечении со столбцом, помеченным терминалом \* и содержащую знак операции **S**8, следует понимать как возможность реализации более чем одной истории работы, начиная с момента выполнения одной из этих двух конфликтующих операций.

Причина различий в поведении автоматов состоит в следующем. Выполнение операции свертки в состоянии 2 (или 10) при входном символе \* эквивалентно преобразованию промежуточного уровня восстанавливаемого дерева разбора, имеющего вид

… S + T \* …,

в цепочку вида

… *S \** …

В этой цепочке сразу после нетерминала *S* следует знак операции умножения \*. Однако множество последователей нетерминала *S* содержит только символы *+*, *)* и ►. Символа \* в этом множестве нет.

Следовательно, в процессе восстановления дерева разбора выполнять любую операцию свертки имеет смысл только в том случае, если в текущем состоянии автомата его входным символом является элемент множества последователей нетерминала, образующегося в результате свертки.

Поэтому при построении управляющей таблицы автомата сформированные на шаге 3 знаки операций свертки следует заносить только в те клетки строки данного состояния, которые находятся на пересечении со столбцами, соответствующими терминальным символам множества последователей нетерминала из левой части правила.

Используя множества последователей нетерминалов, применим модифицированный шаг 3 процедуры преобразования таблицы конфигураций в управляющую таблицу автомата к грамматике ***Ga*1**. Теперь не возникают ранее фиксировавшиеся конфликты типа «сдвиг/свертка», и управляющая таблица автомата получает вид, показанный в табл. 14.

Теперь автомат является детерминированным.

Для грамматики ***Ga*1** при построении управляющей таблицы не возникали конфликты типа «свертка/свертка».

Таблица 14

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | | 2 | | 3 | 4 | 5 | | 6 | | 7 | | 8 | | 9 | |
| *S* | *T* | | *V* | | *+* | *\** | *(* | | *)* | | *i* | | *c* | | ► | |
| 0 | **G**1 | **G**2 | **G**3 | |  | |  | | **S**4 | |  | | **S**5 | | **S**6 | |  |
| 1 |  |  |  | | **S**7 | |  | |  | |  | |  | |  | | **Stop** |
| 2 |  |  |  | | **R**1,0 | | **S**8 | |  | | **R**1,0 | |  | |  | | **R**1,0 |
| 3 |  |  |  | | **R**1,1 | | **R**1,1 | |  | | **R**1,1 | |  | |  | | **R**1,1 |
| 4 | **G**9 | **G**2 | **G**3 | |  | |  | | **S**4 | |  | | **S**5 | | **S**6 | |  |
| 5 |  |  |  | | **R**1,2 | | **R**1,2 | |  | | **R**1,2 | |  | |  | | **R**1,2 |
| 6 |  |  |  | | **R**1,2 | | **R**1,2 | |  | | **R**1,2 | |  | |  | | **R**1,2 |
| 7 |  | **G**10 | **G**3 | |  | |  | | **S**4 | |  | | **S**5 | | **S**6 | |  |
| 8 |  |  | **G**11 | |  | |  | | **S**4 | |  | | **S**5 | | **S**6 | |  |
| 9 |  |  |  | | **S**7 | |  | |  | | **S**12 | |  | |  | |  |
| 10 |  |  |  | | **R**3,0 | | **S**8 | |  | | **R**3,0 | |  | |  | | **R**3,0 |
| 11 |  |  |  | | **R**3,1 | | **R**3,1 | |  | | **R**3,1 | |  | |  | | **R**3,1 |
| 12 |  |  |  | | **R**3,2 | | **R**3,2 | |  | | **R**3,2 | |  | |  | | **R**3,2 |

В том случае, если набор конфигураций некоторого состояния содержит две или более конфигураций вида

*N* : β▼

*M* : β▼,

занесение знаков операций свертки без использования множеств последователей нетерминалов *N* и *M* приведет к возникновению конфликтов типа «свертка/свертка» в каждой клетке данного состояния.

При использовании множеств последователей для занесения знаков операций свертки в таблицу возникновение конфликтов зависит от того, пересекаются ли *M*посл(*N*) и *M*посл(*M*). Если их пересечение пусто, то в данном состоянии конфликты типа «свертка/свертка» не возникнут.

Использование множеств последователей для занесения знаков операций свертки в управляющую таблицу восходящего синтаксического акцептора автоматически разрешает вопрос о возможности использования в грамматике правил вида *N* : ε.

Действительно, конфигурация вида *N* : ε ▼ (или эквивалентная ей *N* : ▼), требующая выполнения свертки пустой цепочки в нетерминал *N*, появляется в таблице конфигураций только для тех состояний, для которых это диктуется совокупностью всех правил грамматики. Знаки операций свертки по таким правилам появятся в управляющей таблице именно в этих состояниях и только в тех столбцах, которые помечены терминалами, принадлежащими множеству последователей *N*.

Отсутствие лишних знаков операции свертки в клетках столбцов управляющей таблицы, помеченных терминалами, не входящими в множества последователей соответствующих нетерминалов, обеспечивает более быстрое обнаружение ошибок в неправильных входных цепочках.

LR(0)- и SLR(1)-грамматики и автоматы

Если при преобразовании грамматики ***G*** в управляющую таблицу восходящего синтаксического акцептора не возникает ни одного конфликта типа «сдвиг/свертка» или «свертка/свертка» даже без использования множеств последователей нетерминальных символов для расстановки знаков операций свертки, то ***G*** называется **LR(0)**-грамматикой.

Здесь буква **L** является сокращением английского слова «Left» и означает, что входная цепочка символов обрабатывается слева направо. Буква **R** есть сокращение слова «Right» и означает, что история работы автомата отражает процесс восстановления так называемого обратного правого дерева грамматического разбора. Число в скобках обозначает наименьшую длину подцепочек входных символов, необходимых для разрешения или предотвращения всех конфликтов типа «сдвиг/свертка» или «свертка/свертка».

Таблица 15

|  |  |
| --- | --- |
| Грамматика ***G****a*0 | |
| 0 | *Z* : *S* ⯈ |
| 1 | *S* : *T* |
| 2 | *S* : *S + T* |
| 3 | *T* : *( S )* |
| 4 | *T* : *ident* |
| 5 | *T* : *const* |

Обозначение **LR(0)**-грамматик свидетельствует о том, что при их использовании для построения управляющей таблицы восходящего синтаксического акцептора конфликтов не возникает вообще. Грамматики, относящиеся к классу **LR(0)**, существуют, в чем можно убедиться при построении восходящего акцептора для системы порождающих правил, показанной в табл. 15.

Однако синтаксис типичных языков программирования не может быть определен **LR(0)**-грамматикой. Этот класс грамматик слишком узок и может представлять только теоретический интерес.

Если при преобразовании грамматики в управляющую таблицу восходящего синтаксического акцептора все конфликты типа «сдвиг/свертка» или «свертка/свертка» удается разрешить (или предупредить) с использованием множеств последователей нетерминальных символов, содержащих цепочки терминалов длины 1, то грамматика относится к классу **SLR(1)**-грамматик. Буква **S** в этом обозначении означает сокращение английского слова «Simple» – простая.

Преобразование грамматики ***Ga*1** в управляющую таблицу восходящего акцептора потребовало, как мы убедились, использования множеств последователей нетерминалов, содержащих одиночные терминальные символы. Поскольку при этом все конфликты были предупреждены, данная грамматика относится к классу **SLR(1)**.

Для большинства современных языков программирования не существует порождающих грамматик класса **SLR(1)**. Поэтому необходимо рассмотреть более широкий класс грамматик, на основе которых может быть организовано восходящее детерминированное восстановление дерева разбора. Этот класс грамматик называют **LR(1)**-грамматиками.

Ожидаемый правый контекст и LR(1)-автоматы

Для разрешения конфликтов типа «сдвиг/свертка» и «свертка/свертка» при построении таблицы конфигураций нужно связать с каждой конфигурацией, требующей выполнения операции свертки, такие множества терминальных символов, которые могут действительно ожидаться на входе автомата после выполнения операции свертки. Очевидно, что эти множества должны быть подмножествами полного множества последователей для каждого нетерминала, находящегося в левой части любой такой конфигурации.

Определять такие множества придется для всех конфигураций в таблице. Это следует из того, что любое состояние восходящего акцептора определяется взаимосвязанным набором конфигураций, перечень которых зависит, во-первых, от того, как (из какого состояния и через какой символ) была сформирована база, а во-вторых, от того, каково множество конфигураций замыкания базы.

Множество символов, допустимых на входе автомата после выполнения операции свертки в любом состоянии (и, естественно, после операции **Go**, обрабатывающей нетерминал, полученный в результате этой свертки), в литературе принято называть ожидаемым правым контекстом (ОПК). Конфигурацию, к которой приписан в точности один символ ожидаемого правого контекста, будем называть канонической расширенной конфигурацией:

*N* : α ▼ β, { *t* }

Рассмотрим правила формирования ожидаемого правого контекста.

1. Ожидаемый правый контекст базовой конфигурации нулевого состояния состоит из псевдотерминального символа ► конца текста (файла). Действительно, если входное предложение является правильным и восходящему акцептору удалось свернуть его в начальный нетерминал грамматики, то после этой свертки на входе автомата никаких других символов быть не может.
2. Перенос маркера через любой символ (образование базы другого состояния, а при работе автомата – выполнение операции **Shift** или **Go**) порождает конфигурацию, правый контекст которой совпадает с правым контекстом исходной конфигурации. Действительно, если взять базовую конфигурацию нулевого состояния и перенести маркер через начальный нетерминал (выполнить операцию **Go** после свертки правильного предложения в начальный нетерминал грамматики), то на входе автомата должен появиться псевдотерминал конца текста.
3. Осталось выяснить, каким образом формируется ожидаемый правый контекст при выполнении замыкания базового набора конфигураций произвольного состояния. Суть процесса замыкания, как определялось ранее, состоит в следующем.

Если имеется конфигурация вида *N* : α ▼ *X* β , то каждое правило для нетерминала *X* превращается в конфигурацию путем установки маркера перед первым символом правой части и добавляется к множеству конфигураций данного состояния, если ее в нем еще нет:

*X* : γ1 → *X* : ▼ γ1

*X* : γ2 → *X* : ▼ γ2

...

*X* : γ*n* → *X* : ▼ γ*n*.

При построении таблицы канонических расширенных конфигураций с любой исходной конфигурацией *N* : α ▼ *X* β связан правый контекст, содержащий символ *t*0, ожидаемый на входе автомата после свертки цепочки α *X* β в нетерминал *N*:

*N* : α ▼ *X* β , {*t*0 }.

Очевидно, что после свертки любой из цепочек γ*i* в нетерминал *X* на входе автомата ожидаются символы, входящие в множество предшественников цепочки β, а если эта цепочка состоит только из аннулируемых нетерминалов или вообще пуста – еще и символ *t*0. Таким образом, определение множества ожидаемых символов при замыкании любой конфигурации производится по формуле

*M*опк(*X* : ▼γi) = *M*пред(β),

если β – цепочка, содержащая хотя бы один терминал или неаннулируемый нетерминал;

*M*опк(*X* : ▼γi) = { *t*0 },

если β – пустая цепочка;

*M*опк(*X* : ▼γi) = *M*пред(β) ⋃ { *t*0 },

если β – цепочка, состоящая только из аннулируемых нетерминалов.

Получив множество символов ожидаемого правого контекста *M*опк(α ▼ *X* β, {t0}) = {*t*1*,* *t*2,…, *tk*}, легко можно построить все дочерние по отношению к исходной канонические конфигурации:

*M* : ▼γ1 , {*t*1}*M* : ▼γ1 , {*t*2}…*M* : ▼γ1 , {*tk*}

*M* : ▼γ2 , {*t*1}*M* : ▼γ2 , {*t*2}…*M* : ▼γ2 , {*tk*}

...

*M* : ▼γ*n*, {*t*1}*M* : ▼γ*n* , { *t*2}…*M* : ▼γ*n* , {*tk*}

Теперь окончательно сформулируем правила построения таблицы канонических конфигураций.

1. Базой нулевого состояния является конфигурация вида

*Z* : ▼*S* ► , {►},

где *S* – начальный нетерминал грамматики; ► – псевдотерминал «конец файла».

1. При формировании замыкания каждая вновь построенная каноническая конфигурация добавляется к множеству конфигураций данного состояния, если в этом множестве еще нет в точности такой конфигурации (с учетом ожидаемого правого контекста).
2. При формировании базы новое состояние образуется, если нет ни одного состояния с точно таким же базовым набором канонических конфигураций. Еще раз подчеркнем, что канонические конфигурации различаются, если их помеченные правила одинаковы, но различны символы ожидаемого правого контекста.

Грамматики, для которых детерминированный восходящий синтаксический акцептор может быть построен только с использованием канонических конфигураций, называются **LR(1)**-грамматиками.

Однако грамматика ***G*LR** на самом деле относится к промежуточному между **SLR(1)**- и **LR(1)**-грамматиками классу грамматик – к так называемым **LALR(1)**-грамматикам. Буквы **LA** в названии класса грамматик являются сокращением от английского термина «lookahead», который переводится как «предварительный просмотр». В данном случае речь идет о предварительном (выполняемом при построении управляющей таблицы автомата) просмотре множеств символов ожидаемого правого контекста для разрешения или предупреждения возможных конфликтов «сдвиг/свертка» и «свертка/свертка».

LALR(1)-грамматики и автоматы

Рассмотренный в предыдущем разделе метод построения восходящего синтаксического акцептора можно модифицировать, используя вместо канонических так называемые расширенные конфигурации. Расширенной называется конфигурация, с которой связано некоторое множество символов ожидаемого правого контекста:

*N* : α ▼ β , {*t*1, *t*2, … *t*k}.

В отличие от канонических две расширенные конфигурации, имеющие одинаковое маркированное правило и разные множества символов ожидаемого правого контекста, могут быть объединены в одну конфигурацию путем слияния ОПК:

*N* : α ▼ β , {*t*1} и *N* : α ▼ β , {*t*2} эквивалентны *N* : α ▼ β , {*t*1, *t*2}.

Сформулируем две разные технологии построения таблицы расширенных конфигураций. Согласно первой технологии должно быть выполнено следующее.

1. Строится таблица канонических конфигураций так, как было показано выше.
2. Просматриваются множества конфигураций каждого состояния. Если в них обнаруживаются пары конфигураций с одинаковым маркированным правилом, то каждая такая пара заменяется одной расширенной конфигурацией с объединенным ожидаемым правым контекстом. Если одна из конфигураций пары являлась базовой, то базовой должна быть и конфигурация, полученная в результате объединения пары конфигураций. Если изменилось множество символов ОПК какой-либо базовой конфигурации (обозначим ее Кисх), то пересчитываются ожидаемые правые контексты всех ее дочерних конфигураций, т. е. таких, для которых Кисх является родительской либо при выполнении замыкания, либо при образовании другой базы. Процедура пересчета должна продолжаться рекурсивно до тех пор, пока фиксируются изменения ожидаемого правого контекста дочерних конфигураций.
3. Просматриваются все возможные пары состояний. Если два состояния имеют одинаковые (по маркированным правилам) наборы базовых расширенных конфигураций, то эти состояния сливаются, а ожидаемые правые контексты пар всех конфигураций с одинаковым маркированным правилом объединяются. При слиянии состояний одно из них удаляется, но в оставшемся состоянии должна быть сохранена информация о том, откуда образовалось удаляемое состояние, т. е. содержимое колонок *Из* и *Через*. Затем, если изменилось множество символов ОПК какой-либо базовой конфигурации (обозначим ее Кисх), то пересчитываются ожидаемые правые контексты всех ее дочерних конфигураций, т. е. таких, для которых Кисх является родительской либо при выполнении замыкания, либо при образовании другой базы. Процедура пересчета должна продолжаться рекурсивно до тех пор, пока фиксируются изменения ожидаемого правого контекста дочерних конфигураций.

Перед слиянием каждой пары состояний можно проверять, не приведет ли это слияние к конфликту, которого не было в исходной таблице канонических конфигураций. Просматриваются пересечения множеств символов ОПК всех возможных пар конфигураций, первая из которых принадлежит одному состоянию, а вторая – другому. Если пересечение не пусто и конфигурации требуют свертки по разным правилам, то состояния слиянию не подлежат. Далее, если одна из конфигураций требует сдвига по терминалу, принадлежащему ожидаемому правому контексту другой конфигурации, то выполнять слияние состояний не следует.

Согласно такой технологии строятся восходящие синтаксические акцепторы в учебном пакете автоматизации проектирования трансляторов «ВебTрансБилдер».

Вторая технология заключается в следующем.

1. Базой нулевого состояния является конфигурация вида

*Z* : ▼*S* ► , {►},

где *S* – начальный нетерминал грамматики; ► – псевдотерминал «конец файла».

1. При формировании замыкания каждая вновь построенная расширенная конфигурация добавляется к множеству конфигураций данного состояния, если в этом множестве еще нет конфигурации с точно таким же маркированным правилом. Если же в множестве конфигураций состояния уже есть конфигурация с точно таким же маркированным правилом (обозначим ее Кисх), то ее множество символов ожидаемого правого контекста объединяется с ОПК новой конфигурации. Если ожидаемый правый контекст конфигурации Кисх изменился в результате объединения, то пересчитываются ожидаемые правые контексты всех тех конфигураций, для которых конфигурация Кисх является родительской как при формировании новой базы, так и при замыкании множеств конфигураций. Процедура пересчета должна продолжаться рекурсивно до тех пор, пока не перестанут изменяться ожидаемые правые контексты дочерних конфигураций.
2. При формировании базы новое состояние образуется, если нет ни одного состояния с точно таким же (по маркированным правилам) базовым набором расширенных конфигураций. Если же существует такое состояние, то новое состояние не формируется. Вместо этого объединяются множества ОПК одинаковых по маркированным правилам базовых конфигураций. Если при этом изменился ожидаемый правый контекст какой-либо базовой конфигурации, то должны быть пересчитаны ОПК всех ее дочерних конфигураций. Процедуру пересчета следует продолжать рекурсивно до тех пор, пока не перестанут изменяться ожидаемые правые контексты дочерних конфигураций.

Обе технологии требуют выполнения большого объема вычислений при пересчете ожидаемых правых контекстов.

При использовании расширенных конфигураций для предупреждения конфликтов используется тот же самый способ формирования базовых конфигураций состояний автомата, как и в случае формирования таблицы простых конфигураций. Поэтому размеры управляющей таблицы для любой грамматики будут одинаковы как при использовании расширенных конфигураций, так и без вычисления ожидаемого правого контекста. Грамматики, для которых удается построить детерминированный восходящий синтаксический акцептор с использованием расширенных конфигураций, называются **LALR(1)**-грам­матиками.

Класс **LALR(1)**-грамматик более широк, чем класс **SLR(1)**, однако существуют языки (в том числе практически все известные языки программирования), для которых не может быть найдена порождающая **LALR(1)**-грамматика. В то же время доказано, что для любого детерминированного контекстно-свободного языка существует хотя бы одна порождающая **LR(1)**-грамматика. Тем не менее задача нахождения грамматики требуемого класса, даже если известно, что она должна существовать для данного языка, алгоритмически неразрешима.

1. Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample5):
   1. Завершить разработку грамматики заданного языка. Грамматика должна определять программы, включающие в себя несколько функций.
   2. Используя пакет «ВебТрансБилдер» и полную грамматику своего языка, выполнить следующие действия:

– построить табличную реализацию восходящего синтаксического анализатора. Найти в программном коде парсера управляющую таблицу автомата и его программную модель. Описать в отчете эти фрагменты кода парсера;

– построить процедурную реализацию восходящего синтаксического анализатора. Сравнить текст построенного программного модуля с кодом табличного парсера. Описать результаты сравнения в отчете;

– изучить структуру таблицы канонических и таблицы расширенных конфигураций (пункты меню «Показать / Канонические конфигурации» и «Показать / Расширенные конфигурации»). Описать на примерах операций **Go**, **Shift** и **Reduce** (по одной, взятой произвольным образом из управляющей таблицы) связь этих таблиц с управляющей таблицей восходящего парсера;

– выявить конфликты типов «сдвиг/свертка» и «свертка/свертка», разрешенные и, возможно, не разрешенные преобразователем. Описать способы разрешения конфликтов.

* 1. Выполнить трассировку процессов нисходящего синтаксического акцепта, включая флажок показа истории работы при запуске транслятора на тестирование. Изучить по историям работы поведение всех построенных синтаксических акцепторов при разборе как правильных предложений, так и предложений с намеренно внесенными синтаксическими ошибками. Описать фрагменты этих историй, включающие не менее чем по одной операции **Go**, **Shift** и **Reduce**.
  2. Проанализировать и сравнить между собой все полученные тексты программ и результаты выполнения п. 2. Оценить степень пригодности изученных вариантов реализации нисходящих синтаксических акцепторов для выполнения курсовой работы. Отразить в заключении к отчету результат анализа и оценки.

1. Подготовить отчет, который должен содержать следующие разделы:
   * + цель работы;
     + фрагменты таблицы расширенных конфигураций; описание процесса построения таблицы;
     + фрагменты кода процедурной реализации и управляющей таблицы автоматной реализации восходящего синтаксического акцептора; описание структур данных и алгоритмов построения управляющей таблицы и работы этих реализаций;
     + фрагменты историй работы процедурной и автоматной реализаций восходящего синтаксического акцептора для правильного и ошибочного тестовых примеров с объяснением принципов работы акцепторов;
     + выводы и заключение.
2. Контрольные вопросы
   1. Что хранится в стеке восходящего парсера?
   2. Что такое операция перехода восходящего парсера?
   3. Если грамматика относится к классу **SLR(1)**, то можно ли по ней построить нисходящий синтаксический акцептор?
   4. Перечислите все знаки операций восходящего синтаксического акцептора, объясните назначение их полей (параметров).
   5. Постройте историю работы восходящего синтаксического акцептора при разборе цепочки: *(((x)))*
   6. Что такое конфигурация?
   7. Напишите **LR**-грамматику для условного оператора С-подобного языка, имеющего и полную и сокращенную форму.
   8. Какие грамматики относятся к классу **LALR(1)**?
   9. Что такое конфликт «сдвиг/свертка»?
   10. Как таблица конфигураций преобразуется в управляющую таблицу восходящего синтаксического акцептора?
   11. Что такое ожидаемый правый контекст?
   12. Что делает восходящий автомат при выполнении операции свертки?
   13. Что такое детерминированное (безвозвратное) восстановление дерева грамматического разбора?
   14. Как вычисляется ожидаемый правый контекст конфигурации?
   15. Опишите технологию разработки процедурной реализации восходящего синтаксического акцептора.
   16. Как выполняется операция сдвига?
   17. Что такое **LR(1)**-грамматика?
   18. Что такое конфликт «свертка/свертка»?
   19. Перечислите все операции восходящего синтаксического акцептора.
   20. Чем различаются понятия расширенной и канонической расширенной конфигурации?
   21. Как используются множества последователей нетерминалов для предупреждения конфликтов при построении управляющей таблицы восходящего синтаксического акцептора по **SLR(1)**-грамматике?
   22. Разработайте **LR**-грамматику для оператора переключателя (switch) С-подобного языка.

## Лабораторная/практическая работа № 6

1. Название работы: «Синтаксис языков программирования. Преобразование транслируемой программы в постфиксную форму записи».
2. Цели работы: изучение задач и методов преобразования текста транслируемой программы постфиксную форму записи (ПФЗ) для выявления заложенной в алгоритм последовательности операций, приобретение навыков разработки действий, реализующих преобразования.
3. Основные теоретические сведения:

Постфиксная форма записи

Конечной целью работы транслятора является эквивалентное преобразование текста программы на исходном языке в код, который может исполнять реальная или виртуальная вычислительная машина. Компиляторы формируют машинный код для реального компьютера, интерпретаторы – для виртуальной машины. Требование эквивалентности преобразования означает, что результаты мысленного выполнения исходной программы и результаты физического выполнения преобразованной программы при одинаковых обрабатываемых данных должны быть идентичными. Процессы (истории) выполнения программ, записанных на различных языках, можно рассматривать как последовательности выполнения различных по сложности операций. В исходной программе это могут быть операции уровня вычисления сложного выражения, выполнения итерации цикла, ветки условного оператора или переключателя. В машинном или виртуальном коде это операции уровня сложения двух чисел, сравнения значений, условной или безусловной передачи управления из одной точки программы в другую. При формальных преобразованиях, выполняемых трансляторами, добиться эквивалентности можно только в том случае, если гарантируется, что в любой паре историй работы выполнению каждой операции исходной программы соответствует выполнение эквивалентной ей последовательности из одной или нескольких операций машинного или виртуального кода (далее машинный или виртуальный код будет называться объектным).

Таким образом, транслятор рано или поздно обязан выяснить, какие операции и в какой последовательности должны выполняться согласно алгоритму обработки данных, определенному текстом исходной программы. Здесь очень важным моментом является соотношение между синтаксисом (определяющим способ записи последовательности операций) и семантикой (определяющей способ выполнения этой последовательности) для языков разного уровня. Любой язык программирования высокого уровня ориентирован на предоставление максимальных удобств разработчику программ. Поэтому, как правило, последовательность появления знаков операций в тексте исходной программы не совпадает с последовательностью их выполнения в истории работы программы.

Приведем простейший пример. Пусть в тексте программы на языке С/С++ записан оператор присваивания

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

Последовательность появления знаков операций в тексте такова: = \* +. Однако выполнение этого оператора, в целом определяемое семантикой языка, эквивалентно последовательности выполнения трех элементарных операций, эквивалентных машинным командам:

1. умножить значение *b* на значение *c* (операция \*);
2. сложить полученное значение со значением *d* (операция +);
3. присвоить последнее полученное значение переменной *a* (операция =).

Следовательно, в объектном коде знаки операций должны быть записаны в последовательности \* + =, поскольку семантика машинно-ориентированных языков предусматривает последовательную выборку выполняемых команд из линейно организованной памяти. Легко можно привести множество других примеров, из которых следует, что привычная для человека форма записи выражений, операторов присваивания и других операторов языков программирования существенно отличается от того вида, в котором они должны быть представлены в объектном коде.

Весьма существенные отличия форм представления исходной и объектной программ характерны для управляющих конструкций языков программирования – условных операторов, переключателей и операторов цикла. Синтаксис таких конструкций не предусматривает явной записи операций передач управления, подразумеваемых семантикой языка, и ориентирован на удобство использования человеком. Однако в процессе эквивалентного преобразования исходной программы эти операции, очевидно, должны появиться в тексте объектной программы. Например, пусть в тексте программы на языке С/С++ записан условный оператор

*if ( c>0 )*

*a = b \* c + d;*

*else*

*a = ( d – b ) \* c;*

Смысл этой записи совершенно ясен человеку и сводится к тому, что должна быть выполнена определенная последовательность действий.

1. Вычислить результат сравнения значения *c* с нулем и получить булево значение *true* или *false*.
2. Если результат сравнения есть *false*, то перейти к шагу 7 (к выполнению оператора, записанного внутри ветки *else*).
3. Перемножить значения *b* и *c*.
4. Сложить полученное значение с *d*.
5. Присвоить полученное значение переменной *a*.
6. Перейти к шагу 10.
7. Вычесть значение *b* из значения *d*.
8. Умножить результат вычитания на значение *c*.
9. Присвоить полученное значение переменной *a*.
10. … (Следующий по тексту оператор программы.)

Именно в этой последовательности должны быть записаны операции в тексте объектного кода, для того чтобы процессор компьютера (или виртуальная машина) мог выбирать их из оперативной памяти и выполнять. В этом представлении появились операции переходов (передач управления) на шагах 2 и 6, отсутствующие в явном виде в исходном тексте, но подразумеваемые семантикой входного языка.

Постфиксная форма записи (ПФЗ), эквивалентная исходному оператору и содержащая близкие к машинно-ориентированному языку элементы, может выглядеть так:

*c 0 >* labelF ***JmpF*** *a b c \* d + =* ***labelEnd Jmp labelF:*** *a d b – c \* =* ***labelEnd:***

В этой записи жирным шрифтом выделены слова, добавленные при преобразовании, и подчеркнуты знаки операций, в том числе операции условной (*JmpF*) и безусловной (*Jmp*) передачи управления. Операнды каждой операции записаны перед знаком этой операции. Все знаки операций, кроме унарной безусловной передачи управления, являются бинарными (используют два операнда). Унарная операция *Jmp* имеет единственный операнд – метку *labelEnd*. Метки, именующие некоторые операторы программы и используемые в операциях перехода, введены при преобразовании исходного кода в постфиксную форму записи.

Особо отметим, что в постфиксной записи второго оператора присваивания отсутствуют скобки, изменяющие порядок выполнения операций в исходном операторе программы. Постфиксная форма записи уникальна по следующим причинам:

– последовательность появления в ней знаков операций совпадает с требуемым порядком их выполнения;

– не нужны скобки для изменения порядка выполнения операций.

Задача выявления последовательности операций, эквивалентной исходной программе, хотя и определяется во многом семантикой двух языков, но имеет глубокие внутренние связи с задачей восстановления дерева грамматического разбора и решается, как правило, на этапе синтаксического анализа.

Синтаксические деревья и постфиксная форма записи

Синтаксическим деревом, или деревом операций, называется графическое представление совокупности операций, связанных значениями обрабатываемых данных (операндами), построенное по следующим правилам:

* узлы (вершины дерева, из которых выходят дуги, ведущие к потомкам) помечены знаками операций;
* листья (концевые вершины дерева, не имеющие потомков) помечены наименованиями операндов;
* нет вершин, помеченных какими-либо другими символами.

Синтаксическое дерево оператора присваивания *a=b\*c+d;* может выглядеть так, как показано на рис. 16, *а*. На рис. 16, *б* для сравнения показано дерево грамматического разбора этого оператора в грамматике ***Ga*1**, расширенной путем добавления правила *P* : *i = S ;* для нового начального нетерминала *P*.

Синтаксическое дерево (рис. 16, *а*) в наглядной форме показывает зависимость операций друг от друга и может быть использовано для определения последовательности их выполнения. Ясно, что до тех пор, пока не вычислено произведение значений *b* и *c*, не может быть выполнена операция сложения.



*а б*

*Рис. 16.* Связь дерева операций и дерева разбора:

*а* – дерево операций; *б* – дерево грамматического разбора

В свою очередь, операция присваивания зависит от результата выполнения операции сложения и может быть выполнена только после нее. Может быть определена точная процедура обхода синтаксического дерева для построения требуемой последовательности операций в линейном представлении.

Дерево грамматического разбора, восстанавливаемое при проверке правильности данного оператора присваивания и показанное на рис. 16, *б*, также содержит всю необходимую информацию для решения поставленной задачи. Однако это дерево содержит «лишние» с точки зрения выявления последовательности операций элементы: вершины, помеченные нетерминалами и выходящими их них дугами, а также вершину, помеченную ограничителем оператора присваивания (*;*) вместе со всеми дугами, ведущими к таким вершинам. В данном операторе не использовались скобки *()*, но если бы они и были, то также считались бы «лишними».

Дерево грамматического разбора может быть преобразовано в дерево операций путем применения следующей процедуры.

**Шаг 1.** Удалить все листья (вершины, помеченные терминальными символами), пометки которых не являются знаками операций и наименованиями операндов.

**Шаг 2.** Просмотреть узлы дерева (вершины, имеющие исходящие дуги), начиная с корня. Для каждого просматриваемого узла сохранять в стеке перечень дочерних узлов. Если какая-либо из дочерних вершин помечена знаком операции, то просматриваемый узел пометить этим знаком и удалить из дерева дочернюю вершину, но только в том случае, если она является листом. Если после обработки очередного узла стек не пуст, то перейти к обработке узла, номер которого снимается с вершины стека. Если же стек опустел, то повторять шаг 2 до тех пор, пока состояние дерева не перестанет изменяться.

**Шаг 3.** Для каждого листа, помеченного операндом, проверить пометку родительского узла. В том случае, если родительский узел помечен нетерминалом, перенести в него наименование операнда из дочернего листа и удалить этот лист. Продолжать выполнение шага 3 до тех пор, пока состояние дерева не перестанет изменяться.

Если применить эту процедуру к приведенному выше дереву разбора, то будет получено именно такое дерево операций, которое изображено на рис. 16, *а*.

Для данной грамматики применение этой процедуры позволит получить желаемый результат из дерева разбора любого оператора присваивания. Объясняется это тем, что вся семантика, определяющая последовательность выполнения операций сложения, вычитания и присваивания, а также изменение порядка их выполнения при использовании скобок в любом выражении неявным образом заложена в совокупность порождающих правил, описывающих синтаксис языка операторов присваивания.

Из дерева операций легко можно получить постфиксную форму записи линейной последовательности знаков операций (с их операндами) – такую, в которой порядок появления операций совпадает с требуемым порядком их выполнения.

Процедура преобразования дерева операций в постфиксную форму записи является рекурсивной и может быть определена следующим образом.

**Шаг 1.** Взять корень дерева операций в качестве текущей вершины.

**Шаг 2.** Если текущая вершина не является листом, то перейти к шагу 3, иначе выдать ее пометку (наименование операнда) на выход и завершить обход поддерева.

**Шаг 3.** Обойти левое поддерево данного корня (рекурсивно вызвать шаг 2 процедуры для корня левого поддерева текущей вершины).

**Шаг 4.** Обойти правое поддерево данного корня (рекурсивно вызвать шаг 2 процедуры для корня правого поддерева текущей вершины).

**Шаг 5.** Выдать пометку текущей вершины (знак операции) на выход. Завершить обход поддерева.

Применение этой процедуры к построенному нами дереву операций для оператора присваивания *a=b\*c+d;* позволит получить постфиксную запись

*a b c \* d + =*

Ее смысл (семантика) состоит в следующем.

Сначала должна быть выполнена операция умножения значений *b* и *c*, наименования которых записаны ***перед*** знаком \*. Можно считать, что в результате выполнения операции умножения получено промежуточное значение, которое мы обозначим через *r*, а исходная ПФЗ превратилась в такую:

*a r d + =*

Затем аналогичным образом должна быть выполнена операция сложения значений *r* и *d* (результат сложения обозначим так же: *r*), после чего запись оператора будет выглядеть так:

*a r =*

Окончательно после выполнения операции присваивания запись оператора получит вид *r*,где под *r* можно понимать (как это делается в языке С/С++) результат выполнения всего оператора присваивания.

Главной особенностью постфиксной записи по сравнению с привычной для человека смесью инфиксной (знак операции находится между наименованиями операндов, например, *a+b*) и префиксной (знак операции находится перед наименованиями своих операндов, например, *sin(x)*) форм записи является то, что порядок следования знаков операций в записи строго совпадает с требуемым порядком их выполнения при полном отсутствии необходимости в скобках, меняющих этот порядок.

Таким образом, если построено дерево разбора правильного предложения, то известен и метод решения основной задачи синтаксического анализа: преобразование предложения в такую промежуточную форму записи, в которой положение знаков операций совпадает с требуемым порядком их выполнения.

Однако отметим следующее:

* в явном виде дерево разбора не строится никаким синтаксическим акцептором (во всяком случае теми, которые были изучены в работах [3–5]). Следовательно, прямое применение вышеописанных процедур невозможно или требуется их модификация;
* в простой грамматике ***Ga*1**нет правил, содержащих в правой части более одного знака операции (на чем и основана данная процедура). При наличии таких правил придется существенно усложнять шаг 2 процедуры преобразования дерева разбора в дерево операций;
* для реальных языков программирования не всегда возможно построить такую грамматику, чтобы все семантические правила, определяющие требуемую последовательность (следующую из приоритетов) выполнения операций, были использованы в синтаксических порождающих правилах. В подобных случаях дерево разбора невозможно преобразовать в дерево операций с помощью вышеописанной процедуры.

Включение действий в грамматику для преобразования предложения в постфиксную форму записи

Для арифметических выражений и операторов присваивания существуют следующие достаточно простые виды определения постфиксной формы записи.

1. Постфиксной записью пустой цепочки символов ε является ε.
2. Постфиксной записью идентификатора *i* является идентификатор *i*.
3. Постфиксной записью константы *c* является константа *c*.
4. Если *E* – произвольное выражение, то постфиксной записью выражения, взятого в скобки, т. е. *( E )* будет просто ПФЗ(*E*).
5. Если *E* – выражение вида 2, 3 или 4, то постфиксной записью выражения –*E* (изменение знака значения *E*) будет ПФЗ(*E*)– .
6. Если *E* – выражение вида 2, то постфиксной записью выражений   
   ++*E* (– –*E* ) является

ПФЗ(*++E*) = ПФЗ( *incPre(&E)* ) = *E & incPre* (ПФЗ(*– –E*) =   
= ПФЗ( *decPre(&E)* ) = *E & decPre*),

где & – знак операции взятия адреса; *incPre* (*decPre*) – имя функции (знак операции), увеличивающей (уменьшающей) значение своего аргумента на единицу и возвращающей измененное значение.

1. Если *E* – выражение вида 2, то постфиксной записью выражений *E++* (*E* – –) является

ПФЗ(*E++*) = ПФЗ( *incPost(&E)* ) = *E & incPost (*ПФЗ(*E– –*) =   
= ПФЗ( *decPost(&E)* ) = *E & decPost*),

где & – знак операции взятия адреса; *incPost* (*decPost*) – имя функции (знак операции), увеличивающей (уменьшающей) значение своего аргумента на единицу и возвращающей еще не измененное значение.

1. Если *E*1, *E*2, …, *Ek* – выражения вида 2…9, то постфиксной записью выражения ○ *(E*1, *E*2, …, *Ek)* является ПФЗ(*E*1) ПФЗ(*E*2) ... ПФЗ(*Ek*) ○, где ○ – знак любой *k*-арной операции (функции с *k* аргументами); ПФЗ(*E*1), ПФЗ(*E*2) … ПФЗ(*Ek*) – постфиксные записи выражений *E*1, *E*2, …, *Ek* соответственно.
2. Если *E*1 и *E*2 – выражения вида 2…7, то постфиксной записью выражения *E*1 ○ *E*2 является ПФЗ(*E*1) ПФЗ(*E*2) ○, где ○ – знак любой бинарной операции (присваивания, сложения, вычитания, умножения и др.), а ПФЗ(*E*1) и ПФЗ(*E*2) – постфиксные записи выражений *E*1 и *E*2 соответственно.
3. Если *E*1, *E*2, *E*3 – выражения вида 2…8, то постфиксной записью выражения*Е*1 ○ *E*2 ● *E*3 (где ○ и ● – знаки бинарных операций) является ПФЗ(*E*1) ПФЗ(*E*2) ○ ПФЗ(*E*3) ●, если приоритет знака операции ○ не меньше приоритета знака операции ●, и ПФЗ(*E*1) ПФЗ(*E*2) ПФЗ(*E*3) ● ○ – в противном случае.
4. Если *E*1*–* выражение вида 2, а *E*2– выражение вида 2…9, то постфиксной записью выражения *E*1○*E*2*;* является ПФЗ(*E*1) ПФЗ(*E*2) ○, где ○ – знак операции присваивания (напомним, что в языке С/С++, например, существует не один, а несколько знаков операции присваивания: =, +=, \*=, …).

Подобные определения можно сформулировать и для других синтаксических конструкций языка программирования. Руководствуясь этими определениями, достаточно просто осуществить преобразование заданной грамматики в грамматику действий таким образом, чтобы построенный на ее основе автомат обеспечивал построение постфиксной формы записи входной цепочки терминальных символов в процессе проверки ее правильности.

Пусть дана грамматика операторов присваивания (расширенная грамматика ***Ga*1**). Будем считать, что имеется функция, выполняющая добавление одного терминального символа (токена) к формируемому в процессе синтаксического анализа промежуточному представлению программы (постфиксной записи), прототип которой выглядит так:

*void PutToPFR(Lexem);*

В правила грамматики будем включать действия (фрагменты программного кода), заключаемые в фигурные скобки. Грамматика операторов присваивания, расширенная действиями по формированию постфиксной записи, может выглядеть следующим образом:

0. *Z* : *P* ►

1. *P* : *{ PutToPFR(CurrentLexem); } i = S { PutToPFR(“=”); } ;*

2. *S* : *S + T { PutToPFR(“+”); }*

3. *S* : *T*

4. *T* : *T \* V { PutToPFR(“\*“); }*

5. *T* : *V*

6. *V* : *( S )*

7. *V* : *{ PutToPFR(CurrentLexem); } i*

8. *V* : *{ PutToPFR(CurrentLexem); } c*

Действия включаются пакетом «ВебТрансБилдер» в код парсера таким образом, чтобы они выполнялись по ходу движения парсера согласно правилам грамматики (нужно помнить, что любой парсер в процессе восстановления дерева разбора движется по правым частям правил слева направо).

В правило 1 добавлены два действия.

Первое действие (*{PutToPFR(CurrentLexem);}*) обеспечивает преобразование идентификатора, находящегося в левой части оператора присваивания, в постфиксную форму (в соответствии с определением ПФЗ для идентификатора). Действие вставлено в правило до терминального символа *i*, потому что при функционировании любого (нисходящего или восходящего) автомата оно должно быть выполнено в тот момент, когда текущим входным символом (значением переменной *CurrentLexem*) является идентификатор из левой части оператора присваивания.

При переходе нисходящего акцептора в состояние, соответствующее знаку оператора присваивания, а восходящего – в состояние, соответствующее точке между терминалами *i* и *=* в этом правиле, текущим входным символом уже будет слово =. Поэтому для помещения лексемы, прочитанной из входной цепочки, в постфиксную запись вызов функции *PutToPFR* должен помещаться в правило непосредственно перед терминалом, обозначающим группу слов типа идентификаторов (или констант). Аналогичные действия в тех же точках можно увидеть в правилах 7 и 8. Здесь приведено обоснование точек вставки подобных действий как для нисходящего, так и для восходящего методов синтаксического акцепта, несмотря на то что рассматриваемая грамматика годится только для восходящего.

Второе действие в правиле 1 (*{ PutToPFR(“=”); }*) находится между нетерминалом *S* и ограничителем оператора присваивания *;*. Для простоты будем предполагать, что функция *PutToPFR* способна преобразовать строку в лексему. Точка вставки этого действия строго соответствует определению постфиксной записи выражений, образованных с помощью бинарного знака операции присваивания. При этом ПФЗ выражения *E*1 формируется первым действием в данном правиле, а ПФЗ выражения *E*2 будет сформировано при разборе той части входной цепочки, которая выводится из нетерминала *S*.   
В этом процессе будут использоваться правила для этого и других нетерминалов и выполняться вставленные в них действия. В тот момент, когда вся цепочка символов, выводимая из *S* и представляющая собой правую часть оператора присваивания, будет обработана и текущим входным символом станет ограничитель *;*, автомат выполнит второе действие и завершит тем самым формирование постфиксной записи оператора присваивания в целом.

Действия, вставленные в правые части правила 2 (*{ PutToPFR(“+”); }*) и правила 4 (*{ PutToPFR(“+”); }*), точно так же обусловлены определением постфиксной записи для выражений, образуемых с использованием бинарных знаков операций. Точки вставки этих действий тоже обусловлены этим определением и для данной грамматики не могут быть изменены. Действия, вставленные в правила 7 и 8, уже описаны.

При построении конечного автомата на основе грамматики действий должно быть обеспечено выполнение этих действий в требуемые моменты времени. Любой преобразователь грамматик в синтаксические анализаторы делает это либо путем добавления специальных полей в управляющую таблицу автомата, либо путем формирования дополнительных структур с указателями на функции, в которые преобразуются действия.

Легко видеть, что расширенный таким образом **LALR(1)**-автомат, который можно построить по данной грамматике действий, обеспечит преобразование любого правильного оператора присваивания в постфиксную форму записи без построения в явном виде дерева грамматического разбора и преобразования его в дерево операций с последующим обходом последнего.

Для многих синтаксических конструкций языков программирования преобразование в постфиксную запись требует значительно больших усилий. Это объясняется необходимостью формирования уникальных меток и операций передачи управления при выявлении линейной последовательности операций для операторов цикла, условных операторов и переключателей.

Преобразование управляющих конструкций языка программирования в постфиксную форму записи

Такое преобразование, как правило, связано с необходимостью решения ряда дополнительных задач. Рассмотрим источники их возникновения и один из возможных методов решения на простом примере оператора цикла *while* языка программирования С/С++.

Допустим, что синтаксис оператора *while* определен в грамматике следующим образом:

*Operator : while ( Expression ) Block*

Предполагается, что определены и другие операторы (присваивания, условный и другие, в том числе оператор *break*, семантика которого состоит в выходе из тела цикла), что *Expression* определено как выражение, значение которого можно преобразовать в логическое значение, и что *Block* определен как одиночный оператор либо как последовательность операторов, заключенная в фигурные скобки.

Запишем желаемый вид постфиксной записи для оператора цикла *while*, используя введенные определения ПФЗ для *Expression* и предполагая, что способ формирования ПФЗ для *Block* также известен (на самом деле он индуктивно зависит от способа формирования ПФЗ оператора *while*, поскольку этот блок может содержать один или несколько операторов *while*):

ПФЗ( *while ( Expression ) Block* ) =

***Label1:*** ПФЗ( *Expression* ) ***Label2 JmpF***ПФЗ( *Block* ) ***Label1 Jmp Label2:***

В этом определении жирным курсивом выделены следующие обозначения:

***Label1:*** – наименование первого элемента постфиксной записи вычисления значения выражения;

***JmpF*** – бинарный знак операции условного перехода, использующий в качестве первого операнда для определения необходимости передачи управления значение, вычисляемое ПФЗ(*Expression*), а в качестве второго – наименование (***Label2***) первого элемента постфиксной записи, следующего непосредственно после ПФЗ данного оператора цикла;

***Jmp*** – унарный знак операции безусловного перехода, использующий в качестве операнда наименование ***Label1***;

***Label2:*** – определение наименования для оператора выхода из цикла (условный переход ***JmpF***).

Согласно этому определению выполнение оператора *while* должно протекать следующим образом:

* вычисляется значение выражения;
* если оно имеет значение *true*, то оператор *JmpF* не передает управление на оператор, помеченный именем *Label2:*, выполняются действия постфиксной записи блока и оператором *Jmp* управление возвращается на начало вычисления выражения (операцию, помеченную *Label1:*);
* если же значение выражения есть *false*, то оператор *JmpF* передает управление, используя *Label2* в качестве адреса.

Если бы требовалось преобразовать в постфиксную запись единственный оператор цикла, то такого определения его ПФЗ было бы достаточно. Однако в тексте программы может встретиться несколько операторов *while*, в том числе и внутри блока, являющегося телом данного цикла.

Для того чтобы при преобразовании в постфиксную запись не формировались идентичные наименования для разных точек переходов (это создаст проблемы при генерации объектного кода), необходимо в приведенном определении ПФЗ все метки понимать как уникальные, однозначно сопоставленные с конкретным оператором цикла. Уникальность меток можно обеспечить при реализации преобразования в постфиксную запись, т. е. при вставке действий в грамматику.

Для обеспечения уникальности меток, создаваемых для машинно-ориентированного эквивалента оператора *while*, можно сделать следующее:

* используя специально определенную для этой цели переменную (счетчик), присваивать уникальный номер каждому оператору цикла, встретившемуся во входной программе при восстановлении дерева ее разбора;
* при увеличении значения счетчика сохранять это значение во вспомогательном стеке, доступном из действий, вставляемых в грамматику;
* использовать значение, находящееся на вершине вспомогательного стека, для формирования наименований адресов перехода;
* удалять верхнее значение из стека в момент завершения обработки оператора цикла.

Приведем пример реализации, предполагая, что счетчик имеет целое значение и называется *WhileCount*, а для операций над вспомогательным стеком имеются функции с прототипами:

*void Push(int);*  – поместить значение аргумента на вершину стека;

*int Pop(void);* – возвратить значение, удалив его с вершины стека;

*int Top(void);* – вернуть значение с вершины, не удаляя его из стека.

Для краткости записи будем считать, что при написании действий можно использовать бинарную инфиксную операцию *#*, преобразующую свои операнды в строковые представления и возвращающую конкатенацию этих строк. Функция *PutToPFR* будет преобразовывать такие строковые аргументы в лексемы. Теперь рассматриваемое правило грамматики, расширенное действиями на основе определения ПФЗ оператора цикла *while*, будет выглядеть так:

*Operator* : *while*

*{Push(++WhileCount); PutToPFR(“Label1” # Top() # “:”);}*

*( Expression )*

*{ PutToPFR(“Label2” # Top()); PutToPFR(“JmpF”);}*

*Block*

*{ PutToPFR(“Label1” # Top()); PutToPFR(“Jmp”);*

*PutToPFR(“Label2” # Pop() # “:”); }*

Кроме операторов цикла *while* язык программирования, как правило, содержит и другие формы операторов цикла (*for*, *do*, …), условные операторы, переключатели и др.

Все метки, генерируемые при формировании постфиксной формы записи этих конструкций, должны быть уникальны в пределах одной транслируемой программной единицы. Поэтому разработчику транслятора придется решать следующие вопросы:

– требуется ли поддерживать отдельные счетчики для разных типов управляющих конструкций или для всех таких операторов использовать один счетчик;

– поддерживать ли несколько вспомогательных стеков или можно обойтись одним.

1. Порядок выполнения работы(рекомендуется использовать в качестве примера систему правил Samples/Sample6):
   1. Реализовать полную грамматику заданного языка.
   2. Используя пакет «ВебТрансБилдер», изучить и освоить следующее:
      * способы включения структур данных и операций в грамматику, полученную в результате выполнения предыдущих работ;
      * структуру и механизмы функционирования нисходящего и восходящего синтаксических анализаторов, способы вызова и моменты активизации действий, расширяющих функциональность синтаксических акцепторов;
      * методику разработки и отладки (трассировки) системы действий, обеспечивающих преобразование в постфиксную запись последовательности лексем, получаемой от лексического анализатора.
   3. Использовать полученные навыки для разработки преобразователя в ПФЗ для выражений, операторов присваивания и по меньшей мере одного управляющего оператора языка, заданного на курсовую работу.
2. Подготовить отчет, который должен содержать следующие разделы:
   * + цель работы;
     + краткое описание свойств постфиксной записи и методов ее формирования;
     + реализацию действий для формирования ПФЗ, включенных в грамматику языка, который задан на курсовую работу;
     + описание структур данных и алгоритмов преобразователя анализируемой программы на заданном языке в ПФЗ в качестве фрагмента расчетно-пояснительной записки;
     + примеры результатов преобразования тестовых программ в ПФЗ;
     + выводы и заключение.
3. Контрольные вопросы
   1. Для чего операторы программы преобразуются в постфиксную форму записи?
   2. Эквивалентны ли абстрактное синтаксическое дерево и постфиксная форма записи программы?
   3. Что такое детерминированное (безвозвратное) восстановление дерева грамматического разбора?
   4. Сформулируйте основные принципы преобразования арифметических выражений в постфиксную запись.
   5. Почему леворекурсивная грамматика не может быть преобразована в нисходящий синтаксический акцептор?
   6. Что такое постфиксная запись?
   7. Каковы свойства постфиксной формы записи?
   8. В чем состоят отличия алгоритма преобразования управляющих конструкций в постфиксную запись от алгоритма преобразования арифметических выражений.

## Лабораторная/практическая работа № 7

1. Название работы: «Семантика языков программирования. Семантический анализ и генерация псевдокода».
2. Цели работы: изучение семантических свойств объектов транслируемой программы, методов их выявления и использования, типов данных и методов контроля типов, областей видимости переменных, локальных и нелокальных сред ссылок, способов передачи параметров, приобретение навыков преобразования ПФЗ в псевдокод.
3. Основные теоретические сведения:

Задачи семантического анализа

Совокупность исходных данных для семантического анализа формируется предыдущими этапами процесса трансляции:

1. постфиксная форма записи (ПФЗ) или ее эквивалент, синтаксическое дерево транслируемой программы, – это промежуточное представление, которое образуется в процессе синтаксического анализа;
2. набор информационных таблиц, в которых накоплены сведения об используемых программой данных.

Постфиксная форма записи обладает важным свойством: в ней порядок следования знаков операций строго совпадает с предусматриваемой алгоритмом решения задачи последовательностью их выполнения (в отличие от исходного текста программы, где этого обычно и не требуется).

Реальное исполнение выявленной транслятором последовательности операций с целью преобразования исходных данных транслируемой программы в ее результаты возможно различными способами, все множество которых принято делить на две основные группы: компиляция и интерпретация.

***Компиляция*** – это продолжение преобразований представления программы на стадии трансляции, завершающееся формированием и сохранением так называемого объектного (целевого) кода, который впоследствии может многократно исполняться процессором компьютера (реальной машиной) уже без какого бы то ни было участия транслятора.

***Интерпретация*** – это исполнение последовательности операций непосредственно транслятором (прямая интерпретация) или специально разработанной для этого программой (так называемой виртуальной машиной – отложенная интерпретация), возможно, с предварительным преобразованием постфиксной записи в более удобную промежуточную или сохраняемую форму.

Четкой границы между компиляцией и интерпретацией не существует. На практике, как правило, реализуются смешанные варианты последовательности действий трансляции / исполнения. Например, в скомпилированных программах обычно содержатся вызовы функций из библиотек так называемого runtime-окружения (поддержки), которые, по существу, интерпретируют операции, являющиеся примитивными в языке программирования, но не реализуемые аппаратурой компьютера на уровне машинных команд. Для обеспечения исполнения подобных операций транслятор вставляет в компилируемый им объектный код команды вызова таких интерпретирующих функций.

В свою очередь, многие системы программирования (такие как Visual Basic), которые первоначально были ориентированы на прямую интерпретацию, впоследствии приобрели ряд признаков компиляции, выполняя перевод текста программы в псевдокод (или байт-код). Псевдокод сохраняется во внешней памяти и может многократно интерпретироваться виртуальной машиной уже без использования таких частей транслятора, как лексический и синтаксический анализатор.

Для того чтобы исполнять программу в целом (после компиляции или при отложенной интерпретации) или даже по частям (при прямой интерпретации), необходимо предварительно убедиться в том, что каждая исполняемая операция действительно может быть применена к значениям своих операндов и способна сформировать требуемый результат. Именно это является основной целью семантического анализа.

Таким образом, в логической последовательности этапов процесса трансляции семантический анализ занимает позицию непосредственно после синтаксического и перед генерацией объектного кода в случае компиляции либо перед исполнением вычислений в случае интерпретации. Реальная последовательность выполнения этапов при трансляции не обязательно совпадает с логической последовательностью.

Семантическая проверка некоторого фрагмента программы может предшествовать синтаксическому анализу последующего по тексту фрагмента и (или) осуществляться после интерпретации текстуально предыдущего фрагмента (или его преобразования в объектный код).

Однако применительно к каждому конкретному синтаксически целостному фрагменту текста (модулю / файлу, подпрограмме / функции, блоку, оператору и др.) логическая последовательность этапов трансляции всегда строго соблюдается. Преобразуются в объектный код при компиляции или исполняются при интерпретации только те фрагменты текста, для которых все этапы анализа (лексический, синтаксический и семантический) завершились успешно.

Семантический анализатор транслятора в процессе проверки правильности транслируемой программы осуществляет преобразование ее постфиксной формы записи (или эквивалента ПФЗ – дерева операций), формируемой синтаксическим анализатором, в псевдокод, как показано на рис. 17.



*Рис. 17.* Структура семантического анализатора

Например, пусть транслятор с языка С/С++ обрабатывает функцию вычисления наибольшего общего делителя двух аргументов:

*int nod( int first, int second ){*

*while ( first != second )*

*if ( first < second )*

*second –= first;*

*else*

*first –= second;*

*return first;*

*}*

Постфиксная запись этой функции, построенная синтаксическим анализатором, может выглядеть так:

*nod int functionActivate first int getArgument second int getArgument label#0#0: first second != label#0#1 jmpOnFalse first second* < *label#1#0 jmpOnFalse second first –= label#1#1 jmp label#1#0: first second –= label#1#1: label#0#0 jmp label#0#1: first return*

Здесь:

*nod int functionActivate first int getArgument second int getArgument* – ПФЗ заголовка функции;

*label#0#0: first second != label#0#1 jmpOnFalse* – ПФЗ заголовка оператора цикла;

*first second < label#1#0 jmpOnFalse second first –= label#1#1 jmp label#1#0: first second –= label#1#1:* – ПФЗ условного оператора, составляющего тело цикла;

*label#0#0 jmp* – ПФЗ завершения оператора цикла;

*label#0#1: first return* – ПФЗ оператора возврата значения функции.

В этой форме записи для наглядности подчеркнуты все знаки операций. Большинство знаков операций (*functionActivate, getArgument, !=, jmpOnFalse,* <, *–=*) являются бинарными, но есть и унарные знаки (*jmp* и *return*).

Требуемая последовательность выполнения операций программы выявлена и зафиксирована в постфиксной форме записи, что позволит впоследствии построить последовательность машинных команд, которые процессор будет выбирать из рядом расположенных ячеек памяти. Однако с этой формой записи могут быть связаны и другие проблемы. Рассмотрим, например, фрагмент ПФЗ заголовка цикла

*label#0#0: first second != label#0#1 jmpOnFalse*

У бинарного знака операции *jmpOnFalse* в этом фрагменте первым операндом является другой бинарный знак операции: *!=*. В действительности это означает, что первым операндом операции *jmpOnFalse* является булево значение, вырабатываемое операцией сравнения. Это значение должно быть сохранено либо в стеке времени выполнения, либо в переменной, формируемой транслятором.

Форматы псевдокода

Для того чтобы выявить все указанные выше переменные и выполнить соответствующие семантические проверки, транслятором формируется очередное внутреннее представление текста программы, называемое псевдокодом.

Псевдокод – это последовательность операций в системе команд некоторой виртуальной машины. Эта машина может быть, например, трехадресной, в этом случае каждая ее команда включает в себя четыре поля (и называется тетрадой): код операции, наименования двух операндов и наименование результата. Фрагмент псевдокода тела функции вычисления наибольшего общего делителя для такой виртуальной машины может выглядеть так, как показано в табл. 16.

Таблица 16

|  |  |  |  |
| --- | --- | --- | --- |
| Код операции  **Нет в УП** | Первый операнд | Второй операнд | Результат |
| *defineLabel* | *label#0#0* |  |  |
| *!=* | *second* | *first* | *push* |
| *jmpOnFalse* | *label#0#1* | *pop* |  |
| *<* | *second* | *first* | *push* |
| *jmpOnFalse* | *label#1#0* | *pop* |  |
| *–=* | *first* | *second* | *second* |
| *jmp* | *label#1#1* |  |  |
| *defineLabel* | *label#1#0* |  |  |
| *–=* | *second* | *first* | *first* |
| *defineLabel* | *label#1#1* |  |  |
| *jmp* | *label#0#0* |  |  |
| *defineLabel* | *label#0#1* |  |  |
| *return* | *first* |  |  |

Виртуальная машина, псевдокод которой показан в табл. 16, является стековой. Результаты некоторых операций заносятся в стек с помощью указания имени вершины стека *push* и извлекаются из стека в последующих тетрадах с использованием другого имени вершины стека *pop*.

Недостатком тетрад является то, что для объявления имени (метки) какой-либо операции приходится добавлять специальную тетраду с кодом операции «Создать метку». Этого недостатка лишены виртуальные машины, у которых каждая операция имеет дополнительное поле метки. Команды такой виртуальной машины называют пентадами (по количеству полей).

В табл. 17 показан псевдокод той же функции в виде последовательности пентад. Количество операций уменьшилось за счет удаления операций «Создать метку». В этом варианте виртуальной машины для хранения промежуточных результатов вычислений вместо стека используются временные переменные, создаваемые семантическим анализатором транслятора.

Таблица 17

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метка | Код  **Нет в УП**  операции | Первый  операнд | Второй операнд | Результат |
| *label#0#0* | *!=* | *second* | *first* | *tmpVar1* |
|  | *jmpOnFalse* | *label#0#1* | *tmpVar1* |  |
|  | *<* | *second* | *first* | *tmpVar2* |
|  | *jmpOnFalse* | *label#1#0* | *tmpVar2* |  |
|  | *–=* | *first* | *Second* | *second* |
|  | *jmp* | *label#1#1* |  |  |
| *label#1#0* | *–=* | *second* | *First* | *first* |
| *label#1#1* | *jmp* | *label#0#0* |  |  |
| *label#0#1* | *return* | *first* |  |  |

Возможны и другие варианты организации виртуальных машин.

Пример псевдокода той же функции для виртуальной машины, операции которой (триады) не имеют ни поля метки, ни поля наименования результата, показан в табл. 18. В этом примере вместо имен операций и промежуточных результатов используется относительная адресация.

Таблица 18

|  |  |  |
| --- | --- | --- |
| Код операции  **Нет в УП** | Первый операнд | Второй операнд |
| *!=* | *second* | *first* |
| *jmpOnFalse* | *+8* | *–1* |
| *<* | *second* | *first* |
| *jmpOnFalse* | *+4* | *–1* |
| *–=* | *first* | *second* |
| *=* | *–1* | *second* |
| *jmp* | *+3* |  |
| *–=* | *second* | *first* |
| *=* | *–1* | *first* |
| *jmp* | *–9* |  |
| *return* | *first* |  |

Числа вида *+8* или *–1* в полях наименований операндов означают указание триады, находящейся на расстоянии восьми триад вперед или одной триады назад по отношению к текущей триаде. Если относительный номер триады используется в операции преобразования данных, то под ним понимается результат, вырабатываемый адресуемой триадой.

Все три вида промежуточного представления программы – тетрады, триады и пентады – примерно одинаковым образом могут быть получены из постфиксной записи. Рассмотрим один из алгоритмов такого преобразования.

Преобразование постфиксной записи в псевдокод

Это преобразование рассматривается для формата тетрад, но описанный алгоритм легко можно адаптировать для других форматов псевдокода.

**Шаг 1.** Подготовка. Очистка стека, установка первого слова постфиксной записи в качестве текущего.

**Шаг 2.** Выявление назначения текущего слова ПФЗ. Если это наименование операнда, то переход к шагу 3, иначе (если это знак операции) – к шагу 4.

**Шаг 3.** Текущее слово заносится в стек и удаляется с входа (текущим становится следующее слово ПФЗ). Возврат к шагу 2.

**Шаг 4.** Знак операции удаляется с входа и заносится в тетраду. Определяется арность этого знака операции, а далее формируются наименования операндов тетрады согласно выявленному случаю.

*Случай* 0 – оба наименования операндов устанавливаются равными *null*.

*Случай* 1 – с вершины стека снимается одно наименование и заносится в тетраду в качестве первого операнда, а в качестве второго операнда устанавливается *null*.

*Случай* 2 – с вершины стека снимаются два наименования и заносятся (в порядке извлечения из стека) в тетраду в качестве операндов.

*Случай* *k*> 2 – с вершины стека снимаются *k* слов, для каждого строится вспомогательная тетрада со знаком операции присваивания, снятым со стека наименованием в качестве первого операнда, значением *null* в качестве второго операнда и наименованием формального аргумента процедуры / функции в качестве результата. Каждая вспомогательная тетрада выдается на выход. После этого формируется тетрада с наименованием процедуры / функции в качестве знака операции и пустыми (*null*) наименованиями операндов.

Если для знака операции последней сформированной тетрады подразумевается образование промежуточного результата вычислений, то создается, заносится в таблицу идентификаторов и записывается в тетраду наименование результата, иначе в качестве наименования операнда используется последний снятый со стека операнд. Построенная тетрада выдается на выход.

**Шаг 5.** Если был достигнут конец входной ПФЗ, то выполняется останов процесса преобразования, иначе – переход к шагу 2. Конец алгоритма.

Отметим, что этим алгоритмом не предусматривается никаких проверок корректности входной постфиксной записи. ПФЗ формально является корректной, если для любого знака операции на шаге 4 из стека удается извлечь соответствующее его арности количество наименований операндов и если в момент завершения преобразования стек оказывается пустым.

Предполагается, что ПФЗ строится в процессе синтаксического анализа и является правильной, если транслируемая программа не содержит синтаксических ошибок. В случае если это предположение несправедливо, легко можно дополнить данный алгоритм соответствующими проверками состояния стека.

Кроме того, заметим, что для случая *k*-арного знака операции при   
*k*> 2 возможна модификация соответствующего случая шага 4 алгоритма, согласно которой строится *k –*2 вспомогательных тетрады, а два последних извлекаемых из стека слова используются в качестве наименований операндов последней формируемой тетрады. В рассматриваемом ниже примере это привело бы к исчезновению тетрад 3 и 4 и замене обозначений *null* на *x* в поле *операнд1* тетрады и на *y* – в поле *операнд2*.

Теперь, когда определено внутреннее представление транслируемой программы в виде псевдокода, основную функцию семантического анализа можно сформулировать так: при построении каждой операции виртуальной машины требуется выполнить следующее:

1. определить тип данных первого операнда;
2. определить тип данных второго операнда;
3. проверить, применим ли код операции к данным этих типов;
4. сформировать и запомнить тип данных результата операции.

Все эти задачи можно решать и непосредственно в процессе формирования операций псевдокода.

Аналогичным образом можно преобразовывать ПФЗ в пентады и различные виды триад.

Теперь о вызовах функций, передаче значений аргументов, входе в функции, получении аргументов, подготовке возвращаемого результата, возврате в точку вызова и получении результата для использования. Все это нужно реализовывать, используя стек времени выполнения, иначе невозможно будет обеспечить рекурсивные вызовы функций.

Вызов функции нужно преобразовывать в:

1. Формирование операций занесения значений всех фактических аргументов в стек (нужно создать ровно столько операций, сколько аргументов должна получать функция).
2. Формирование операции занесения в стек адреса возврата. Для этого сразу после операции передачи управления в точку входа в функцию (см. следующий пункт) Вы должны сформировать уникальную метку и именно эту метку в качестве аргумента положить в стек.
3. Формирование операции безусловной передачи управления на метку имени функции.

Пусть, например, формат псевдокода выглядит так: триады Код Оп Оп. Пусть функция a1sq определена в программе так:

a1sq( a1arg, a2arg, a3arg ) { … }

Для примера вызова a1a := a1sq(n1n, x5y, 2) сформированная последовательность операций может выглядеть примерно так:

push 2 –

push x5y –

push n2n –

push retAddr1 –

jmp a1sq –

deflabel retAddr1 –

Соответственно, в точке входа в функцию a1sq нужно сформировать:

1. Операцию снятия с верхушки стека адреса возврата и запись его в локальную переменную с именем, которое нельзя написать в программе на языке, для которого разрабатывается транслятор.
2. Последовательность операций извлечения из стека и записи в переменные, перечисленные как формальные аргументы, значений фактических аргументов.

Для данного примера это:

pop addrForRet –

pop a1arg ­–

pop a2arg ­–

pop a3arg ­–

Далее формируется последовательность операций тела функции.

В точке возврата из функции нужно:

1. Сформировать операцию занесения на верхушку стека вычисленного возвращаемого значения.
2. Сформировать операцию безусловной передачи управления по имени, лежащему в переменной адреса возврата (косвенной передачи управления).

Для данного примера это:

push r0value –

jmp (addrForRet) –

И, наконец, в точке, находящейся после вызова функции (после операции deflabel retAddr1 –), нужно снять возвращенный результат с верхушки стека для использования в соответствующей операции. Выглядеть это может примерно так:

< pop 100

Таким образом может обеспечиваться взаимная связь вызываемой и вызывающей функции по передаче/возврату значений и по передаче/возврату управления

1. Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample7):
   1. Расширить ранее разработанную грамматику путем включения в нее действий для выполнения статических семантических проверок, соответствующих заданному варианту курсовой работы.
   2. Реализовать преобразование постфиксной записи транслируемой программы в псевдокод в формате, заданном вариантом курсовой работы.
   3. Построить компилятор с учебного языка на псевдокод, убедиться в его работоспособности на тестовых примерах программ на учебном языке.
2. Подготовить отчет, который должен содержать следующие разделы:
   * + цель паботы;
     + краткое изложение задач семантического анализа;
     + описание структур данных и алгоритмов совокупности действий, разработанных для реализации семантического анализатора по заданному варианту языка;
     + логически связанное описание фрагментов ПФЗ и полученных из нее фрагментов псевдокода;
     + результаты тестирования разработанного транслятора;
     + выводы и заключение.
3. Контрольные вопросы
   1. В чем состоят функции контроля структуры программы?
   2. Перечислите известные Вам способы представления типов данных.
   3. Что такое тетрада?
   4. Опишите этапы алгоритма преобразования постфиксной записи в последовательность тетрад.
   5. Перечислите известные Вам способы образования производных типов данных.

## Лабораторная/практическая работа № 8

1. Название работы: «Семантика языков программирования. Типы данных. Виртуальные машины, интерпретирующие псевдокод».
2. Цели работы: изучение семантических свойств объектов транслируемой программы, методов их выявления и использования, типов данных и методов контроля типов, областей видимости переменных, локальных и нелокальных сред ссылок, способов передачи параметров, приобретение навыков разработки элементов виртуальной машины для интерпретируемого языка.
3. Основные теоретические сведения:

Классы виртуальных машин

Существует два основных класса виртуальных машин (ВМ), используемых в качестве интерпретаторов: стековые и регистровые.

Стековые ВМ хранят все или почти все обрабатываемые данные в одном или нескольких стеках периода исполнения. Если выполняемая операция вырабатывает результат, то он автоматически помещается в стек и становится его новой вершиной. Для обращения к данным в качестве операндов используется адресация относительного положения требуемого элемента от текущей вершины стека. В подавляющем большинстве случаев в инструкциях псевдокода используется просто верхний элемент стека. Указание такого операнда вообще не требует поля в формате инструкции псевдокода. Это является наиболее существенным достоинством стековых ВМ: компактность кодировки инструкций. Большинство операций псевдокода не хранят явных адресов операндов, так как они работают с вершиной стека.

В стеке обычно хранятся скалярные данные выполняющихся методов и аргументы инструкций псевдокода, такие как ссылки на объекты и массивы, в свою очередь, хранящиеся в отдельной области – «куче» (heap). Кроме обрабатываемых данных в стеке могут также храниться адреса, используемые при возврате из вызванных функций в вызывающие. Возможен вариант, когда для хранения адресов возврата используется отдельный стек.

Нужно отметить, что в стековых ВМ приходится иметь как минимум два регистра: один для хранения указателя на текущую операцию псевдокода, а другой – для указателя на вершину стека. Однако эти регистры, как правило, недоступны для прямого их использования в программе.

Виртуальные машины этого класса – Java VM (во множестве реализаций для интерпретации байткода языка Java) и CLR (от корпорации Microsoft для языка Common Intermediate Language) – в настоящее время используются повсеместно.

Альтернативный подход к хранению обрабатываемых данных состоит в использовании выделенного набора ячеек памяти с фиксированными именами / номерами – регистрами. Инструкции псевдокода в основном оперируют данными из регистров, при необходимости загружая отсутствующие значения из памяти или выгружая не нужные в ближайшее время данные в основную память. Во многих регистровых архитектурах стек также имеется в наличии. Однако он не играет центральной роли в работе ВМ, а используется для поддержки механизма возврата из функций (и в этом случае не обязательно является напрямую доступным для программ). Особенности регистровых ВМ проще всего увидеть, сравнивая их со стековыми, как описано ниже.

* Машинные инструкции регистровых ВМ длиннее, так как в них приходится кодировать операнды. В стековых ВМ операнды задаются неявно.
* При работе регистровых ВМ выполняется меньшее количество обращений к памяти. Стековые ВМ вынуждены постоянно перекладывать данные на стеке, так как время жизни последних невелико (чаще всего операция «съедает» входные значения с вершины стека и замещает их своим результатом). Значение, помещенное в регистр, живет до момента его перезаписи. Это позволяет избавиться от постоянного вычисления или подгрузки из памяти значений для цикловых инвариантов: они просто размещаются в регистрах.
* Однако возникает необходимость использования непростых алгоритмов распределения регистров при трансляции с языков высокого уровня. Для нетривиальных программ всегда будет возникать ситуация нехватки регистров (их количество ограниченно) для размещения всех используемых при вычислении данных. При этом как-то надо отслеживать, в каком регистре что находится в каждый момент времени. В случае же стековых ВМ вершина стека одна, и она всегда доступна. Эти обстоятельства могут значительно усложнять логику работы формирования псевдокода для регистровой ВМ.

Свойства языка, влияющие на реализацию виртуальной машины

При реализации виртуальной машины (интерпретатора) необходимо учитывать множество свойств проектируемого языка программирования.

1. Способы определения и реализации типов обрабатываемых данных, в том числе базовых и производных.
2. Необходимость и степень контроля типов данных объектов программы.
3. Способы выявления эквивалентности типов данных.
4. Правила формирования областей видимости объявлений переменных.
5. Способы активации функций.
6. Способы передачи аргументов из вызывающей функции в вызываемую.
7. Формат записи активации, разделение операций по ее заполнению между вызывающей и вызываемой функциями.
8. Вызывающие и вызываемые последовательности операций активации.
9. Структура среды ссылок периода исполнения.
10. Локальные среды ссылок функций.
11. Способы доступа к нелокальным объектам функций.

Эти свойства следует изучить с использованием источников [1, 6–9].

1. Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample8):
   1. Расширить систему действий синтаксического анализатора, построенного при выполнении работ 3–7, действиями для исполнения последовательности операций псевдокода (триад, тетрад или пентад согласно варианту задания).
   2. Обеспечить выполнение некоторых семантических проверок для выявления возможных ошибок времени исполнения (например, деления на ноль) при интерпретации.
   3. Построить интерпретатор (виртуальную машину) для учебного языка или его подмножества, убедиться в его работоспособности путем исполнения тестовых примеров программ.
2. Подготовить отчет, который должен содержать следующие разделы:
   * + цель работы;
     + краткое изложение задач, возникающих при интерпретации псевдокода;
     + описание структур данных и алгоритмов совокупности действий, разработанных для реализации виртуальной машины для заданного варианта учебного языка;
     + выводы и заключение.
3. Контрольные вопросы
   1. Что такое побочный эффект вызова процедуры/функции?
   2. Что такое глобальный, локальный и нелокальный объект?
   3. Что такое текстуальная область видимости без вложенных функций?
   4. Что такое передача аргумента по имени? Как она реализуется?
   5. Перечислите способы передачи аргументов в функцию.
   6. Что такое жизненный цикл наименования объекта?
   7. Что такое кодированное представление типов данных? В чем состоят его достоинства и недостатки?
   8. Что такое вложенность процедур/функций?
   9. Что такое нелокальная среда ссылок?
   10. Расшифруйте словами тип данных языка С:

*(int\*[](unsigned)).*

* 1. Что такое именное представление типов данных? В чем состоят его достоинства и недостатки?
  2. Что такое передача аргументов методом копирование-восстановление?
  3. Что такое запись активации процедуры/функции?
  4. Как делится ответственность за формирование записи активации между вызывающей и вызываемой функциями?
  5. Что такое дерево активации функций?
  6. Перечислите минимально необходимые поля записи активации функции.
  7. Что такое блочная область видимости переменной?
  8. Что такое вызывающая последовательность?
  9. Что такое l-value и r-value?

## Литература

1. *Малявко А.А.* Формальные языки и компиляторы: учебное пособие для вузов. – М., Изд-во Юрайт, 2021
2. *Малявко А.А.* Формальные языки и компиляторы: учебник НГТУ. – Изд-во НГТУ, 2014, 004 М219, Id = 000184529
3. *Малявко А.А.* Системное программное обеспечение. Формальные языки и методы трансляции: Учеб. пособие. – Новосибирск: Изд-во НГТУ, 2010. – Ч.1, 004 M 219, Id = 143812
4. *Малявко А.А.* Системное программное обеспечение. Формальные языки и методы трансляции: Учеб. пособие. – Новосибирск: Изд-во НГТУ, 2011. – Ч.2, 004 M 219, Id=155235
5. *Малявко А.А.* Системное программное обеспечение. Формальные языки и методы трансляции: Учеб. пособие. – Новосибирск: Изд-во НГТУ, 2012. – Ч.3, 004 M 219, Id=170641
6. *Ахо А., Сети Р., Ульман Д.* Компиляторы: принципы, технологии и инструменты. – М.: «Вильямс», 2001, Id=16803
7. *Карпов Ю.Г*. Теория и технология программирования. Основы построения трансляторов: учеб. пособие. – СПб.: БХВ-Петербург, 2005, Id=64347
8. *Свердлов С. З.* Языки программирования и методы трансляции: учебное пособие для вузов - СПб., 2007, Id=65534
9. *Гавриков М.М., ИванченкоА.Н., Гринченков Д.В.* Теоретические основы разработки и реализации языков программирования. – М.: Кнорус, 2010.

Содержание

Лабораторный практикум по дисциплине «Теория формальных языков и компиляторов»

Основные требования к отчетам по лабораторным работам 1

Лабораторная/практическая работа № 1 2

Лабораторная/практическая работа № 2 17

Лабораторная/практическая работа № 3 25

Лабораторная/практическая работа № 4 36

Лабораторная/практическая работа № 5 52

Лабораторная/практическая работа № 6 73

Лабораторная/практическая работа № 7 85

Лабораторная/практическая работа № 8 93

Литература 97