Лабораторный практикум

Основные требования к отчетам по лабораторным работам

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

Несоблюдение этих требований может привести либо к снижению оценочного балла за работу, 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:

Регулярные выражения представляют собой формализм, 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. <u>*Первичным*</u> называется регулярное выражение, порождающее (описывающее) цепочки, состоящие ровно из одного символа. Это выражения вида:

- [<*произвольный символ*>], например: [*a*]. Говорят, что такое регулярное выражение порождает единственную цепочку, состоящую только из этого символа ([*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*] → *a* и [*aA*] → *A*).
- [<*диапазон символов*>], например [*0–9*]. Это выражение можно считать сокращенной формой записи выражения вида [*0123456789*]. Оно порождает

 10 цепочек, каждая из которых содержит единственную десятичную цифру.

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

С двумя формами записи первичного регулярного выражения (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*].

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 дефиса (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 по четыре цепочки: –,+,*,/, но выражение [+–*/] в системе кодирования ASCII порождает ровно 256 цепочек длины 1, т.е. цепочки, содержащие все символы этого алфавита.

С учетом умолчаний 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.

Аналогичный способ (экранирование следующего символа) применяется для представления символов \, [,], ", являющихся одновременно метасимволами языка регулярных выражений:

\\ – один символ \ ,
\[– символ [,
\] – символ] ,
\" – символ " .

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 применяются следующие знаки операций (первые три называются квантификаторами), используемые для описания цепочек, длина которых больше единицы, и пустых цепочек (т. е. цепочек длины 0).

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

Запись вида [–+]? следует понимать, как возможно отсутствующий знак числа. Это выражение порождает либо пустую цепочку, либо один символ –, либо один символ +.

2. Операция «Одно или несколько» (знак операции +).

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

3. Операция «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*]*

Это регулярное выражение порождает цепочки, начинающиеся с любой латинской буквы, за которой в произвольном порядке может следовать сколько угодно латинских букв и/или цифр.

К первичным выражениям, простым выражениям, и выражениям, построенным с использованием квантификаторов ?, + и * могут применяться еще две операции – конкатенации и выбора.

4. Операция конкатенации (не имеет 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 C/C++.

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].

5. 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, приходится использовать все или некоторые операции 1-5.

Пакет ВебТрансБилдер не поддерживает такие виды первичных выражений, используемые во многих диалектах, как:

– метасимвол «.» для обозначения произвольного символа определяемого языка; вместо этого используется пустая пара квадратных скобок [] (следует помнить, что пробел – это тоже символ и выражение [] вовсе не эквивалентно выражению [ ]);

– знак операции инверсии перечня/диапазона символов «^»; вместо этого используются квантифицируемые выражения (или выражение []) в сочетании

с выражениями, следующими за ними. Так например, запись вида [/][/][]*[\n] пакетом ВебТрансБилдер понимается и обрабатывается так: два символа «косая черта», за которыми следует произвольная цепочка, завершающаяся символом новой строки (это традиционная 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. 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 : [=]

Во второй и третьей строках этой 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 ВебТрансБилдером. 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 заданных групп. Начальный этап алгоритма этого преобразования необходимо изучить по [1], стр. 103-107.

4) Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Пример1):

4.1) 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.

4.2) 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»).

4.3) 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).

4.4) Разработать и сохранить систему правил для всех групп слов языка(или, как минимум, для идентификаторов, констант, знаков операций, пробельных последовательностей), заданного в курсовой работе.

4.5) 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.

4.6) 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»).

4.7) 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.

4.8) 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.

4.9) Подготовить, сдать и защитить отчет к лабораторной работе.

5) 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.

6) Контрольные вопросы.

6.1) Опишите состав и назначение компонентов учебного пакета автоматизации проектирования трансляторов ВебТрансБилдер.

6.2) Перечислите основные функции пакета ВебТрансБилдер.

6.3) С помощью каких пунктов (и подпунктов) меню осуществляются основные операции пакета ВебТрансБилдер?

6.4) Как можно создать новое правило?

6.5) Где пишутся действия в лексических правилах?

6.6) Что такое регулярное выражение?

6.7) Что такое квантификатор?

6.8) Какие квантификаторы можно использовать в пакете ВебТрансБилдер?

6.9) Что такое шаблон построения транслятора?

6.10) Какие знаки операций можно использовать в регулярных выражениях?

6.11) Как в регулярном выражении определить диапазон символов?

6.12) Может ли внутри одной пары квадратных скобках быть записано несколько диапазонов символов?
6.13) Как можно изменить личные настройки в пакете ВебТрансБилдер?
6.14) Может ли быть создано несколько одноименных регулярных определений?
6.15) Как можно задать лексическое правило для слова, содержащего метасимволы языка регулярных выражений, такие как '[', ']', '\', …?
6.16) Каковы приоритеты знаков операций в языке регулярных выражений пакета ВебТрансБилдер?
6.17) Для чего предназначен пункт меню «Показать»?
6.18) Что такое действие в лексическом правиле?
6.19) Как с помощью действий можно заблокировать возврат правильно распознанного слова из лексического анализатора?
6.20) 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:

3.1. Конечные автоматы без памяти

Конечный автомат без памяти есть совокупность

$\boldsymbol{K = \{A, C, C_0, G, F\}}$,

где $\boldsymbol{A}$ – алфавит входных символов; $\boldsymbol{C}$ – конечное множество состояний; $\boldsymbol{C_0}$ – начальное состояние; $\boldsymbol{G}$ – функция переходов, которая по текущему состоянию автомата и входному символу формирует его состояние в следующий момент времени; $\boldsymbol{F}$ – конечное множество состояний останова (финальных состояний).

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

- имеет определенный набор рабочих состояний $\boldsymbol{C}$, среди которых выделено особое начальное (стартовое) состояние $\boldsymbol{C_0}$ и некоторый набор состояний останова (финальных состояний) $\boldsymbol{F}$;
- имеет один вход и не имеет ни одного выхода;
- в любой момент времени на входе имеет либо в точности один символ входного алфавита $\boldsymbol{A}$, либо пустую цепочку ε;
- функционирует в дискретном времени $\boldsymbol{t_0}$, $\boldsymbol{t_1}$, $\boldsymbol{t_2}$, ...;
- при запуске, т. е. в момент времени $\boldsymbol{t_0}$, всегда оказывается в начальном состоянии $\boldsymbol{C_0}$, на входе в этот момент находится самый первый символ входной цепочки;
- в любой момент времени $\boldsymbol{t_i}$ по текущему состоянию и символу, находящемуся на входе, в соответствии с функцией $\boldsymbol{G}$ определяет номер рабочего или финального состояния, в котором автомат окажется в момент времени $\boldsymbol{t_{i+1}}$;
- если по каким-либо причинам номер следующего состояния не может быть определен, то автомат останавливается по ошибке (для остановов по ошибке подразумевается существование особого финального состояния, не принадлежащего множеству $\boldsymbol{F}$);
- при любом переходе в рабочее состояние или состояние останова по ошибке текущий входной символ заменяется следующим символом из входной цепочки;
- при переходе в финальное состояние обнаружения правильного слова входной символ автомата не изменяется (такой переход выполняется по

пустой цепочке ε; действия со входным символом при переходе по пустой цепочке состоят в чтении символа со входа + обнаружении того, что символ не принадлежит текущему слову + возврате этого символа на вход автомата);
- каждому финальному состоянию, не являющемуся состоянием останова по ошибке, поставлена в соответствие группа правильных слов, возможно, не единственная; останов автомата в таком состоянии понимается как обнаружение им правильного слова этой группы (или слова, принадлежащего нескольким группам).

Пусть 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.

1. 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, поскольку 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.

2. 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 (имеющая обычно номер 0), 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, распознающего двоичные числа и последовательности пробелов в представлении, формируемом ВебТрансБилдером:

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.

3. 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 управляющей таблицы (УТ) автомата, предназначенного для акцепта двоичных чисел, формируемой пакетом ВебТрансБилдер показан в табл. 2.1.

Таблица 2.1

	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 показанной 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*.

3.2.Понятие истории работы конечного автомата

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 $\boldsymbol{t}_{\,i}$ указывается порядковый номер ***i*** такта работы автомата.

Пусть дана входная цепочка *101 10*, тогда история работы автомата будет выглядеть так (табл. 2.2.):

Таблица 2.2.

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, в состоянии ***-1*** – об обнаружении конца входной цепочки.

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

Такие автоматы называются *детерминированными*.

3.3.Детерминированность и недетерминированность

Однако 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. Такие автоматы играют важную роль в процессе преобразования систем регулярных определений в детерминированный конечный автомат, поэтому необходимо рассмотреть причины возникновения недетерминированности. Есть два рода (причины) недетерминированного поведения конечных автоматов.

Недетерминированностью первого рода называется наличие хотя бы одного перехода по пустой цепочке символов ε в рабочее состояние. Простой пример совокупности состояний и переходов фрагмента автомата с такой недетерминированностью, формируемый ВебТрансБидером:

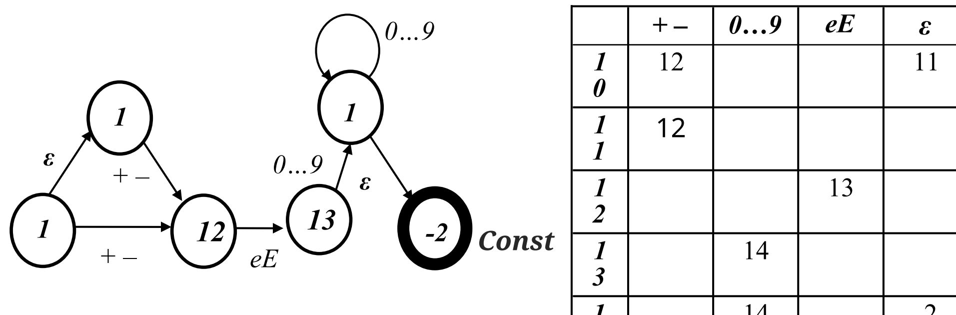
0: [*other*] -> **2** [–+] -> **1**
1: [*other*] -> **2** [0-9] -> **3**
2: [0-9] -> **3**

3: [*other*] -> **-2** [0-9] -> **3**

Этот фрагмент получен в результате выполнения первого шага преобразования [2] формального определения десятичных констант в экспоненциальной форме, содержащих показатель со знаком или без знака:

Const : [0-9]+([.][0-9]*([-+]?[eE][0-9]+)?)?

Другие формы представления этого фрагмента (он не является полностью определенным в силу того, что в управляющей таблице есть пустые клетки) показаны на рис. 2.1.:



	+ –	0…9	eE	ε
10	12			11
11	12			
12			13	
13		14		
1		14		-2

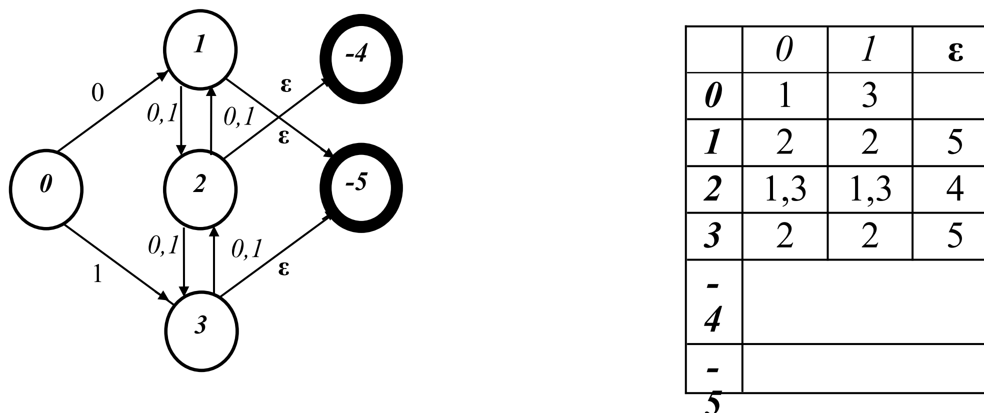
Рис. 2.1. Фрагменты графа состояний и УТ

Автомат, содержащий этот фрагмент, в отличие от детерминированных автоматов, при разных запусках для одной и той же входной цепочки символов может реализовывать разные истории работы. Если он находится в состоянии 10 и на входе один из символов + или –, то возможна два разных продолжения истории работы: либо переход в состояние 12 по + или –, либо переход в состояние 11 по пустой цепочке, а затем в состояние 12 по + или –.

Реализуя разные истории работы, этот автомат тем не менее каждой входной цепочке ставит в соответствие всегда одно и то же финальное состояние.

Недетерминированностью второго рода называется наличие в управляющей таблице клеток, содержащих два или более номера состояния (или столбцов, помеченных одним и тем же символом, но не являющихся идентичными).

В графе состояний и переходов это соответствует дугам, выходящим из одной вершины, помеченным одним и тем же символом, но ведущим в разные вершины. Недетерминированность второго рода, так же, как и первого, порождает возможность реализации нескольких различных историй работы автомата для одной и той же входной цепочки при разных запусках.Пример автомата, с недетерминированностью второго рода, показан на рис. 2.2.



	0	1	ε
0	1	3	
1	2	2	5
2	1,3	1,3	4
3	2	2	5
-4			
-5			

Рис. 2.2. Автомат с недетерминированностью второго рода

В управляющей таблице этого автомата есть клетки, содержащие несколько разных номеров состояний.

Недетерминированность состоит в том, что автомат, оказавшись в состоянии номер ***2***, может перейти по любому из символов *0* или *1* как в состояние ***1***, так и в состояние ***3***.

Несмотря на то что поведение автомата при разных запусках для одной и той же входной цепочки может быть различным, его финальное состояние для любой цепочки будет одним и тем же независимо от того, какая история работы была им реализована.

Недетерминированность автомата как первого, так и второго рода, может возникнуть в результате его построения формальными методами путем преобразования описания лексики языка.

Кроме того, недетерминированность одного рода может возникнуть в процессе эквивалентных преобразований автомата при ликвидации недетерминированности другого рода.

3.4.Недостижимые, тупиковые и эквивалентные состояния

На рис. 2.3 показан граф состояний автомата, содержащего недостижимые, тупиковые и эквивалентные состояния. Автомат имеет два тупиковых состояния с номерами ***2*** и ***3***, три недостижимых состояния с номерами ***6***, ***7*** и ***8*** и эквивалентные состояния ***4*** и ***5***.

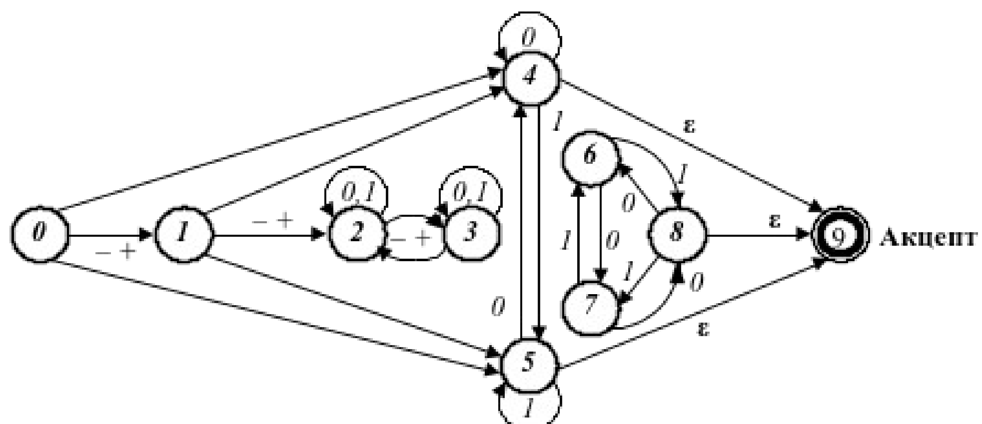


Рис. 2.3. Недостижимые, тупиковые и эквивалентные состояния

Недостижимым называется такое состояние автомата, в которое не существует ни одного пути из начального состояния.

Тупиковым называется состояние, из которого не существует ни одного пути в какое-либо финальное состояние акцепта.

Как тупиковые, так и недостижимые состояния могут быть удалены из автомата вместе со всеми своими (как входящими, так и выходящими) дугами переходов, в результате будет получен автомат, эквивалентный исходному.

Эквивалентными называются такие два состояния, для которых строки в управляющей таблице либо полностью совпадают, либо различаются только переходами друг на друга. Любую пару эквивалентных состояний всегда можно заменить одним состоянием без потери функциональности автомата.

3.5. 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 таблице нет одинаковых столбцов.
8. В рабочей зоне управляющей таблицы нет пустых клеток.

Любой конечный автомат без памяти, не удовлетворяющий хотя бы одному из этих критериев, может быть преобразован в эквивалентный ему оптимальный автомат. Методы и алгоритмы такого преобразования подробно рассматриваются в [1,2], результаты преобразования разработанной студентом системы правил должны быть изучены в этой работе.

4) Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample2):

4.1) Используя пакет ВебТрансБилдер, освоить:

- создание лексических правил на языке регулярных выражений (РВ);
- построения сложных регулярных выражений с использованием способа определения любого символа в виде «[]», квантификаторов вида {<число>,<число>} и операций выбора «|» языка РВ;

4.2) Разработать (доработать разработанный 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.

4.3) Изучить структуру программных модулей, построенных ВебТрансБилдером, 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 по программному коду способ реализации вызова действий, определенных в лексических правилах и алгоритм работы формирователя лексем.

4.4) Проверить функционирование конечных автоматов, построенных ВебТрансБилдером:

- 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.

4.5) Построить вручную в виде двух таблиц из трех строк каждая две истории работы табличного и графового автоматов как результаты обработки ими последовательностей символов, образующих два различных оператора присваивания языка, заданного на курсовую работу.

Строки таблиц должны содержать:

1. Номер шага.
2. Текущий входной символ.
3. Текущее состояние автомата.

Сравнить последовательности обнаруженных слов в построенных историях работы с теми, которые формируются при тестировании сканера, объяснить расхождения, если они имеются.

4.6) Разработать **<u>полное</u>** описание лексики учебного языка, заданного на курсовое проектирование в качестве заготовки фрагмента расчетно-пояснительной записки к курсовой работе. Описание лексики должно включать все группы слов языка. Для каждой группы должны быть сформулированы:

- 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.

4.7) Подготовить, сдать и защитить отчет к лабораторной работе.

5) Требования к содержанию отчета.

Отчет должен содержать:

- цель работы;
- доработанную систему правил, определяющую полностью все группы слов языка, заданного на курсовую работу;
- управляющую таблицу и граф состояний и переходов сканеров, построенных ВебТрансБилдером по этой системе, краткое описание алгоритмов функционирования этих автоматов; результаты отладочной трассировки сканеров;
- две истории работы табличного и графового лексических автоматов, построенные путем ручной имитации работы автоматов при лексическом анализе двух различных оператора присваивания языка, заданного на курсовую работу;
- описание лексики учебного языка, определенного заданием на курсовую работу;
- выводы и заключение;
- приложение – тестовая программа на языке, заданном на курсовую работу, содержащая все элементы языка (функции, блоки операторов и все управляющие операторы, т.е. условные, циклы, переключатели, присваивания и все виды констант).

6) Контрольные вопросы.

6.1) Что такое конечный автомат без памяти?

6.2) Какие существуют способы задания конечных автоматов без памяти?

6.3) Что такое первичное регулярное выражение?

6.4) Как можно задать перечень и диапазон символов в регулярном выражении?

6.5) Что такое квантификаторы, какие квантификаторы можно использовать в системах правил для пакета ВебТрансБилдер?

6.6) Что такое эквивалентность конечных автоматов без памяти?

6.7) Что такое тупиковые, недостижимые и эквивалентные состояния конечного автомата без памяти?

6.8) Что такое оптимальность конечного автомата?

6.9) Перечислите основные этапы процесса преобразования системы регулярных выражений в конечный автомат без памяти.

6.10) Что такое недетерминированность конечного автомата без памяти? Перечислите виды недетерминированности.

6.11) Что такое полностью и неполностью определенные конечные автоматы без памяти?

6.12) Опишите существо процесса эпсилон-замыкания множества вершин графа состояний и переходов при ликвидации недетерминированностей.

6.13) Какие способы преобразования графа состояний и переходов соответствуют операциям языка регулярных выражений?

6.14) Как осуществляется поиск и удаление тупиковых состояний конечного автомата без памяти?
6.15) Как осуществляется поиск и удаление недостижимых состояний конечного автомата без памяти?
6.16) Как осуществляется поиск и слияние эквивалентных состояний конечного автомата без памяти?
6.17) Как осуществляется преобразование не полностью определенного конечного автомата без памяти в полностью определенный?
6.18) Что такое рабочая зона управляющей таблицы конечного автомата без памяти?
6.19) Что такое история работы управляющей таблицы конечного автомата без памяти?
6.20) 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:

3.1. Формальные грамматики

Формальной грамматикой ***G*** называется совокупность

$$\boldsymbol{G = \{ A_t, A_n, S, P \}},$$

состоящая:

- из алфавита терминальных символов A_t;
- алфавита нетерминальных символов A_n;
- начального нетерминального символа S;
- системы правил подстановки P

Алфавит терминальных символов A_t есть конечное множество всех слов языка, порождаемого данной грамматикой. Понятие «терминальный» в данном 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 просто 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: A_t={идентификатор, константа, +, –, *, /, =}

Алфавит нетерминальных символов A_n есть конечное множество названий синтаксических конструкций, например: <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 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* есть один из нетерминальных символов. Этим символом обычно 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.

Для иллюстрации приведем пример фрагмента грамматики С-подобного языка (пока будем считать этот фрагмент полноценной грамматикой $\boldsymbol{G_1}$):

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) предложения арифметические выражения, содержащие только идентификаторы, константы и знаки операций сложения и вычитания.

3.2. 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 $\sigma \to \psi$ не следует возможность непосредственного вывода $\psi \to \sigma$.

Например, в грамматике $\boldsymbol{G_1}$ цепочка $S + const$ непосредственно выводится из цепочки $S + T$ путем подстановки правой части правила $T : const$ вместо T.

Цепочка ψ <u>*выводится*</u> из цепочки σ обозначается $(\sigma \Rightarrow \psi)$, если существует последовательность непосредственных выводов:

$$\sigma \to \psi_1 \to \psi_2 \to \psi_3 \ldots \psi_n \to \psi$$

Например, в грамматике $\boldsymbol{G_1}$ цепочка x – 1 выводится из начального гнетерминала S путем выполнения такой последовательности непосредственных выводов: $S \to S - T \to T - T \to ident - T \to x - T \to x - const \to x - 1$.

<u>*Правильным предложением*</u> языка ***L***, определяемого грамматикой ***G***, называется цепочка, состоящая <u>**только**</u> из терминальных символов и <u>**выводимая**</u> 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 $\boldsymbol{A_t}$, 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. 3.1.

Таблица 3.1

Грамматика G_{a1}		Грамматика G_{a2}	
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	$V : const$

Эти две грамматики различаются не только количеством правил и обозначениями нетерминалов. Пусть дано предложение $(a + b) * c$, где a, b и c – идентификаторы (или константы). Построим выводы этого предложения из начальных нетерминалов грамматик G_{a1} и G_{a2}, заменяя на каждом шаге самый левый нетерминал в очередной цепочке символов на правую часть одного из правил для этого нетерминала:

Для G_{a1}:

$$S \rightarrow T \rightarrow T*V \rightarrow V*V \rightarrow (S)*V \rightarrow (S+T)*V \rightarrow (T+T)*V \rightarrow (V+T)*V \rightarrow (a+T)*V \rightarrow (a+V)*V \rightarrow (a+b)*V \rightarrow (a+b)*c$$

Для G_{a2}:

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

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.

3.3. 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 или 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.

$X \Rightarrow \mu X \eta$,

Существует много вариантов свойства рекурсивности нетерминалов: прямая и 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 ограничено. 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. Грамматики G_{a1} и G_{a2} являются однозначными, но для того же самого языка можно привести пример неоднозначной грамматики, показанный в табл. 3.2.

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 3.2

Грамматика G_{a3}	
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 важное значение для решения задач синтаксического анализа имеют 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 называется *неаннулируемым*. В литературе по формальным языкам 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. Эти отношения вычисляются в виде множеств, сопоставляемых с символами грамматики и содержащих опять-таки символы этой грамматики.

3.4. 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 учебнике) этот термин используется именно в том смысле, который определен в предыдущем абзаце.

3.5.Множества предшественников цепочек символов

При построении синтаксических анализаторов часто приходится оперировать не с отдельными символами, а с 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 $\beta = ABCD...$, 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:

$$M_{\text{пр}}(\beta) = M_{\text{пр}}(A) \cup \begin{cases} \emptyset, \text{ если } A \text{ - неаннулируемый символ,} \\ M_{\text{пр}}(B) \cup \begin{cases} \emptyset, \text{ если } B \text{ - неаннулируемый символ.} \\ \ldots \end{cases} \end{cases}$$

Здесь $\emptyset$ – пустое множество, под неаннулируемым символом понимается либо терминал, либо неаннулируемый нетерминал.

3.6.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:

$S \Rightarrow \omega = \ldots X\, Y \ldots$

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

$N : \varphi\, X\, Y\, \psi$, где φ и ψ – произвольные, возможно пустые цепочки.

Условие 2. В грамматике есть правило вида

$N : \varphi\, X\, \sigma\, Y\, \psi$, где φ и ψ – произвольные, возможно пустые цепочки, а 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 (неважно – непосредственным или нет) некоторого символа Z и выполняется условие 3:

Условие 3. В грамматике есть правило вида

$Z : \chi\, X\, \sigma$, где χ – произвольная, возможно пустая цепочка, а σ – цепочка, либо состоящая только из аннулируемых нетерминалов, либо просто пустая.

И, наконец, символ Y является последователем символа X, если Y есть последователь некоторого символа Z (неважно, непосредственный или нет) и выполняется условие 4:

Условие 4. В грамматике есть совокупность правил следующего вида:

$$
\begin{aligned}
Z &: \chi_1\, Z_1\, \sigma_1 \\
Z_1 &: \chi_2\, Z_2\, \sigma_2 \\
&\dots \\
Z_{n-1} &: \chi_n\, Z_n\, \sigma_n \\
Z_n &: \chi_0\, X\, \sigma_0,
\end{aligned}
$$

где $\chi_0, \chi_1, \dots, \chi_n$ – произвольные цепочки, а $\sigma_0, \sigma_1, \dots, \sigma_n$ – цепочки, состоящие только из аннулируемых нетерминалов или просто 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 лабораторных работ 3-8 и курсовой работы.

4) Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample3):

4.1) Разработать описание синтаксиса языка, заданного на курсовую работу. Описание синтаксиса должно включать:

- структуру файла программы (состав модулей/секций файла);
- структуру и назначение каждой секции/модуля;
- формат и точный способ выполнения всех операторов (присваивания, цикла, …) в виде последовательности операций, сложность которых не выше сложения/ умножения/сравнения/… двух значений, условного или безусловного перехода.

4.2) Используя систему правил Samples/Sample3 как основу:

- 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, содержащих такие символы;
- разобраться в том, каким образом пакет ВебТрансБилдер разделяет правила на лексические и синтаксические; описать принципы этого разделения.

4.3) Изучить по [2-5] теоретические определения и алгоритмы вычисления отношений предшествования и последования для символов грамматики.

4.4) 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»).

4.5) 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.

4.6) 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.

4.7) Подготовить, сдать и защитить отчет к лабораторной работе.

5) Требования к содержанию отчета.

Отчет должен содержать:

- цель работы;
- описание синтаксиса языка как результат выполнения пункта 4.1 и как часть расчетно-пояснительной записки к курсовой работе;
- результаты разработки (пункт 4.5) и тестирования (пункт 4.6) грамматики для языка, заданного на курсовую работу;
- матричное представление отношений предшествования и последования для символов этой грамматики, сформированное пакетом ВебТрансБилдер; краткое описание этих отношений;
- результаты своего 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 пункта 4.5;
- выводы и заключение.

6. Контрольные вопросы.

6.1) Что такое терминальные и нетерминальные символы формальных грамматик?

6.2) Что такое рекурсия? Перечислите виды рекурсии.

6.3) Что такое контекстно-свободные грамматики?

6.4) Вычислите множество предшественников цепочки символов *W R* для грамматики $\boldsymbol{G_{a2}}$.

6.5) Является ли однозначной грамматика:

1. S : S "|" S
2. S : S S
3. S : "(" S ")"
4. S : ident

6.6) Что такое аннулируемый нетерминал?

6.7) Что такое стековая память? Какие операции обычно определяются над стековой памятью?

6.8) Что такое множество последователей символа грамматики?

6.9) Чем понятие вывода отличается от понятия непосредственного вывода?

6.10) Что такое недетерминированность конечного автомата без памяти? Какие бывают виды недетерминированности?

6.11) Устраните левую рекурсию в такой грамматике:

1. *S* : *"(" L ")"*

2. $S : ident$
3. $L : L \text{","} S$
4. $L : S$

6.12) Что такое дерево грамматического разбора?
6.13) Какие нетерминальные символы называются бесплодными?
6.14) Что такое рекурсивный цикл в грамматике?
6.15) Какие грамматики называются эквивалентными?
6.16) Могут ли быть эквивалентными два конечных автомата без памяти, имеющие разное количество финальных состояний?
6.17) Вычислите множества последователей для грамматики из вопроса 6.5.
6.18) Что такое контекстно-зависимая грамматика?
6.19) Что такое отношение предшествования символов?
6.20) Можно ли удалить из грамматики пряморекурсивный нетерминал?
6.21) Постройте дерево разбора предложения (ident, (ident, ident)) в грамматике из вопроса 6.11.
6.22) Что такое факторизация?
6.23) Сформулируйте физический смысл перехода по пустой цепочке в финальное состояние конечного автомата без памяти.
6.24) Можно ли и, если можно, то как преобразовать косвенную рекурсию в прямую?
6.25) Определите алфавиты терминальных и нетерминальных символов для грамматики из вопроса 6.11.
6.26) Какие конечные автоматы без памяти называются эквивалентными?
6.27) Перечислите условия, которые определяют отношение последования символов грамматики.
6.28) Вычислите вручную множества предшественников и последователей и все свойства символов для фрагмента грамматики, определяющий синтаксис условного оператора языка С:
   1. *Operator : ...*
   2. *Operator : "if" "(" LogicalExpression ")" Operator Rest*
   3. *Rest : "else" Operator*
   4. *Rest :*
6.29) Является ли грамматика вопроса 6.29 однозначной?
6.30) Как соотносятся между собой языки и грамматики?
6.31) Могут ли быть эквивалентными два конечных автомата без памяти, имеющие различное количество рабочих состояний?
6.32) Как вычисляется множество предшественников цепочки символов?
6.33) Почему из системы порождающих правил грамматики не может быть удален ее начальный нетерминал?
6.34) 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:

3.1. 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.

Отсюда следует, что выбор правила на каждом шаге должен осуществляться таким образом, чтобы гарантировать:
1. Восстановление дерева для любого правильного предложения и
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.

В этом разделе вначале будет сформулирована основная идея группы нисходящих методов, затем определены критерии выбора правила для выполнения каждого очередного непосредственного вывода и на этой основе – условий, которым должна удовлетворять грамматика для организации восстановления дерева разбора сверху вниз. В общей форме будут определены алгоритм нисходящего синтаксического акцепта, способы процедурной реализации синтаксического акцептора и соответственно преобразования порождающей грамматики в текст программы так

называемого «рекурсивного спуска». Затем будут рассмотрены две разные автоматные реализации нисходящего синтаксического акцептора и соответствующие методы преобразования порождающей грамматики в управляющие таблицы таких автоматов.

3.2. 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 следующем.

A1. Создается корень дерева (он же – самый верхний его уровень), в виде узла, содержащего начальный нетерминал грамматики; из анализируемого предложения считывается первое слово – это текущий терминал (считывание слова – это вызов лексического анализатора). Далее нисходящий синтаксический анализатор постоянно циклически работает ровно с двумя символами: один берется из текущего узла нижнего уровня восстанавливаемого дерева, другим является текущий терминал. В каждом повторении цикла реализуется либо шаг A2, либо шаг A3.

A2. 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:

A2.1. 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. Процесс синтаксического анализа нужно остановить по обнаружению ошибки.

A2.2. Если из правой части просматриваемого правила можно вывести цепочку символов, начинающуюся с текущего терминала из предложения, 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 первый узел этой цепочки или, если правая часть правила пуста, то узел, следующий за обработанным текущим узлом. Текущий терминал не изменяется, выполняется возврат к шагу A2.

A2.3. Если правил, из правой части каждого из которых можно вывести цепочку символов, начинающуюся с текущего терминала из предложения, найдено несколько, 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, потому что:

– 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 (N - количество слов в анализируемой программе).

А3. Если текущий узел нижнего уровня дерева содержит терминал, то возможны ровно три случая:

А3.1 Этот терминал не совпадает с текущим терминалом из предложения. В этом 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 обнаружению ошибки.

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.

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

Таким образом, 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 2.3. Одновременно будет определен способ проверки того, может ли из правой части правила быть выведена цепочка терминалов, начинающаяся с имеющегося на шаге 2 текущего терминала.

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

Пусть есть правило $N : \alpha_j$.

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

1) $M_{\text{выб}}\ (N : \alpha_j) = M_{\text{пр}}\ (\alpha_j)$, если α_j содержит хотя бы один терминал или неаннулируемый нетерминал;

2) $M_{\text{выб}}\ (N : \alpha_j) = M_{\text{посл}}\ (N)$, если α_j есть пустая цепочка;

3) $M_{\text{выб}}\ (N : \alpha_j) = M_{\text{пр}}\ (\alpha_j) \cup M_{\text{посл}}\ (N)$, если α_j не пуста, но состоит только из аннулируемых нетерминалов.

Здесь $M_{\text{пр}}(\alpha_j)$ – множество предшественников цепочки α_j;

$M_{\text{посл}}(N)$ – 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 $\boldsymbol{G}_{a1}$ и $\boldsymbol{G}_{a2}$, 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 $\boldsymbol{G}_{a2}$ (табл. 4.1, 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 (правила 2 и 3 для *R*, правила 5 и 6 для *W*, правила 7, 8 и 9 для *V*), не пересекаются. Поэтому при восстановлении дерева разбора в любом цикле на шаге 2 по любому текущему входному символу можно выбрать ровно одно правило для замены нетерминала – то, чье множество выбора содержит текущий входной терминал.

Таблица 4.1

Грамматика $\boldsymbol{G}_{a1}$				Грамматика $\boldsymbol{G}_{a2}$			
	Правило	Способ	Множество		Правило	Способ	Множество
0	*Z* : *S* ►	$M_{\text{пр}}$(*S* ►)	*(, ident, const*	**0**	*Z* : *S* ►	$M_{\text{пр}}$(*S* ►)	*(, ident, const*
1	*S* : *S* + *T*	$M_{\text{пр}}$(*S* + *T*)	*(, ident, const*	**1**	*S* : *U R*	$M_{\text{пр}}$(*U R*)	*(, ident, const*
2	*S* : *T*	$M_{\text{пр}}$(*T*)	*(, ident, const*	**2**	*R* : + *S*	$M_{\text{пр}}$(+ *S*)	+
3	*T* : *T* * *V*	$M_{\text{пр}}$(*T* * *V*)	*(, ident, const*	**3**	*R* :	$M_{\text{посл}}$(*R*)	*(,* ►
4	*T* : *V*	$M_{\text{пр}}$(*V*)	*(, ident, const*	**4**	*U* : *V W*	$M_{\text{пр}}$(*V W*)	*(, ident, const*
5	*V* : *(S)*	$M_{\text{пр}}$(*(S)*)	*(*	**5**	*W* : * *U*	$M_{\text{пр}}$(+ *S*)	*
6	*V* : *ident*	$M_{\text{пр}}$(*i*)	*ident*	**6**	*W* :	$M_{\text{посл}}$(*W*)	+, *(,* ►
7	*V* : *const*	$M_{\text{пр}}$(*c*)	*const*	**7**	*V* : *(S)*	$M_{\text{пр}}$(*(S)*)	*(*
				8	*V* : *ident*	$M_{\text{пр}}$(*ident*)	*ident*
				9	*V* :	$M_{\text{пр}}$(*const*)	*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 ткущий 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:

$$S \Rightarrow (\, x + y\,)\ \boxed{W}\, R$$

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 $\boldsymbol{G_{a2}}$ принадлежит к классу **LL(1)**-грамматик.

Обратимся теперь к грамматике $\boldsymbol{G_{a1}}$ (табл. 4.1, 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

$S \rightarrow T$

будет выбрана подстановка по правилу 1

$S \rightarrow S + T$.

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 $\boldsymbol{G_{a1}}$ возникает каждый раз, когда на текущем уровне

восстанавливаемого дерева самым левым оказывается либо нетерминал *S*, либо нетерминал *T*.

Грамматика $\boldsymbol{G_{a1}}$ 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*:

$N : N\,\beta$

$N : \gamma$.

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

$M_{\text{выб}}(N\,\beta) = M_{\text{пр}}(N\,\beta)$,

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

Итак, любая леворекурсивная грамматика не принадлежит классу **LL(1)**-грамматик. Левую рекурсию всегда можно преобразовать в правую или в 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 т.д. 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:

- парсер, как нисходящий стековый автомат с одним состоянием;
- парсер, как нисходящий стековый автомат с несколькими состояниями;
- парсер, как совокупность функций.

3.4. 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, последний символ окажется на самом верху стека;

^ – снятие одного символа с верхушки 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.

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 мы будет показано, что в процессе преобразования возможно появление дополнительных строк). Озаглавить столбцы терминалами грамматики (порядок следования столбцов не имеет значения), строки – нетерминалами (опять же в произвольном порядке).

Шаг 2. В силу т ого, 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 $N : M\ \alpha$, где M – нетерминальный символ, а α – цепочка символов $s_1\ s_2\ ...\ s_k$ (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:

^ ! $s_k\ ...\ s_2\ s_1\ M$

Если среди символов $s_1\ s_2\ ...\ s_k$ 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 $N : t\ \alpha$, где t – терминальный символ, а α – возможно, пустая цепочка символов $s_1\ s_2\ ...\ s_k$, 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:

^ ! $s_k\ ...\ s_2\ s_1$ >

Если среди символов $s_1\ s_2\ ...\ s_k$ 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 верхушки 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 верхушке 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. И, 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

3.5. 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, на каждом такте по входному символу и текущему состоянию 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 верхушки 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.

2. 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 при соблюдении определенных правил нумерации состояний и введении операции управления остановом по ошибке можно обойтись только одним адресом перехода.

С каждым состоянием должны быть также связаны операции управления стековой памятью (занесение адреса возврата, снятие адреса с верхушки стека и переключение в состояние возврата) и операция управления чтением следующего входного символа. Все операции управления могут задаваться

булевскими 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 1) в стек;
- флажок **r** обеспечивает переключение автомата в состояние, номер которого снимается с верхушки 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 включаются дополнительное поле, указывающее 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 $\boldsymbol{G}_{a2}$, 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:

- 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 и .

Таблица 4.2.

Грамматика $\boldsymbol{G}_{a2}$	
0	$Z : S_0$ ►$_1$
1	$S_2 : U_{11}\, R_{12}\, \boldsymbol{\varepsilon}_{13}$
2	$R_3 : +_{14}\, S_{15}\, \boldsymbol{\varepsilon}_{16}$
3	$R_4 : \boldsymbol{\varepsilon}_{17}$
4	$U_5 : V_{18}\, W_{19}\, \boldsymbol{\varepsilon}_{20}$
5	$W_6 : *_{21}\, U_{22}\, \boldsymbol{\varepsilon}_{23}$
6	$W_7 : \boldsymbol{\varepsilon}_{24}$
7	$V_8 : (_{25}\, S_{26}\,)_{27}\, \boldsymbol{\varepsilon}_{28}$
[illegible]	[illegible]

В табл. 4.2. приведены результаты выполнения шага 1 для модифицированной грамматики G_{a2}.

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 $N : \alpha$, 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, находящимся 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, находящимся 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** (в том случае если флажок **r** установлен, переход осуществляется по адресу, снимаемому со стека возвратов). Если флажок в данном состоянии **r** установлен, 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 (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. 4.3. приведены результаты применения этой процедуры преобразования грамматики в управляющую таблицу автомата для грамматики $\boldsymbol{G}_{a2}$ (в полях флажков управления значению *true* сопоставлено 1, значению *false* – пустая клетка).

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.

Таблица 4.3.

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* +( )* ▶	

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:

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

Шаг 5.1.2. Если стек пуст – перейти к шагу 7.

Шаг 5.2. Если флажок **r** не установлен, то:

Шаг 5.2.1. Если текущим является состояние 1:

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

Шаг 5.2.1.2. Если стек не пуст, 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.

3.6. 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;
- при положительном результате проверки «реализует» правую часть правила, двигаясь по ее символам слева направо и:
  - вызывая соответствующие функции парсера (возможно и сама себя), если очередной символ – это нетерминал;
  - сравнивая символ из правила с текущим терминалом, если это терминал и:
    - вызывая лексический анализатор для чтения следующего терминала из предложения при совпадении символов;
    - возвращая значение false (ложь) при несовпадении;
  - возвращая значение true (истина), если был обработан последний символ правой части правила (или правая часть пуста);
- возвращает значение false (ложь), если не было найдено ни одного подходящего правила.

Детально способ преобразования **LL(1)**-грамматики в программный код описан в [1-5].

4) Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample4):

4.1) Используя пакет ВебТрансБилдер:

- 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; изучить

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 «~»;
- построить конечный автомат со стековой памятью и несколькими состояниями, найти в коде построенного автомат управляющую таблицу, описать для отчета ее структуру;
- построить конечный автомат со стековой памятью и одним состоянием, управляемый входным символом и символом, снятым с верхушки стека, найти в коде построенного автомат управляющую таблицу, описать для отчета ее структуру;
- построить программную реализацию нисходящего рекурсивного спуска, найти в коде функцию, построенную для нетерминала, определяющего оператор присваивания, описать ее алгоритм для отчета.

4.2) Выполнить трассировку процессов нисходящего синтаксического акцепта для всех построенных синтаксических анализаторов, 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.

4.3) Проанализировать и сравнить между собой все полученные тексты программ и результаты выполнения пункта 4.2. Оценить степень пригодности изученных вариантов реализации нисходящих синтаксических акцепторов для выполнения курсовой работы, отразить результаты этой оценки в заключении к отчету.

4.4) Подготовить, сдать и защитить отчет к лабораторной работе.

5) Требования к содержанию отчета.

Отчет должен содержать:
- цель работы;
- систему правил LL(1)-грамматики для языка, заданного на курсовую работу;
- описание этой грамматики в качестве фрагмента расчетно-пояснительной записки к курсовой работе;
- 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 фрагментов этих историй для разбора оператора присваивания;
– выводы и заключение.

6) Контрольные вопросы.
   6.1) Что такое множество выбора правила грамматики?
   6.2) Постройте историю работы нисходящего синтаксического акцептора с одним состоянием при разборе цепочки: *(((x)))*
   6.3) Напишите **LL(1)**-грамматику для условного оператора С-подобного языка, имеющего и полную и сокращенную форму.
   6.4) Что такое s-грамматика?
   6.5) Какие операции способен выполнять автомат нисходящего восстановления дерева грамматического разбора с одним состоянием?
   6.6) Что такое **LL(1)**-грамматики?
   6.7) Что такое детерминированное (безвозвратное) восстановление дерева грамматического разбора?
   6.8) Почему леворекурсивная грамматика не может быть преобразована в нисходящий синтаксический акцептор?
   6.9) Опишите технологию разработки процедурной реализации рекурсивного спуска.
   6.10) Как вычисляются множества выбора правил грамматики?
   6.11) Перечислите поля клетки нисходящего автомата с несколькими состояниями.
   6.12) Как формируются значения флагов в управляющей таблице нисходящего синтаксического акцептора с несколькими состояниями?
   6.13) Разработайте **LL(1)**-грамматику для оператора переключателя (switch) С-подобного языка.
   6.14) В каком порядке нумеруются символы грамматики при преобразовании в управляющую таблицу автомата с несколькими состояниями?
   6.15) Чем управляется нисходящий автомат с одним состоянием?
   6.16) Каково назначение каждого управляющего флажка управляющей таблицы автомата с несколькими состояниями?
   6.17) Как формируются значения управляющих флагов нисходящего автомата с несколькими состояниями?
   6.18) Опишите последовательность действий при преобразовании грамматики в процедурную реализацию нисходящего синтаксического акцептора.
   6.19) Как формируются множества выбора состояний нисходящего синтаксического акцептора с несколькими состояниями?
   6.20) 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:

3.1. 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 все 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. Опять-таки, как и при нисходящем анализе, не любая грамматика обладает такими свойствами, чтобы по ее правилам можно было осуществлять детерминированное восходящее восстановление дерева разбора.

3.2. Структура автомата для восходящего восстановления дерева грамматического разбора

Восходящий синтаксический акцептор представляет собой конечный автомат со стековой памятью (рис. 5.1.), управляемый входным символом и текущим состоянием.

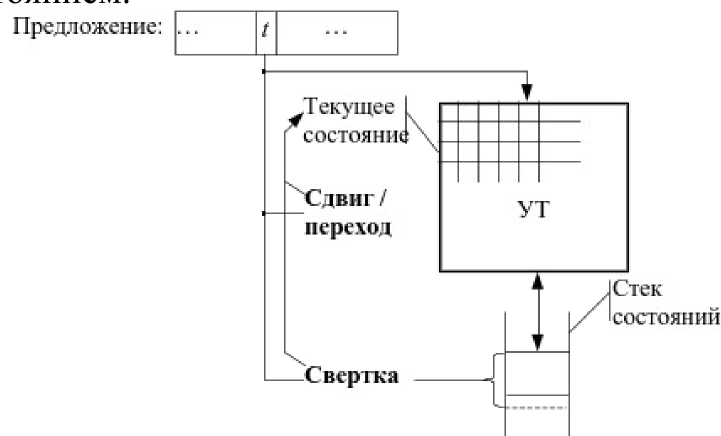


Рис. 5.1. Восходящий парсер как автомат с несколькими состояниями и стеком состояний

Перед запуском автомата номер его начального состояния (состоянии с номером 0) должен быть занесен в стек. Далее на каждом шаге работы номер состояния, находящий на верхушке стека, используется в качестве текущего и определяет выбор строки управляющей таблицы. Из входного предложения прочитан 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, таковы:

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

Gn – переход (**Go**) в новое состояние. Номер состояния n заносится на верхушку 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.

Rk,N – свертка (**Reduce**) правой части правила для нетерминала N, имеющей длину k символов. С верхушки 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 верхушке 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 следует т в качестве парсера. При обнаружении отрицательного результата можно попытаться построить автомат, применяя более сложный алгоритм, или же попытаться модифицировать грамматику или ее свойства для повторного применения уже использовавшегося алгоритма. Алгоритмы преобразования используют то или иное понятие конфигурации. Начнем с простейшего.

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

Для иллюстрации процессов, протекающих при построении восходящего парсера, будем использовать грамматику языка арифметических выражений $\boldsymbol{G}_{a1}$ с добавленным специальным правилом вида Z : S ► (табл. 5.1.)

Таблица 5.1.

Грамматика $\boldsymbol{G}_{a1}$	
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*

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 ▼):

$$S : \blacktriangledown\ S + T$$

$$S : S\ \blacktriangledown\ + T$$

$$S : S +\ \blacktriangledown\ T$$

$$S : S + T\ \blacktriangledown$$

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.

3.4. Связи между конфигурациями и операциями.

Если в некоторой конфигурации маркер находится перед терминалом ($N : \alpha \blacktriangledown t\ \beta$), то в состоянии номер m, которому соответствует эта конфигурация, должна выполняться операция **Shift**, осуществляющая переключение автомата в такое состояние, которому соответствует конфигурация с маркером, размещенным после этого терминала ($N : \alpha\ t\blacktriangledown\ \beta$),. 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 $N : \alpha \blacktriangledown M\ \beta$ 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. $N : \alpha\ M\blacktriangledown\ \beta$. 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).

И, наконец, если маркер в конфигурации находится после последнего символа 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 соответствующего правила), переключающая автомат в состояние, номер которого оказался на верхушке стека после этого удаления, и готовящая номер столбца n управляющей таблицы для последующей операции Go.

3.5. Связи между конфигурациями и состояниями автомата.

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

$$N : \alpha \blacktriangledown M \beta ,$$

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

$$M : \gamma_1$$

$$M : \gamma_2$$

$$...$$

$$M : \gamma_n ,$$

то все конфигурации вида $M : \blacktriangledown \gamma_1$, $M : \blacktriangledown \gamma_2$, ... $M : \blacktriangledown \gamma_n$ 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 $N : \alpha \blacktriangledown M \beta$.

Поскольку входная цепочка правильна, 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 в нем 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,n_x. Здесь k – длина цепочки χ, а n_x – 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 **Go**, если *w* – нетерминал. Выполнение такой операции переключает автомат в некоторое другое состояние С, номер которого заносится на верхушку стека.

Применительно к подмножеству конфигураций, в которых маркер находится перед символом *w*, такую операцию следует трактовать как перенос маркера через этот символ. Тем самым образуется подмножество конфигураций, которое 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 просто 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.

Эта процедура выполняется в два этапа.

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

Этап 2. Преобразование таблицы конфигураций в управляющую таблицу восходящего акцептора.

3.6. Этап 1. Определение набора состояний восходящего акцептора

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

Начальное состояние таблицы подмножеств конфигураций (далее – просто таблицы конфигураций) для любой грамматики (с точностью до наименования начального нетерминала грамматики) определяется добавочным правилом и выглядит, как показано в табл. 5.2.

Таблица 5.2.

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

Назначение колонок таблицы конфигураций.

Состояние – содержит номер, присвоенный состоянию автомата, определяемому подмножеством конфигураций.

Образовано – показывает, из подмножества конфигураций какого состояния образовано данное состояние путем переноса маркера 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 – пояснения не требуется.

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. 5.3.

Таблица 5.3.

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), 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. Возможны 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, что 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 $\boldsymbol{G}_{a1}$ приведен в табл. 5.4. 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 применении второго шага процедуры. Все остальные шаги приводили к образованию уже существующих базовых подмножеств конфигураций.

Таблица 5.4.

Состояние	*Образовано*		*База*	*Конфигурация*	*Символ*	*Отм.*
	Из	*Через*				
0			Да	Z : ▼ S ►	S	1

				$S : ▼ S + T$	S	1
				$T : ▼ T$	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	$V : (S) ▼$		

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

В результате построения таблицы конфигураций (табл. 5.4.) было выяснено, что автомат, реализующий восходящее восстановление дерева грамматического разбора предложений языка, порождаемого грамматикой G_{a1}, должен иметь ровно 13 состояний (с номерами 0…12). Общее количество символов (как терминальных, так и нетерминальных) вместе с псевдотерминалом ► равно 10.

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. ?). 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 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. 5.4.) позволит получить такую управляющую таблицу автомата (табл. 5.5.).

Таблица 5.5.

	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 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*.

В табл. 5.5. 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 $\boldsymbol{G_{a1}}$, возникновение конфликтов обусловлено не их свойствами и даже не свойствами порождаемого языка, а всего только несовершенством способа занесения знаков операций свертки в управляющую таблицу, сформулированного в шаге 3 процедуры.

3.8. 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), 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 G_{a1}. Теперь не возникают ранее фиксировавшиеся конфликты типа сдвиг/свертка, и управляющая таблица автомата получает вид, показанный в табл. 5.6.

Теперь автомат является детерминированным.

Для грамматики G_{a1} при построении управляющей таблицы не возникали конфликты типа свертка/свертка. В том случае если набор конфигураций некоторого состояния содержит две или более конфигураций вида

$N : \beta \blacktriangledown$

$M : \beta \blacktriangledown$,

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

Таблица 5.6.

	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

При использовании множеств последователей для занесения знаков операций свертки в таблицу возникновение конфликтов зависит от того, пересекаются ли $M_{\text{посл}}(N)$ и $M_{\text{посл}}(M)$. 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.

3.9. 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 5.7.

Грамматика G_{a0}	
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. 5.7.

Однако синтаксис типичных языков программирования не может быть определен **LR(0)**-грамматикой. Этот класс грамматик слишком узок и может представлять собой 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 G_{a1} 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.

3.10. 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. Конфигурацию, к которой приписан в точности один символ ожидаемого правого контекста, будем называть канонической расширенной конфигурацией:

$N : \alpha$ ▼ $\beta, \{\, t \,\}$

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 в разделе 3.3.4 ранее, состоит в следующем.

Если имеется конфигурация вида $N : \alpha \blacktriangledown X \beta$, 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:

$X : \gamma_1 \rightarrow X : \blacktriangledown \gamma_1$

$X : \gamma_2 \rightarrow X : \blacktriangledown \gamma_2$

...

$X : \gamma_n \rightarrow X : \blacktriangledown \gamma_n$

При построении таблицы канонических расширенных конфигураций с любой исходной конфигурацией $N : \alpha \blacktriangledown X \beta$ связан правый контекст, содержащий символ t_0, ожидаемый на входе автомата после свертки цепочки $\alpha X \beta$ в нетерминал N:

$N : \alpha \blacktriangledown X \beta , \{t_0\}$.

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

$M_{опк}(X : \blacktriangledown \gamma_i) = M_{пред}(\beta)$,

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

$M_{опк}(X : \blacktriangledown \gamma_i) = \{ t_0 \}$,

если β – пустая цепочка;

$M_{опк}(X : \blacktriangledown \gamma_i) = M_{пред}(\beta) \cup \{ t_0 \}$,

если β – цепочка, состоящая только из аннулируемых нетерминалов.

Получив множество символов ожидаемого правого контекста $M_{опк}(\alpha \blacktriangledown X \beta, \{t_0\}) = \{ t_1, t_2 \dots t_k\}$, легко можно построить все дочерние по отношению к исходной канонические конфигурации:

$M : \blacktriangledown \gamma_1 , \{ t_1\}$ $M : \blacktriangledown \gamma_1, \{ t_2\}$ … $M : \blacktriangledown \gamma_1 , \{ t_k\}$

$M : \blacktriangledown \gamma_2 , \{ t_1\}$ $M : \blacktriangledown \gamma_2 , \{ t_2\}$ … $M : \blacktriangledown \gamma_2 , \{ t_k\}$

...

$M: \blacktriangledown\gamma_n, \{t_1\}$ $M: \blacktriangledown\gamma_n, \{t_2\}$ … $M: \blacktriangledown\gamma_n, \{t_k\}$

Теперь окончательно сформулируем правила построения таблицы канонических конфигураций.

1. Базой нулевого состояния является конфигурация вида

$Z: \blacktriangledown S$ ► , {►},

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

2. 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).

3. 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 – 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».

3.11. **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:

$N: \alpha \blacktriangledown \beta, \{t_1, t_2, \ldots t_k\}$.

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:

$N: \alpha \blacktriangledown \beta, \{t_1\}$ и $N: \alpha \blacktriangledown \beta, \{t_2\}$ эквивалентны $N: \alpha \blacktriangledown \beta, \{t_1, t_2\}$.

1. Сформулируем две разных технологии построения таблицы расширенных конфигураций. Согласно одной технологии должно быть

выполнено следующее. Строится таблица канонических конфигураций как в разделе 3.3.9.

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 $К_{исх}$), то пересчитываются ожидаемые правые контексты всех ее дочерних конфигураций, т.е. таких, для которых $К_{исх}$ является родительской либо при выполнении замыкании, 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, 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 $К_{исх}$), то пересчитываются ожидаемые правые контексты всех ее дочерних конфигураций, т. е. таких, для которых $К_{исх}$ 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 ВебТрансБилдер.

Вторая технология заключается в следующем:

1. Базой нулевого состояния является конфигурация вида

$Z : \blacktriangledown S \blacktriangleright , \{\blacktriangleright\},$

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

2. 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 $К_{исх}$), то ее множество символов ожидаемого правого контекста объединяется с ОПК новой конфигурации. Если ожидаемый правый контекст конфигурации $К_{исх}$ изменился в результате объединения, то пересчитываются ожидаемые правые контексты всех тех конфигураций, для которых конфигурация $К_{исх}$ 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.

3. 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, что 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.

4) Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample5):

4.1) Завершить разработку грамматику языка, заданного на курсовую работу. Грамматика должна определять программы, включающие несколько функций.

4.2) Используя пакет ВебТрансБилдер и полную грамматику своего языка:

- 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.

4.3) 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.

4.4) Проанализировать и сравнить между собой все полученные тексты программ и результаты выполнения пункта 4.2. Оценить степень пригодности изученных вариантов реализации нисходящих синтаксических акцепторов для выполнения курсовой работы. Отразить в заключении к отчету результат анализа и оценки.

4.5) Подготовить, сдать и защитить отчет к лабораторной работе.

5) Требования к содержанию отчета.

Отчет должен содержать:

- цель работы;
- фрагменты кода процедурной реализации восходящего и управляющей таблицы автоматной реализации восходящего

синтаксического акцептора, описание структур данных и алгоритмов 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.

6) Контрольные вопросы

6.1) Что хранится в стеке восходящего парсера?
6.2) Что такое операция перехода восходящего парсера?
6.3) Если грамматика относится к классу **SLR(1)**, то можно ли по ней построить нисходящий синтаксический акцептор?
6.4) Перечислите все знаки операций восходящего синтаксического акцептора, объясните назначение их полей (параметров).
6.5) Постройте историю работы восходящего синтаксического акцептора при разборе цепочки: *(((x)))*
6.6) Что такое конфигурация?
6.7) Напишите **LR**-грамматику для условного оператора С-подобного языка, имеющего и полную и сокращенную форму.
6.8) Какие грамматики относятся к классу **LALR(1)**?
6.9) Что такое конфликт «сдвиг/свертка»?
6.10) Как таблица конфигураций преобразуется в управляющую таблицу восходящего синтаксического акцептора?
6.11) Что такое ожидаемый правый контекст?
6.12) Что делает восходящий автомат при выполнении операции свертки?
6.13) Что такое детерминированное (безвозвратное) восстановление дерева грамматического разбора?
6.14) Как вычисляется ожидаемый правый контекст конфигурации?
6.15) Опишите технологию разработки процедурной реализации восходящего синтаксического акцептора.
6.16) Как выполняется операция сдвига?
6.17) Что такое **LR(1)**-грамматика?
6.18) Что такое конфликт «свертка/свертка»?
6.19) Перечислите все операции восходящего синтаксического акцептора.
6.20) Чем различаются понятия расширенной и канонической расширенной конфигурации?
6.21) Как используются множества последователей нетерминалов для предупреждения конфликтов при построении управляющей таблицы восходящего синтаксического акцептора по **SLR(1)**-грамматике?
6.22) 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:

3.1. 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 выполнения исходной (мысленного) и преобразованной (физического) программ 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 или ветки условного оператора или переключателя. В машинном или в 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 C/C++ 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, таких как условные операторы, переключатели и операторы 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 C/C++ 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 булевское 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 тем, что:

– 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.

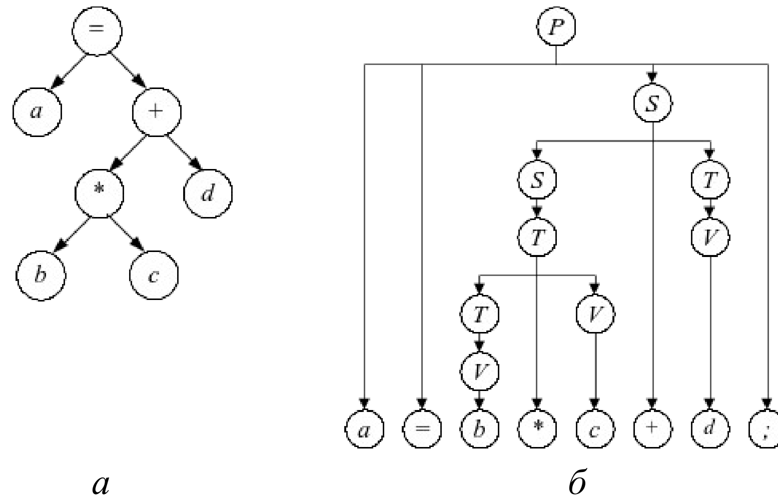
3.2. Синтаксические деревья и постфиксная форма записи

Синтаксическим деревом или деревом операций называется такое графическое представление совокупности операций, связанных значениями обрабатываемых данных (операндами), в котором:

- 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. 6.1, *а*. На рис. 6.1, *б* для сравнения показано дерево грамматического разбора этого оператора в грамматике $\boldsymbol{G_{a1}}$,

расширенной путем добавления правила $P : i = S ;$ для нового начального нетерминала P.



а *б*

Рис. 6.1. *Связь дерева операций и дерева разбора:*

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

Синтаксическое дерево (рис. 6.1., *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.

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. 6.1., *б*, также содержит всю необходимую информацию для решения этой 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 верхушки 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 приведено на рис. ?, *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, 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 дереву операций, построенному нами для оператора присваивания *a=b*c+d;*, позволит получить такую 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 C/C++) 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 следует заметить, что:

- 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 $\boldsymbol{G_{a1}}$ 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.

3.3. Включение действий в грамматику для преобразования предложения в постфиксную форму записи

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

1. Постфиксной записью пустой цепочки символов $\boldsymbol{\varepsilon}$ является $\boldsymbol{\varepsilon}$.

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, а *incPre* (*decPre*) – имя функции (знак операции), увеличивающей (уменьшающей) значение своего аргумента на единицу и возвращающей измененное значение.

7. 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, а *incPost* (*decPost*) – имя функции (знак операции), увеличивающей (уменьшающей) значение своего аргумента на единицу и возвращающей еще не измененное значение.

8. Если E_1, E_2, … E_k – выражения вида 2…9, то постфиксной записью выражения ○ *(*E_1, E_2, … E_k*)* является ПФЗ(E_1) ПФЗ(E_2) ... ПФЗ(E_k) ○, где ○ – знак любой k-арной операции (функции с k аргументами), а ПФЗ(E_1), ПФЗ(E_2) … ПФЗ(E_k) – постфиксные записи выражений E_1, E_2 … E_k соответственно.

9. 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, …), а ПФЗ(E_1) и ПФЗ(E_2) – постфиксные записи выражений E_1 и E_2 соответственно.

10. Если E_1, E_2, E_3 – выражения вида 2…8, то постфиксной записью выражения E_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.

11. 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 C/C++, например, существует не один, а несколько знаков операции присваивания: =, +=, *=, …).

Подобные определения можно сформулировать и для других синтаксических конструкций языка программирования. Руководствуясь этими определениями, можно 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 G_{a1}). 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 по 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* по той простой причине, 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 восходя щего 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 так же 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 это делает 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 передач управления при выявлении линейной последовательности операций для операторов цикла, условных операторов и переключателей.

3.4. 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 C/C++.

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

Operator : *while (Expression) Block*

Предполагается, что определены и другие операторы (присваивания, условный, … 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:

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*), 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 так: 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, что

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:

- 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 его во вспомогательном стеке, доступном из действий, вставляемых в грамматику;
- использовать значение, находящееся на верхушке 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*, и что для операций над вспомогательным стеком имеются функции с прототипами:

void Push(int); – поместить значение аргумента на верхушку стека;
int Pop(void); – возвратить значение, удалив его с верхушки стека;
int Top(void); – вернуть значение с верхушки, 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 другие формы операторов цикла (*for, do, …*), условные операторы, переключатели и т. д.

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.

4) Порядок выполнения работы(рекомендуется использовать в качестве примера систему правил Samples/Sample6):

4.1) Реализовать полную грамматику языка, заданного на курсовую работу.

4.2) Используя пакет ВебТрансБилдер, изучить и освоить:

- способы включения структур данных и операций в грамматику, полученную в результате выполнения предыдущих лабораторных 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.

4.3) Использовать полученные навыки для разработки преобразователя в ПФЗ для выражений, операторов присваивания и по меньшей мере одного управляющего оператора языка, заданного на курсовую работу.

4.4) Подготовить, сдать и защитить отчет к лабораторной работе.

5) Требования к содержанию отчета.

Отчет должен содержать:

- цель работы;
- краткое описание свойств постфиксной записи и методов ее формирования;
- реализацию действий для формирования ПФЗ, включенных в грамматику языка, заданного 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.

6) Контрольные вопросы

6.1) Для чего операторы программы преобразуются в постфиксную форму записи?

6.2) Эквивалентны ли абстрактное синтаксическое дерево и постфиксная форма записи программы?

6.3) Что такое детерминированное (безвозвратное) восстановление дерева грамматического разбора?
6.4) Сформулируйте основные принципы преобразования арифметических выражений в постфиксную запись.
6.5) Почему леворекурсивная грамматика не может быть преобразована в нисходящий синтаксический акцептор?
6.6) Что такое постфиксная запись?
6.7) Каковы свойства постфиксной формы записи?
6.8) 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:

3.1. 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 – промежуточное представление, которое образуется в процессе синтаксического анализа.
2. Набор информационных таблиц, в которых накоплены сведения об используемых программой данных.

Постфиксная форма записи обладает замечательным 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.

1. Компиляция – 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.
2. Интерпретация – 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 run-time 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 анализаторы.

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 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 совсем не обязательно совпадает с логической последовательностью.

Семантическая проверка некоторого фрагмента программы может предшествовать во времени 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, …) 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.

В том случае если возможно выполнение всех этапов анализа произвольного фрагмента текста программы без обработки всего ее текста целиком, говорят, что язык допускает однопроходную трансляцию. Ряд языков (Pascal, C и др.) был спроектирован таким образом, чтобы обеспечить возможность быстрой однопроходной трансляции.

Однако существуют и такие языки программирования, в которых выполнение всех функций анализа некоторого фрагмента текста программы невозможно без предварительной обработки последующих фрагментов для извлечения и накопления сведений об используемых объектах и их свойствах (например, Алгол-68 или Ада).

Трансляторы с таких языков вынужденно реализуют более чем один проход по тексту программы (или по одному из ее промежуточных представлений – последовательности токенов (лексем), постфиксной записи, …).

Увеличение количества проходов, с одной стороны, приводит к росту затрат времени на трансляцию программ и сложности транслятора, а с другой – хорошо согласуется с потребностями алгоритмов оптимизации программы.

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. 7.1.

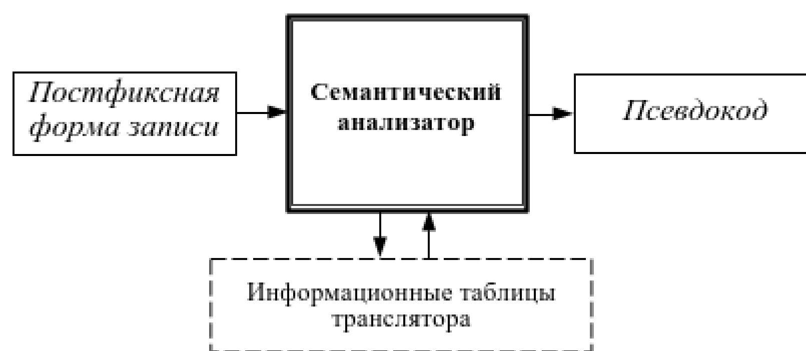


Рис. 7.1. Структура семантического анализатора

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

label#0#0 <u>*jmp*</u> – ПФЗ завершения оператора цикла;

label#0#1: first <u>*return*</u> – ПФЗ оператора возврата значения функции.

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

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 <u>*!=*</u> *label#0#1 jmpOnFalse*

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

Для того чтобы выявить все такие 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 4 поля (и называется тетра дой): 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. 7.1.

Виртуальная машина, псевдокод которой показан в этой таблице, является стековой. Результаты некоторых операций заносятся в стек с помощью указания имени верхушки стека *push* и извлекаются из стека в последующих тетрадах с использованием другого имени верхушки 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 дополнительной поле метки. Команды такой виртуальной машины называют пентадами (по количеству полей).

Таблица 7.1.

Код операции	**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*		

В табл. 7.2. 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 7.2

Метка	**Код 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. 7.3.

Таблица 7.3.

Код операции	**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*	

В этом примере вместо имен операций и промежуточных результатов используется относительная адресация. Числа вида *+8* или *–1* в полях наименований операндов означают указание триады, находящейся на расстоянии 8 вперед или 1 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.

3.2.Преобразование постфиксной записи в последовательность тетрад

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 (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, далее формируются наименования операндов тетрады согласно выявленному случаю.

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

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

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

Случай k>2 – с верхушки 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.

Следует отметить, 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 том 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 2 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 и на y в поле операнд2 тетрады.

Теперь, когда мы выяснили как выглядит внутреннее представление транслируемой программы в виде псевдокода, основную функцию семантического анализа можно определить так. Для каждой операции виртуальной машины требуется:

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 прямо в процессе формирования операций псевдокода.

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

4) Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample7):

4.1) Расширить ранее разработанную грамматику путем включения в нее действий для выполнения статических семантических проверок, соответствующих заданному варианту курсовой работы.

4.2) Реализовать преобразование постфиксной записи транслируемой программы в псевдокод в формате, заданном вариантом курсовой работы.

4.3) Построить компилятор с учебного языка на псевдокод, убедиться в его работоспособности на тестовых примерах программ на учебном языке.

4.4) Подготовить, сдать и защитить отчет к лабораторной работе.

5) Требования к содержанию отчета.

Отчет должен содержать:

- цель работы;
- краткое изложение задач семантического анализа;
- описание структур данных и алгоритмов совокупности действий, разработанных для реализации семантического анализатора по заданному варианту курсовой работы;
- результаты тестирования разработанного транслятора в виде связанного описания фрагментов ПФЗ и полученных из нее фрагментов псевдокода;
- выводы и заключение.

6) Контрольные вопросы

6.1) В чем состоят функции контроля структуры программы?

6.2) Перечислите известные Вам способы представления типов данных.

6.3) Что такое тетрада?

6.4) Опишите этапы алгоритма преобразования постфиксной записи в последовательность тетрад.

6.5) 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:

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

3.1. Базовые типы данных

Перечень базовых типов и их свойства, как правило, полностью определяются стандартом языка программирования, хотя некоторые детали могут определяться аппаратной платформой и конкретной реализацией транслятора. Количество базовых типов данных в любом языке программирования обычно очень ограничено. Так, например, в языке C существует всего три базовых типа данных – символьный (*char*), целый (*int*) и вещественный (*float*). Для целого типа определены три разновидности, не обязательно различающиеся по диапазону значений: короткий (*short*), обычный и длинный (*long*), для вещественного – две разновидности: обычный и двойной точности (*double*). Кроме того, символьные и целые значения могут быть либо знаковыми (*signed*, по умолчанию), либо беззнаковыми (*unsigned*).

В языке Java к аналогичному (только нет беззнаковых) набору базовых типов данных добавлен булевский тип (*boolean*). В языке Pascal также существует булевский тип данных, но целые и вещественные типы не имеют разновидностей по размеру значений.

Под типом элемента данных принято понимать:

- с одной стороны, его внутреннее устройство (диапазон возможных значений, размер области памяти в минимально адресуемых единицах, необходимой для хранения значения, формат значения, т. е. назначение каждой двоичной цифры – бита и т. д.);
- с другой стороны, перечень и смысл операций, которые могут применяться к значениям этого типа.

Большинство деталей внутреннего устройства данных (за исключением диапазона значений) обычно скрывается от программистов. Для построения транслятора (и понимания принципов его работы), наоборот, очень важны детали внутреннего строения данных. Именно с ними имеют дело процессы семантического анализа, генерации объектного кода и его оптимизации.

Внутреннее устройство объектов может быть как очень простым, так и чрезвычайно сложным. Например, объект символьного типа в языке C мож но считать одним из простейших. Внутреннее представление значения такого

объекта занимает минимальный адресуемый участок памяти – один байт. Диапазон значений внутреннего представления – от 0 до 255 (в десятичной системе счисления). В операциях символьное значение рассматривается как целое число без знака.

Другой пример – функция в языке С. Функция есть программная единица, возвращающая значение некоторого типа (на данный момент будем считать, что функция возвращает значение одного из базовых типов). Имя функции может быть использовано в выражении, следовательно, она является объектом программы. Каждая конкретная функция имеет свой собственный уникальный тип, внутреннее устройство которого в самых общих чертах можно описать следующим образом.

Функция – это как минимум одна область последовательно расположенных элементов памяти, содержащая исполняемые команды и отдельно расположенные области памяти для хранения адресов связи с другими функциями, значений аргументов, значений локальных переменных, значения результата. Имена функций (с фактическими аргументами) могут быть использованы в выражениях таким образом, что может возникнуть впечатление, будто к ним (функциям) применимы арифметические или иные операции. На самом деле эти операции применяются к значениям, возвращаемым объектом типа «функция». К самим же функциям, как к объектам, применима только операция вызова, т. е. операция передачи управления.

Информация о внутреннем устройстве объектов программы нужна транслятору для определения того, как именно должны использоваться значения объектов. Пусть, например, имеется выражение $x + y$.

Вычисление значения этого выражения может протекать по-разному не только для разных сочетаний типов данных в одном языке программирования, но и в том случае, если типы данных объектов x и y одинаковы, но это выражение появляется в программах на разных языках программирования. Например, если объекты x и y имеют символьный тип, то в программе на языке С/С++ будет выполняться арифметическое сложение численных эквивалентов текущих значений символов с образованием целого значения в качестве результата, а в программе на языке Object Pascal – конкатенация этих символов с образованием значения типа «массив символов» или «строка». Таким образом, результатом вычисления выражения *'0'+'A'* в программе на языке С будет целое беззнаковое число 113 (которое можно рассматривать и как символ *'q'*), а в программе на языке Object Pascal – строка *"0A"* (последовательность из двух символов *'0'* и *'A'*).

Однако знания только внутреннего строения типов данных недостаточно, для того чтобы проверять правильность программы, в которой используются объекты этих типов. Для каждого типа должен быть известен перечень операций, применимых к его значениям. Так, например, к объектам символьного типа в языке С могут применяться операции присваивания, сравнения, арифметические операции, логические (битовые) операции, но не

может применяться операция передачи управления на этот символ. В языке Pascal к данным символьного типа не могут применяться битовые и арифметические операции, а могут только операции присваивания и сравнения. К функции, наоборот, может быть применена операция передачи управления (с предварительным сохранением адреса возврата), но не может применяться ни одна арифметическая, логическая или сравнивающая операция (как уже было сказано, не следует путать операции с функцией как объектом программы и операции со значением, возвращаемым ею в результате вызова).

Базовые типы данных определены стандартом языка во всех деталях, т. е. разработчику транслятора заранее известно их внутреннее устройство, множество применимых к ним операций, в том числе операций преобразования в другие базовые типы, преобразования из внешнего во внутреннее представление, и наоборот, из внутреннего во внешнее. Базовыми типами данных могут обладать так называемые простые переменные и константы.

Константы бывают именованными и литеральными.

Именованные константы с точки зрения решения задач семантического анализа почти полностью эквивалентны переменным. Единственное отличие состоит в том, что к именованной константе неприменима операция присваивания. Способы объявления именованных констант в разных языках различны.

Литеральными константами (или просто литералами) называются объекты, не имеющие имени и объявленные просто в виде их значений прямо в тексте инструкции.

С литеральными константами связано несколько проблем, которые могут разными способами решаться разработчиками трансляторов:

1. Являются ли несколько текстуально одинаковых литеральных констант, встречающихся в разных точках текста программы, одним объектом времени исполнения, или каждая такая константа есть самостоятельный объект, занимающий собственный элемент памяти?
2. В какой момент времени (на каком этапе трансляции) должно осуществляться отнесение каждой встреченной в тексте программы константы к тому или иному типу данных и соответственно преобразование литерала (текстового представления константы) во внутреннее представление, т. е. значение?

Единых ответов на эти вопросы не существует. От того, как разработчик языка отвечает на них, существенно зависят свойства языка, а также характеристики программ на нем. Проиллюстрируем существо этих проблем на небольшом примере. Пусть в программе на языке C встречается такая последовательность операторов, показанная на рис. 8.1.

(1) *int val;*
(2) *double values[10];*
(3) *double * pointer;*

(4) *val=1;*
(5) *pointer=values;*
(6) **pointer=1;*
(7) *pointer+=1;*

Рис. 8.1. *Литеральные константы в языке С*

В этой последовательности появляются три литеральные константы с идентичным текстовым представлением 1. Казалось бы, после обработки текста лексическим анализатором, обнаружившим три вхождения целочисленной константы 1, транслятор должен каждое из этих вхождений связать с одним и тем же объектом.

Однако семантические правила языка С таковы, что в операторе присваивания

(4) *val=1;*

должно использоваться целое значение 1, результатом выполнения оператора

(5) **pointer=1;*

в конечном итоге должна быть запись вещественного значения 1.0 (заметим, что внутреннее двоичное представление вещественной единицы может не совпадать с представлением целой единицы) в самый первый элемент массива *values*, а фактическое значение литеральной константы 1 в строке (7) зависит от реализации транслятора (и от аппаратной платформы) и может оказаться равным 8 или 10.

Этот простой пример показывает, что (по крайней мере, в некоторых языках) установление типа данных литеральных констант и формирование внутреннего представления их значений не может выполняться на этапах лексического или синтаксического анализа. Оно должно выполняться семантическим анализатором позднее.

3.2. Производные типы данных

Производные типы конструируются программистом по правилам, определенным стандартом языка. Для разных языков эти правила различаются.

В силу того, что возможность и степень удобства конструирования в точности таких типов, которые необходимы для решения каждой конкретной задачи, чрезвычайно важны при программировании, именно средства, предоставляемые языком для этих целей, во многом определяют как потенциальную применимость языка, так и его популярность. В зависимости от того, что понимается под типом данных, существует несколько принципиально различающихся возможностей для определения производных типов (здесь перечислены только некоторые, наиболее часто употребляемые способы).

1. Модификация тех или иных параметров внутреннего устройства базового типа. Например, в языке С существует возможность определения

коротких (short) и длинных (long), знаковых (signed) и беззнаковых (unsigned) вариантов обычного целого типа.

2. Образование однородных (содержащих элементы одного и того же типа) массивов требуемых размерности и границ по каждому измерению. Идентификация элементов массива для доступа к их значениям осуществляется с помощью указания целочисленных индексов по каждому измерению. К элементам массивов обычно оказываются применимы в точности те же самые операции, что и к одиночным элементам данных того же самого типа. К массивам в целом обычно оказываются применимыми только передача в качестве аргумента процедуры или функции, возврат в качестве значения функции и (не во всех языках программирования) присваивание.

3. Определение неоднородных совокупностей из данных разного типа. Примеры таких совокупностей – записи (record) в языке Pascal, структуры (struct) в языке С. Идентификация элементов таких совокупностей для доступа к их значениям обычно осуществляется по составному имени, включающему имя совокупности и имя элемента. Перечень операций, применимых к элементам, полностью определяется их типами. К записям (структурам) в целом обычно оказываются применимы такие же операции, что и к массивам (передача в качестве аргументов, возврат в качестве результата и присваивание).

4. Определение процедур/функций. Это объекты, к которым могут применяться операции вызова с передачей им аргументов требуемых типов и/или операции передачи их в качестве аргументов других процедур/функций. С другой точки зрения процедуры/функции можно рассматривать как новые операции, определяемые программистом и применяемые к значениям их аргументов.

5. Объектно-ориентированное программирование. Это развитие способа 4 в сочетании со способом 3 (структуры и записи), которое привело к образованию ряда базовых понятий: классов, их свойств и методов. Свойство экземпляра класса, выглядящее в тексте программы как составное имя элемента данных, реализуется путем создания двух одноименных функций, одна из которых предназначена для извлечения значения свойства (аналогичное понятие – r-value), а другая – для присваивания ему значения (аналог – l-value). В зависимости от того, в каком контексте употребляется имя свойства, транслятор подставляет вместо него вызов либо той, либо другой функции. Методы, более похожие на обычные функции, стали основой для определения и переопределения операций, применяемых к экземплярам классов (например, операции >> и << классов cin и cout, определенных в языке С++ для ввода и вывода данных).

6. Создание указателя на существующий тип данных. Не в каждом языке допускается явное программирование операций с адресами как со значениями. Как уже упоминалось, в языках, не предназначенных для системного программирования, адресная арифметика обычно полностью скрывается от программиста. Если же разрешается образовывать

указательные типы данных, как правило, к ним допускается применять только операции присваивания.

И только в языках, предназначенных в основном для написания системных программ (например, в языке С), к указательным значениям оказывается возможным применение некоторых арифметических операций (сложение и вычитание с целым значением) и операций сравнения.

Для каждого из способов конструирования производных типов данных в разных языках предусматриваются различные моменты связываний атрибутов объектов с самими объектами. Выбор момента выполнения связывания, как уже было отмечено, влияет как на мощность изобразительных средств языка, так и на скорость работы программ, разработанных средствами данного языка. Независимо от того, к какому типу – базовому или производному – относятся объекты программы, для каждой операции с некоторым объектом, обнаруженной в ее тексте, семантический анализатор транслятора должен иметь возможность выявления как внутреннего устройства объекта, так и применимости данной операции к его значению. Это делается на основе явных или неявных объявлений типов объектов программы и обычно называется контролем типов данных.

3.3. Контроль типов данных объектов программы

В различных языках программирования реализован широкий спектр подходов к контролю типов данных.

На одном конце спектра находятся такие языки программирования (например, язык Perl), в которых не требуется явно указывать типы используемых в программе объектов (или явное определение типов опционально, как в языке Visual Basic). В этом случае тип объекта может стать известным (определенным) только после первого присваивания ему значения во время исполнения программы. До этого присваивания тип объекта просто не определен. Последующие во времени операции с объектом могут приводить к изменению его типа. Ясно, что в таком случае семантические проверки возможности выполнения операций над значениями объектов могут быть произведены только во время исполнения программы, поскольку типы данных операндов всех или некоторых операций не известны во время трансляции. При интерпретации эти проверки будут выполняться транслятором или виртуальной машиной. Если реализуется компиляция, то семантический анализатор должен встраивать в программу дополнительные действия для проверки возможности исполнения операций над объектами неизвестного типа, построенных по тексту транслируемой программы.

Такие действия обычно называются динамическими проверками в отличие от статических проверок, которые выполняются во время трансляции программы. Динамические проверки могут значительно увеличивать время исполнения программы независимо от того, интерпретируется она или компилируется. Необходимо отметить, что этот подход, т.е. потенциальная возможность изменения типов объектов

программы «на лету», постоянно подвергается жесткой критике вследствие крайней затруднительности разработки безопасных программ (программ, не содержащих трудно обнаруживаемые ошибки). Тем не менее основанные на таком подходе языки программирования существуют и продолжают развиваться.

Другой крайней точкой рассматриваемого спектра подходов к контролю типов данных является так называемая строгая (сильная) типизация. Под строгой типизацией понимается выполнение следующей совокупности требований к программе:

- для каждого используемого в программе типа должны быть известны множество значений (т. е. внутреннее устройство) и множество применимых к ним операций;
- каждый объект программы должен иметь однажды определенный и неизменяемый тип;
- при выполнении любого присваивания значения объекту тип присваиваемого значения должен быть эквивалентен типу объекта;
- использование значения объекта допускается только в качестве операнда операций, допустимых для данного типа.

К строгой типизации стремились разработчики языков Pascal, Ada, Java и ряда других.

В случае строгой типизации подавляющая часть полного набора семантических проверок возможности применения операций к объектам может быть выполнена в процессе трансляции программы, т. е. статически. К сожалению, абсолютно все проверки реализовать во время трансляции нельзя, достаточно вспомнить операцию деления на неизвестное транслятору значение, которым при исполнении программы может оказаться ноль. Первичная информация для проверок извлекается семантическим анализатором из операторов объявления данных и сохраняется в таблице идентификаторов в качестве атрибутов объектов. Как часть общего текста программы эти операторы вначале обрабатываются лексическим и синтаксическим анализаторами и могут преобразовываться в постфиксную форму записи.

Большое количество языков программирования, в том числе такие популярные (по крайней мере, в свое время), как Algol-60, Fortran, C и многие другие, занимают то или иное промежуточное положение в этом спектре, предоставляя, с одной стороны, полную информацию о типах и возможность (но необязательность в Fortran'е) явного единственного объявления типа каждого объекта в рамках одной программной единицы, а с другой – нарушение двух последних требований строгой типизации. Так, например, в каждом из перечисленных языков допускается возможность присваивания значения одного типа объекту другого типа с подразумеваемым, т. е. неявно выполняемым, преобразованием. В языках Fortran и C существует возможность выполнения операций, не применимых к значениям данного типа за счет неконтролируемой подстановки – присваивания значения объекту другого типа (такие возможности

предоставляют объявления common, equivalence в Fortran и union в C). Любое нарушение правил строгой типизации – потенциальная причина появления в программе ошибок, которые иногда чрезвычайно трудно обнаружить. Именно за это языки, не гарантирующие строгую типизацию, часто подвергаются суровой критике. Вместе с тем требования строгой типизации заметно сужают область применения языка программирования. Системные программы, такие как операционные системы и их утилиты, очень трудно разрабатывать, соблюдая абсолютно все требования строгой типизации.

Явные объявления типов данных, с одной стороны, позволяют обеспечивать более или менее строгий контроль применимости знаков операций к операндам во время трансляции, с другой – порождают ряд проблем, которые так или иначе должны быть разрешены разработчиками языка программирования и трансляторов для него. Потенциальная возможность неоднократного (намеренного или случайного) объявления типа данных для одного и того же наименования порождает первую проблему: разрешать такую возможность или запрещать ее. Запрет множественных объявлений разных объектов с одним именем в пределах одной транслируемой программы в принципе возможен, но обычно считается слишком жестким ограничением и практически не применяется. В том случае, когда такие объявления разрешаются стандартами языка, ими же должны быть определены правила, позволяющие любое использование наименования в тексте программы отождествить с единственным объектом (либо обнаружить и диагностировать ошибку). Такие правила можно определить на основе понятия ассоциации и использовать их во время трансляции для определения того, к какому именно объекту будет применяться операция во время исполнения программы.

Исполнение программы есть последовательное исполнение операций, преобразующих значения своих операндов в значения, которые будут обрабатываться другими операциями. Для любой операции не позже чем к моменту ее фактического исполнения должны быть установлены действительная возможность ее применения к значениям операндов и конкретный способ обработки значений. Для примера вспомним, что складывать целые числа, целое и вещественное, целое и указательное и т. д. нужно по-разному. Выражение a+b должно быть преобразовано в разные машинные команды для различных сочетаний типов данных a и b. Для выявления смысла каждой операции в общем случае нужно выполнить такую последовательность действий:

– определить количество операндов (напомним, что одно и то же слово может применяться для разных операций: скажем, знак «–» обычно используется и для обозначения бинарной операции вычитания и для обозначения унарной операции изменения знака числа);

– определить тип каждого операнда;

– проверить, допустим ли этот тип к данной позиции операнда в операции путем сравнения типов;

– на основании результатов всех проверок принять решение, может ли быть выполнена данная операция и если может, то как.

Предполагая, что задачи определения количества операндов и их типов решены (первая может считаться тривиальной, а вторая обычно решается путем выборки атрибутов объекта из таблицы идентификаторов, см. раздел 4.8), сосредоточимся на способах решения задачи сравнения типов. Под сравнением будем понимать только установление эквивалентности или неэквивалентности двух типов.

3.4. Эквивалентность типов данных

Для описания типа языковой конструкции будем использовать понятие «выражение типа». Выражение типа, как и любое выражение, имеет вычисляемое значение, которым может являться либо базовый, либо производный тип. В том случае если тип является производным, значение выражения типа должно быть построено с помощью применения оператора, называемого конструктором типа, к другим выражениям типа. Множества базовых типов и конструкторов производных типов зависят от проверяемого языка.

В некоторых языках программирования типам могут даваться имена. Например, во фрагменте программы на языке Pascal, представленном на рис. 8.2, идентификатор *link* объявлен в качестве имени типа *^cell* (указатель на ячейку).

```
(1)    type link = ^cell; var next : link;
(2)    last : link;
(3)    p : ^cell; q, r :^cell;
```

Рис. 8.2. Имена типов в языке Pascal

Для реализации последующих операций важно знать, идентичны ли типы переменных *next*, *last*, *p*, *q* и *r*? Для реализации правил проверки очень важно иметь точное определение равенства двух типов. Потенциальная неоднозначность возникает с именами выражений типа, которые затем используются в последующих выражениях типа. Ключевой вопрос обычно состоит в том, является ли имя в выражении типа само по себе выражением или сокращением для другого выражения типа.

Для обеспечения высокой эффективности работы транслятора требуется использовать такие представления выражений типа, которые позволяют быстро определять, являются ли сравниваемые типы эквивалентными. Представление типов может быть либо именным, либо структурным, либо кодированным.

- Именное представление и сравнение типов

Именная эквивалентность проверяется путем:

- извлечения текстового представления имени типа из программы или формирования его путем применения конструктора типа;

- приведения этого представления к стандартному виду (замена возможных сокращений, удаление незначащих пробелов, добавление всех возможных скобок и т.д.);
- сравнения двух текстовых представлений как обычных строк.

При применении этого способа к фрагменту программы на рис. 8.2 будут получены имена выражений типов переменных, приведенные в табл. 8.1.

Таблица 8.1

nex	*link*
last	*link*
p	***pointer**(cell*
q	***pointer**(cell*
r	***pointer**(cell*

С точки зрения эквивалентности имен переменные *next* и *last* имеют один и тот же тип, поскольку с ними связаны одинаковые выражения типа. Типы переменных *p*, *q* и *r* также эквиваленты, но, например, переменные *p* и *next* разнотипны, поскольку связанные с ними имена выражений типа различны.

При именном представлении каждое уникальное имя представляет собственный тип, отличный от любого типа с другим именем.

- Структурное представление и сравнение типов

Концепция структурной эквивалентности основывается на представлении выражений типа в виде графов с листьями для базовых типов и внутренними узлами для конструкторов типов. При этом рекурсивно определенные типы приводят к появлению циклов в таком графе. Сравнивая структурное и именное представления, можно считать, что в графе имена типов заменяются выражениями типа, определяемыми этими именами. Следовательно, два выражения типа структурно эквивалентны, если представить себе, что в их текстовых представлениях все имена развернуты вплоть до базовых типов.

На рис. 8.3 показан пример фрагмента графа структурного представления типов.

С точки зрения структурной эквивалентности все пять переменных в рассматриваемом примере имеют один и тот же тип, поскольку *link* представляет собой не что иное, как имя для выражения типа ***pointer**(cell)*.

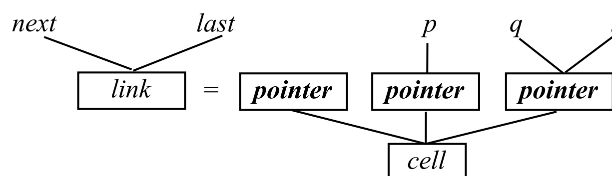


Рис. 8.3. Связь переменных и узлов в графе типов

Типичная реализация функций контроля типов состоит в построении для представления типов соответствующего графа. Всякий раз, когда в объявлении объекта встречается конструктор типа или имя типа, создается новый узел, а при появлении имени базового типа – новый лист. При использовании такого представления два выражения типа считаются эквивалентными, если они представлены одним и тем же узлом в графе типов.

Многие полезные структуры данных, такие как связанные списки и деревья, зачастую определяются в программах рекурсивно, например связанный список либо пуст, либо состоит из ячейки с указателем на связанный список. Такие структуры данных обычно реализуются с использованием записей, содержащих указатели на точно такие же записи. Это создает определенные проблемы для решения задачи проверки эквивалентности типов структурными методами. Заметим, что именно поэтому при именном представлении типов не делается попыток развернуть все имена вплоть до базовых типов, поскольку применение такого развертывания к рекурсивно определенным типам привело бы к зацикливанию и бесконечному росту текстового представления.

Структурная эквивалентность типов в направленных ациклических графах может быть проверена с использованием рекурсивного алгоритма, параметрами которого являются две заданные вершины A и B:

- если A и B есть один и тот же лист (базовый тип), то вернуть истину,
- иначе, если и A и B есть конструкторы, последовательно обойти все пары дуг, исходящие из A и B, и:
  - если найдется хотя бы одна непарная дуга, вернуть ложь;
  - или, если хотя бы одна пара вершин, в которые ведут эти дуги, не эквивалентна (для каждой такой пары применяется этот же алгоритм), вернуть ложь;
  - иначе вернуть истину.

Свойства языка программирования сильно зависят от того, какие параметры конструкторов типов считаются важными при формировании направленного ациклического графа. Например, если (как в языке Pascal) границы изменения индексов по каждому измерению однородных массивов считаются частью конструктора типа, то тип массива из двух целочисленных элементов не будет эквивалентен типу массива, содержащего три целых числа и т.д. В результате для программистов практически непреодолимой трудностью была разработка на этом языке процедур/функций для обработки массивов с заранее неизвестным количеством элементов или измерений.

- Кодирование выражений типа

В некоторых реализациях трансляторов для выражений типов стремятся найти значительно более компактную по сравнению с графом запись. Информация о выражении типа может быть закодирована последовательностью битов, а значит, может интерпретироваться как целое число. Кодирование осуществляется таким образом, что различные числа представляют структурно неэквивалентные выражения типов. Этот подход может использоваться для существенного ускорения проверки структурной эквивалентности и в более сложных языках, в которых полная проверка эквивалентности типов не может быть осуществлена путем кодированного представления. Ускорение может быть достигнуто за счет того, что сначала проверяется неэквивалентность путем сравнения целых (но не полных для

некоторых конструкторов) представлений типов, и только в случае их равенства применяется приведенный выше алгоритм структурной проверки.

Например, в первом трансляторе с языка С [6] базовые типы кодируются с использованием четырех битов. Два младших бита обозначают базовые типы: 00 – *char*, 01 и 10 – *integer,* 11 – *real*. Следующие два бита (которые маскируются при большинстве проверок эквивалентности) используются для хранения значений модификаторов длины (*short* или *long*) и знака (*signed*/*unsigned*).

Для конструкторов типов (в этом примере рассматриваются только указатели, массивы и функции) используются каждые два следующих бита, обозначающие:

00 – отсутствие конструктора;

01 – указатель (*pointer)* на тип *t*, закодированный левее;

10 – массив (*array*) неопределенной длины элементов типа *t*;

11 – функция, возвращающая объект типа *t* (*freturns*).

Размерности и границы изменения индексов массивов, как и количество, и типы аргументов функций, совершенно не учитываются в таком кодировании, их приходится хранить вне кода типа (скорее всего – в таблице идентификаторов, в которой хранится и обсуждаемый код типа).

Таким образом, объекты, структурно эквивалентные с точки зрения целочисленного кодирования, могут быть неэквивалентны с точки зрения полной проверки эквивалентности. Поскольку каждый из этих конструкторов представляет собой унарный оператор, выражения типа, образованные применением этих конструкторов к базовым типам, имеют весьма однородную структуру. Приведем (рис. 8.4.) примеры выражений типа в тексте программы (первый столбец), во внутреннем представлении транслятора (второй столбец) и соответствующих им кодов (третий столбец):

int	*int*	000000 0001
int function(…)	*freturns(int)*	000011 0001
**(int function(…))*	*pointer(freturns(int))*	000111 0001
((int function(…)))[…]*	*array(pointer(freturns(int))*	100111 0001

Рис. 8.4. Примеры кодированных выражений типа

Такое представление является чрезвычайно компактным (по сравнению и с именным, и со структурным) и отслеживает последовательность применения конструкторов, появляющихся в любом выражении типа. Две различные последовательности битов не могут представлять один и тот же тип, поскольку при этом различны либо базовые типы, либо примененная к ним последовательность конструкторов. Тем не менее разные типы могут иметь одну и ту же последовательность битов – в силу того, например, что размеры массивов и аргументы функций не представлены в данной системе кодирования.

Кодирование из этого примера в принципе может быть расширено для включения типов записей и объединений (*struct* и *union*). При этом каждая

запись при кодировании рассматривается как базовый тип; но отдельная последовательность битов кодирует тип каждого поля записи.

При использовании кодированного представления типов задача выявления эквивалентности резко упрощается, так как вместо сравнения имен типов (с предварительным приведением к каноническому представлению) или рекурсивной обработки вершин графа надо всего лишь сравнивать числа, поставленные в соответствие именам типов.

3.5. Среды ссылок периода исполнения

Вся совокупность объектов, значения которых доступны из произвольной точки текста программы, в литературе называется средой ссылок. Различают два понятия – текстуальная среда ссылок (или среда ссылок периода трансляции; иногда ее называют статической, или лексической) и динамическая, или среда ссылок периода исполнения программы. Задача транслятора – спроецировать текстуальную среду ссылок на среду ссылок периода исполнения.

Строение среды ссылок периода исполнения и способ отображения на нее текстуальной среды определяется тем, как разработчики языка отвечают на следующие вопросы:

1. Допускаются ли рекурсивные вызовы функций?
2. В какой момент создаются локальные объекты функций?
3. Что происходит с локальными переменными при возвращении управления из функций?
4. Может ли функция обращаться к нелокальным объектам, не являющимся ее фактическими аргументами?
5. Каким образом в функцию передаются параметры?
6. Может ли функция быть передана в качестве параметра?
7. Может ли функция быть возвращена в качестве результата?
8. Может ли память выделяться динамически, по явному запросу программы?
9. Должна ли динамически выделенная память освобождаться также по явному запросу?

Возможный спектр ответов на эти очень широк. В последующих разделах будут рассматриваться в основном типичные решения и вкратце обсуждаться последствия других возможных ответов на данные вопросы.

Среду ссылок периода исполнения будем рассматривать применительно к компиляторам. Интерпретаторы моделируют программно те же самые процессы, которые реализуются аппаратно-программным способом при исполнении скомпилированной программы.

Будем считать, что для исполнения программы операционная системы компьютера выделяет один связный блок элементов памяти. Способ выделения этого блока (будем называть его памятью задачи) и его отображения на физическую память компьютера может быть различен для разных операционных систем и тем более – для разных аппаратных платформ. Если программа запускается неоднократно, то вполне вероятно,

что при разных запусках место расположения памяти задачи в физической памяти компьютера будет различным. Ясно, что в этих условиях транслятору не может быть известно будущее местоположение программы в памяти. Известными могут быть:

- максимально возможный размер памяти задачи;
- ограничения на использование некоторых фрагментов этой памяти, занятых модулями поддержки периода исполнения;
- аппаратные особенности (направление роста аппаратного стека, количество байт, извлекаемых из памяти за одно обращение и, соответственно, рекомендации по выравниванию объектов, и т.д.).

Некоторые возможные варианты внутреннего устройства памяти задачи для разных программно-аппаратных платформ представлены на рис. 8.5 (предполагается, что адреса памяти возрастают по направлению сверху вниз).

Здесь под функциями понимается совокупность исполняемых инструкций, под областью подгрузки – фрагмент памяти, выделенный для динамически подгружаемых в процессе исполнения программы библиотек (тоже совокупностей функций), под кучей – специально организованная область памяти, блоки которой выделяются/освобождаются динамически по явным запросам из программы. Статические данные – совокупность объектов, существующих в течение всего времени работы программы (в некоторых языках в явном виде возможность использования статических данных может отсутствовать). Аппаратно реализуемый стек используется многопланово. В частности, именно в стеке, как правило, размещаются локальные данные функций.

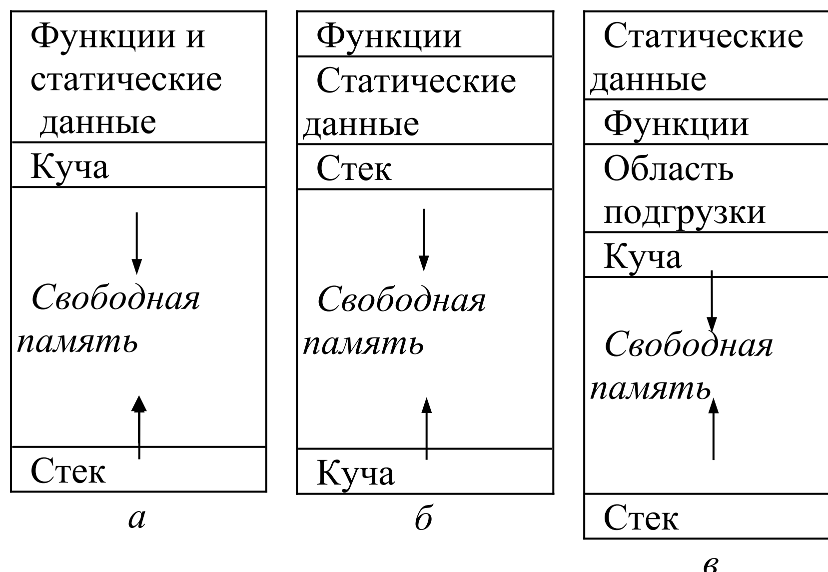


Рис. 8.5. Варианты размещения совокупностей объектов в памяти задачи

Организация памяти образа задачи, представленная на рис. 8.5 (все варианты), предполагает, что память времени исполнения состоит из единого непрерывного блока, выделенного программе при запуске. При этом суммарный размер стека и кучи, обычно растущих навстречу друг другу в

процессе исполнения программы, должен быть достаточно большим, чтобы эти две области никогда не пересекались. Столкновение границ стека и кучи (т.е. пересечение этих областей) обычно приводит к аварийному завершению программы, поскольку для ее выполнения оказалось недостаточно отведенной системой памяти. Для обнаружения такой ситуации предусматриваются программные и/или аппаратные средства контроля значений границ стека и кучи.

В этих условиях при трансляции логично и естественно считать, что адресация памяти задачи для транслятора начинается с нуля и что процессор компьютера обеспечивает эффективное отображение такой памяти на любой предоставленный операционной системой блок. Именно на такие потребности запуска и исполнения программ и ориентированы базируемые и индексируемые способы адресации памяти современных процессоров, обеспечивающие «на лету» модификацию адресов, сформированных трансляторами, в реальные адреса памяти. Заметим, что термин «реальные» здесь относится только к адресам памяти внутри образа выполняемой программы. Фактическая система адресации памяти может быть значительно сложнее и предусматривать страничную, сегментную либо странично-сегментную организацию виртуальной памяти, при которой последовательно расположенным областям (страницам или сегментам) образа памяти ставятся в соответствие участки физической памяти компьютера, выделяемые операционной системой по мере необходимости и возможности. При этом отнюдь не обязательно весь образ программы в течение всего времени ее исполнения присутствует в оперативной памяти. Делается это «прозрачным» для исполняемых программ образом и поэтому на логику работы трансляторов влияния не оказывает.

В том случае, когда язык программирования сконструирован для совместной трансляции, компилятор может распоряжаться всеми известными ему в пределах памяти задачи свободными участками по собственному усмотрению (естественно, с учетом ограничений и особенностей аппаратной платформы) для размещения всей совокупности объектов программы: функций и обрабатываемых ими данных. Задачи компилятора при раздельной трансляции могут показаться несколько более сложными в связи с тем, что в этом случае ему уже не может быть известным будущее размещение транслируемых частей единой программы внутри памяти задачи.

В действительности все сложности по размещению раздельно транслируемых частей программы в одной линейно организованной памяти задачи выпадают на долю других компонент программного обеспечения: либо редактора связей (другие названия: линковщик, компоновщик, сборщик, …), либо загрузчика операционной системы. Компилятор же, как и в случае совместной трансляции, обеспечивает размещение объектов транслируемых частей программы в линейном образе памяти.

Отметим, что размеры сгенерированных при трансляции последовательностей инструкций каждой функции становятся известны во время компиляции, так что компилятор может разместить их в образе памяти

задачи при генерации объектного кода. Под размещением здесь пока что понимается просто связывание инструкций и адресов элементов памяти задачи, отводимых компилятором для хранения этих инструкций.

Аналогично инструкциям формируются и связываются с памятью и такие объекты данных, которые должны в единственном экземпляре существовать в течение всего периода исполнения программы. Совокупность таких объектов названа статическими данными. Одной из причин стремления к статическому выделению памяти для как можно большего количества объектов данных является то, что адреса этих объектов могут быть использованы прямо при компиляции для формирования использующих эти объекты инструкций. Ниже мы увидим, что для преобразования операторов исходной программы, работающих не со статическими объектами, в инструкции целевой машины приходится формировать дополнительные структуры данных и специальные последовательности машинных команд для их создания и обработки. Это приводит к появлению накладных расходов как памяти, так и затрат процессорного времени при исполнении программы. В некоторых языках, в частности в языке Fortran, память для всех объектов данных может быть выделена статически, т.е. доля таких накладных расходов равна нулю. Однако цена этого «преимущества» – запрет рекурсивных вызовов функции – может оказаться слишком большой.

В тех языках, которые предполагают возможность рекурсивных вызовов функций, неизбежно появляются и не статические данные – локальные данные активаций функций и некоторые другие объекты, необходимые для реализации доступа из функций к не принадлежащим им данным. Понятие активации функции необходимо рассмотреть во всех деталях.

3.6. Активация функции

Активацией функции называется процесс ее выполнения в результате одного конкретного вызова. Этот процесс включает следующие действия (возможно, не все из перечисленных ниже и не обязательно выполняемые в приведенном порядке):

1. Формирование адреса точки входа в вызываемую функцию (в том случае, если функция является частью динамически загружаемой библиотеки и этой библиотеки еще нет в памяти выполняемой программы, может выполняться предварительное обращение к операционной системе с целью ее подгрузки).
2. Формирование значений, которые должна обработать вызываемая функция, – фактических параметров. Обеспечение доступности этих значений для инструкций вызываемой функции.
3. Формирование и сохранение адреса возврата – адреса той инструкции вызывающей функции, которая должна получить управление в момент завершения выполнения вызываемой функции.
4. Обеспечение возможности доступа к значениям локальных объектов текущей активации вызывающей функции (и, возможно, функций,

находящихся еще раньше в цепочке вызовов) из вызываемой функции, если такая возможность предусмотрена языком программирования.

5. Сохранение содержимого регистров процессора, необходимых для восстановления правильного выполнения текущей активации вызывающей функции.
6. Собственно вызов функции, т.е. передача управления в ее точку входа.
7. Создание локальных объектов (выделение памяти для их размещения).
8. Выполнение функции, т.е. обработка значений фактических параметров, протекающая путем:
   - формирования и использования значений локальных объектов текущей активации;
   - формирования и последующего использования результатов промежуточных вычислений;
   - использования/модификации локальных объектов других существующих активаций этой же или других функции;
   - использования/модификации значений статических данных программы.
9. Формирование результата – значения, возвращаемого из функции. Обеспечение доступности этого значения для инструкций вызывающей функции (как правило, только для той инструкции, к которой возвращается управление и которая должна обработать или сохранить результат вызова).
10. Уничтожение объектов, созданных исключительно для выполнения данной активации функции.
11. Восстановление содержимого ранее сохраненных регистров.
12. Возврат из функции – передача управления в ту точку вызывающей функции, адрес которой сформирован на шаге 3.

Заметим, что в тексте исходной программы (имеются в виду, естественно, языки высокого уровня) в явном виде обычно присутствует только вызов функции и ее описание (определение типов параметров и типа возвращаемого значения) или объявление – то же самое описание, за которым следует тело. Все вышеперечисленные действия подразумеваются, но явно не указываются. Задача транслятора – обеспечить выполнение этих действий. От того, каким образом будут реализовываться эти действия в процессе исполнения программы, зависит и то, как должны выполняться семантические проверки во время трансляции.

Некоторые из описанных выше действий (например, 1–3, 6) могут быть выполнены только инструкциями вызывающей функции. В свою очередь, действия 8, 9, 12 могут быть выполнены только инструкциями вызываемой функции. Остальные действия (4, 5, 7, 10, 11) в принципе могут выполняться инструкциями как вызывающей, так и вызываемой функций. В разных языках (и даже в разных реализациях одного и того же языка) могут приниматься различные решения по отнесению этих действий к вызывающей или вызываемой функции.

3.7. Запись активации функции

Все данные, необходимые только для однократного исполнения функции, обычно размещаются в одном связном блоке памяти и называются записью активации. Запись активации может содержать следующие поля (все или только часть – определяется свойствами языка, его реализации и текущими условиями) (рис. 8.6):

Локальные данные
Результаты промежуточных вычислений
Регистры процессора
Связь по доступу (необязательная)
Связь по управлению (необязательная)
Фактические параметры
Возвращаемое значение

Рис. 8.6. Состав записи активации функции

Для последующего рассмотрения важно знать, в какой момент и каким образом может быть определен размер всей записи активации в целом, а следовательно, размер каждого из ее полей. Поэтому кратко рассмотрим назначение и, соответственно, способ формирования содержимого записи активации.

1. *Локальные данные.* Здесь в течение времени жизни данной активации функции хранятся значения объектов, объявленных в ее теле. Размер этого поля, как правило, может быть вычислен во время трансляции вызываемой функции. Существуют языки программирования, в которых возможно исключение из этого правила, такие как C/C++: допускающие возможность объявления в функции локальных массивов, размеры которых зависят от значений фактических параметров. Их значения желательно размещать в регистрах процессора на время выполнения функции.

2. *Результаты промежуточных вычислений.* В том случае если в инструкциях вызываемой функции имеются сложные выражения наподобие *a*b+c*d*, возникает необходимость временного хранения значения произведения *a*b* (или *c*d,* или и того и другого) от момента его вычисления и, по крайней мере, до момента последнего использования. Такие данные во многом эквивалентны локальным данным функции, но не имеют никакого имени, присвоенного программистом в тексте программы. В процессе трансляции по мере необходимости образования объектов для хранения промежуточных результатов имена для них могут формироваться и использоваться семантическим анализатором и генератором объектного кода/виртуальной машиной. Как будет показано ниже, перечень и типы данных объектов, необходимых для хранения промежуточных результатов, могут быть определены транслятором в процессе преобразования

постфиксной записи в последовательность тетрад/триад. Таким образом, размер этого поля тоже может быть вычислен во время трансляции вызываемой функции.

3. *Регистры процессора.* В этом поле сохраняется состояние регистров процессора, каким оно было непосредственно перед вызовом функции и, соответственно, каким должно быть сразу после возврата из нее. В частности, здесь же обычно хранится адрес возврата, поскольку он является значением одного из регистров, а именно – счетчика команд. Размер памяти, необходимой для сохранения значений регистров, определяется архитектурой компьютера и фиксируется в качестве константы на этапе разработки транслятора.

4. *Связь по доступу.* Используется для обеспечения доступа из вызываемой функции к нелокальным данным, хранящимся в другой записи активации. Размер этого поля фиксирован и равен размеру любого указательного значения.

5. *Связь по управлению.* Это поле используется для хранения указателя на запись активации вызывающей функции. Размер его фиксирован. Используется в реализациях языков, обеспечивающих не текстуальную, а динамическую видимость нелокальных объектов.

6. *Фактические параметры.* Обычно фактические параметры стремятся передавать через регистры процессора. Тем не менее всегда необходимо учитывать возможность того, что количество фактических параметров превысит доступное количество регистров, поэтому в записи активации функции для них должно предусматриваться место. Большинство языков программирования предусматривает фиксированное количество и типы значений параметров каждой функции. В этом случае размер поля фактических параметров может быть определен во время трансляции вызываемой функции. Однако существуют языки (C/C++), допускающие возможность приема вызываемой функцией заранее неопределенного количества фактических параметров. В этом случае размер поля фактических параметров может быть определен только при трансляции текста вызывающей функции.

7. *Поле возвращаемого значения* используется вызываемой функцией для возврата значения вызывающей функции. Для повышения эффективности программы это значение, как правило, возвращается в регистре. Если же для возвращаемого значения резервируется поле в записи активации, то его размер определяется типом значения и может быть легко определен во время трансляции функции.

Не во всех языках используется каждое из этих полей (и не все компиляторы с одного и того же языка делают это одинаково). В силу того что поля записи активации интенсивно используются при выполнении функции, для хранения наиболее часто используемых из них разработчики трансляторов стремятся использовать регистры процессора. За формирование полей записи активации обычно отвечают и вызывающая и вызываемая функции, но каждая – строго за определенные поля записи.

Запись активации может создаваться (здесь имеется в виду выделение памяти для нее, а не формирование значений полей) следующими способами:

1) статически при трансляции функции;
2) динамически в куче;
3) динамически в стеке времени исполнения программы.

Первый способ – создание записи активации в процессе трансляции – преследует цель минимизировать накладные расходы по времени на вызовы функций при исполнении программы. Типичный пример применения – язык Fortran. С каждой функцией связана в точности одна запись активации, поскольку статически невозможно создать большее, но заранее не известное количество записей активации, а создание ограниченного, но большего единицы количества записей никак не решает проблему рекурсивности вызовов, но приводит к росту накладных расходов. При статическом создании записей активации не может быть разрешен рекурсивный вызов функций. Записи активаций функций размещаются транслятором в области статических данных (см. рис. 8.6).

Динамическое создание записей активации в куче во время исполнения программы предполагает наличие механизма управления кучей (библиотеки функций, выполняющих учет свободного пространства и обработки запросов на выделение связного блока требуемого размера и, возможно, освобождение ранее выделенного блока). Вполне вероятно, что куча будет использоваться не только для хранения записей активации, поэтому необходимо обеспечивать учет созданных записей (см. рис. 8.6, поле связи по управлению, которое в данном случае становится обязательным). Следовательно, реализация этого способа приводит к высоким накладным расходам по времени на вызовы, но обеспечивает наибольшую гибкость во взаимодействии вызывающих и вызываемых функций. В частности, при этом способе время жизни активации вызываемой функции в принципе может быть дольше, чем время жизни вызывающей ее активации (см. ниже). Возможна также реализация таких экзотических возможностей, как, например, сопрограммы.

Средняя между предыдущими способами величина накладных расходов в сочетании с наиболее естественной текстуальной видимостью нелокальных данных обеспечивается при динамическом создании записей активации в стеке исполняемой программы. Аппаратная реализация стека минимизирует накладные расходы по времени на выделение памяти для записей активации вызываемых функций. Рекурсивные вызовы функций допускаются, при этом количество записей активации одной и той же функции может быть сколь угодно большим, и ограничиваться только размером области памяти, отведенной под стек программы.

При создании записей активации функций в стеке последовательность их уничтожения обратна последовательности образования. Поэтому время жизни активации любой функции не может быть больше времени жизни активации вызывающей функции. Другими словами, две произвольные активации любых функций (возможно, одной функции) либо не

пересекаются во времени, либо одна из них полностью вложена в другую. На рис. 8.7 показано перемещение потока управления между активациями двух функций во времени.

Внизу сплошной линией со стрелкой обозначена ось времени. Сплошные вертикальные линии показывают либо вызовы функций, либо возвраты из них. Жирными горизонтальными линиями показаны процессы исполнения соответствующих активаций функций. Пунктирными горизонтальными линиями для некоторых активаций выделены интервалы, когда активации существуют, но не исполняются, поскольку в это время процессор занят исполнением другой активации либо той же самой функции (интервалы $t_2 - t_3$ и $t_4 - t_5$ для активации 1 функции *A*), либо другой функции (интервалы $t_1 - t_2$, $t_3 - t_4$, $t_5 - t_6$ и t_6 –… для активации 1 функции *A*, $t_2 - t_3$ и $t_4 - t_5$ для активации 1 функции *B*).

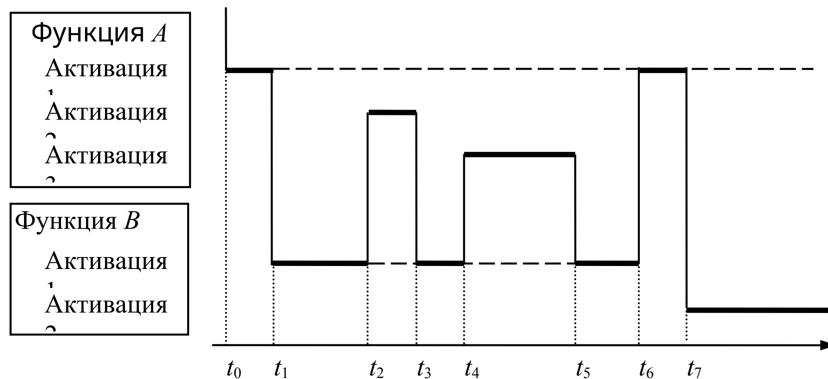
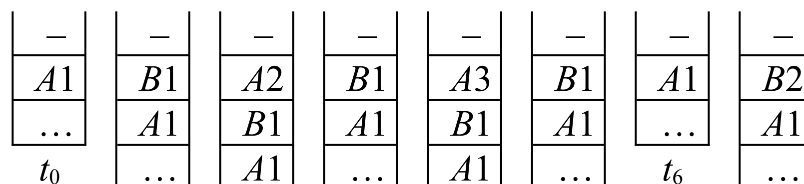


Рис. 8.7. Времена жизни активаций функций

Согласно дисциплине существования элементов стека после создания, к примеру, активации 2 функции *A* и до ее завершения не может завершиться существование ни активации 1 функции *B*, ни активации 1 функции *A*. Заметим, что и то и другое событие в принципе возможно при динамическом создании записей активаций функций в куче. Для реализации этих событий нужно просто обеспечивать явное уничтожение записей ненужных активаций и возвраты по адресам, предварительно взятым из них.

На рис. 8.8. показана последовательность состояний стека программы, соответствующая моментам времени t_0, … t_7. Записи активаций обозначены буквенным названием функции (*A* или *B*) и порядковым номером активации.

Для изображения того, как управление передается активациям и покидает их, может оказаться полезным так называемое дерево активаций. Это графическое представление процесса исполнения совокупности функций, для которого выполняются следующие условия:



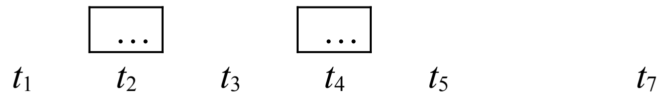


Рис. 8.8. Последовательность состояний стека программы

1. Корень дерева представляет активацию основной программы.
2. Каждый узел представляет активацию функции.
3. Узел *A* является родительским для узла *B* тогда и только тогда, когда поток управления передается из активации А в активацию *B* (узел *B* называется потомком узла *A*).
4. Если два узла *B* и *C* являются потомками одного и того же узла, то узел *B* располагается слева от узла *C* тогда и только тогда, когда время жизни активации *B* начинается раньше времени жизни активации *C*.

Поскольку каждый узел представляет единственную активацию, и наоборот, удобно говорить о том, что управление находится в узле, когда оно находится в активации, представленной этим узлом. Если пометить каждый узел дерева именем (обозначением) функции и порядковым (во времени) номером ее активации, то все узлы дерева будут иметь уникальные пометки. Фрагмент дерева активаций для случая, изображенного на рис. 8.7, может выглядеть так, как показано на рис. 8.9.

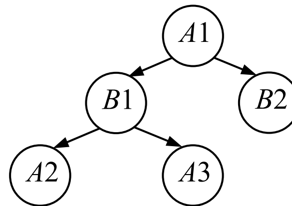


Рис. 8.9. Фрагмент дерева активаций

Последовательность событий создания/уничтожения активаций функций может быть получена путем обхода дерева, начинающегося с корня и заключающегося в последовательном посещении всех поддеревьев каждой вершины строго слева направо. Размещение записей активации функций в стеке исполняемой программы наиболее часто используется в реализациях языков программирования общего назначения.

Независимо от того, где хранятся записи активации в процессе исполнения программы, для любой из них должен быть отведен связный блок памяти такого размера, чтобы в нем размещались все поля записи. Другими словами – к моменту фактического выделения блока памяти под запись активации необходимо точно знать его требуемый размер, который равняется сумме размеров всех полей. Рассмотрим, каким образом определяются размеры полей локальных данных. Способы формирования полей промежуточных вычислений и фактических параметров будут рассмотрены позже. Остальные поля записи активации, как уже упоминалось, имеют фиксированные размеры.

3.8. Локальные данные функций

Для определения размера поля локальных данных транслятор должен разметить (распределить) его образ памяти. Для каждой функции этот процесс начинается с фиксации начального адреса образа памяти поля локальных данных, обычно равного сумме размеров всех предшествующих полей записи активации. Далее последовательно перебираются наименования объектов, ассоциации которых должны быть активными при исполнении инструкций функции. Текущее значение адреса первого свободного элемента (обычно – байта) памяти заносится в соответствующую объекту запись таблицы ассоциаций (обычно это просто таблица идентификаторов), из этой записи извлекается тип объекта, по нему определяется требуемое количество байт памяти и на это количество увеличивается счетчик занятых байт (т.е. указатель первого свободного байта). При генерации объектного кода сформированный таким образом адрес объекта будет использован в качестве смещения в любой команде, оперирующей с этим объектом. При исполнении программы в момент вызова функции адрес записи активации будет занесен в строго определенный регистр процессора (если записи активации размещаются в стеке, то для доступа к ним может использоваться указатель стека).

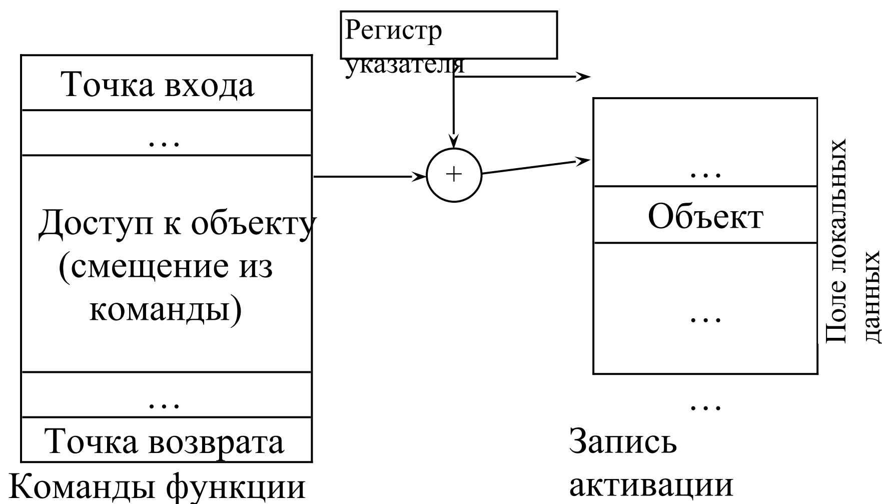


Рис. 8.10. Формирование исполнительного адреса, объект внутри поля локальных данных

Для обеспечения этого транслятор формирует и помещает в точку входа в функцию дополнительные команды. Доступ к значению любого локального объекта осуществляется по адресу, вычисляемому как сумма базового адреса записи активации (из регистра процессора, специально выделяемого для этой цели) со смещением (из команды). Если стек растет в направлении увеличения адресов памяти, то все значения смещений будут отрицательными. Для ситуации, представленной на рис. 8.10, предполагается, что стек растет в сторону уменьшения адресов памяти, поэтому значения смещений положительны. Кажущаяся примитивность процесса формирования смещений локальных объектов осложняется двумя обстоятельствами. Первое состоит в том, что на компоновку памяти для объектов данных сильное влияние оказывают ограничения системы

адресации целевого компьютера и/или требования оптимизации по времени выполнения. Пусть, например, выборка данных из памяти выполняется четырехбайтными кадрами, адрес первого из которых должен быть кратен четырем. В таком случае для доступа к двухбайтному целому значению может потребоваться два обращения к памяти, если его адрес нечетен. Такой случай возможен, если транслятор экономно использует память и размещает данные слитно. Если же при трансляции преследуется цель построения быстрой программы, то определение размера памяти, отводимой под каждый объект, должно производиться с учетом оптимального выравнивания адреса его первого байта.

Второе усложняющее обстоятельство ранее уже упоминалось: если в функции в качестве локального объявлен динамический массив, у которого хотя бы одна граница изменения индекса зависит от значений фактических параметров, то во время трансляции размер этого массива определить невозможно. Следовательно, до момента вызова функции невозможно определить размер записи активации. Существует по меньшей мере два способа преодоления этой трудности. Независимо от того, какой способ применяется, в точку входа в функцию неявно для программиста транслятор добавляет инструкции, вычисляющие для каждого такого массива требуемый размер памяти.

– При первом способе (см. рис. 8.11) окончательное определение размера поля локальных данных и размера всей записи активации откладывается до момента входа в функцию.

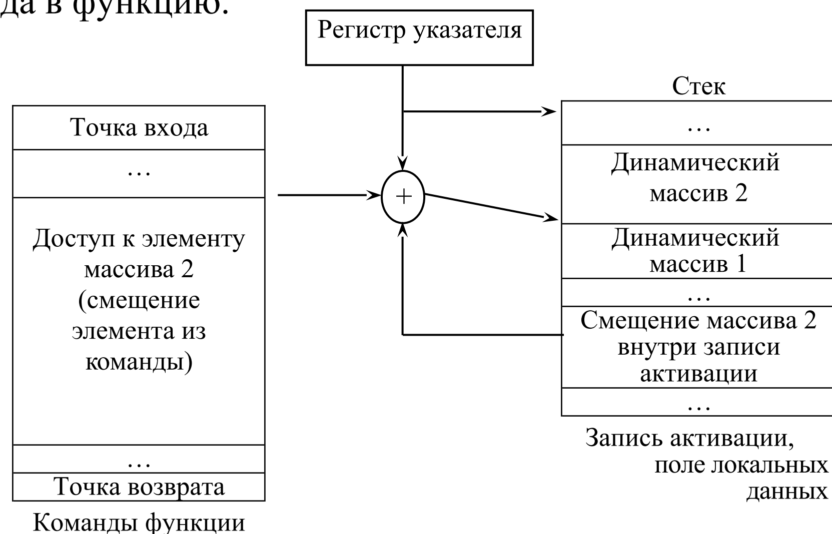


Рис. 8.11. Формирование исполнительного адреса, элемент динамического массива внутри поля локальных данных

Ясно, что адреса всех таких массивов, кроме первого из них, не могут быть зафиксированы во время трансляции. Следовательно, машинные команды, оперирующие с элементами второго и всех последующих массивов, не могут быть построены тем же способом, что и команды, работающие с обычными локальными данными. Вызвано это тем, что для формирования исполнительного адреса элемента массива теперь приходится складывать три величины:

– базовый адрес записи активации;

– смещение первого элемента динамического массива внутри записи активации (вычисляется после входа в функцию и хранится в качестве локального объекта, построенного транслятором);

– смещение элемента массива (берется из команды либо вычисляется до ее выполнения, если транслятору неизвестны индексы этого элемента).

Второй способ, показанный на рис. 8.12, состоит в том, что память для динамических массивов не резервируется в записи активации, а выделяется из кучи после того, как вычислен требуемый размер массива.

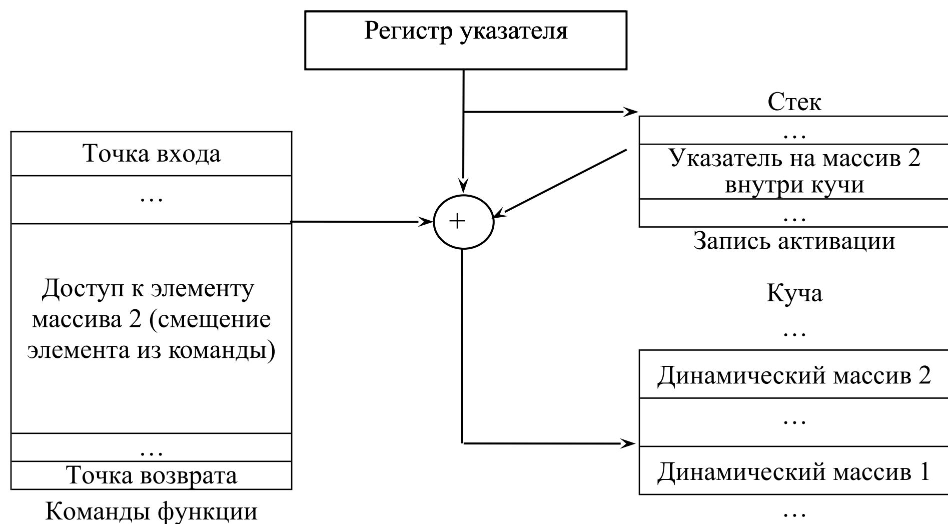


Рис. 8.12. Формирование исполнительного адреса, память для динамического массива выделяется из кучи

В записи активации во время трансляции резервируется фиксированное количество байт памяти для хранения указателя на память, взятую из кучи. Указатель используется в командах, оперирующих с элементами массива, которые строятся транслятором не так, как команды, работающие с обычными локальными данными (в том числе массивами). Адрес памяти для доступа к элементу динамического массива при хранении его в куче вычисляется так:

- вначале известное транслятору смещение указателя массива складывается с адресом записи активации и используется для выборки адреса массива и помещения его в регистр процессора;
- затем значение этого регистра складывается со смещением нужного элемента массива и по полученному адресу извлекается операнд.

Главным достоинством использования кучи для хранения динамических массивов является вычисление размера записи активации во время трансляции, а не во время исполнения программы.

3.9.Вызывающие последовательности

Вызовы функций реализуются путем исполнения так называемых вызывающих последовательностей команд. Эти последовательности размещаются частично в вызывающей, частично в вызываемой функции. Они обеспечивают передачу управления и данных из вызывающей функции в

вызываемую (последовательность вызова), и наоборот – из вызываемой в вызывающую (последовательность возврата). Последовательность вызова создает запись активации и заполняет ее поля необходимой информацией. Последовательность возврата восстанавливает состояние машины таким образом, чтобы вызывающая функция могла продолжать исполнение.

Последовательности вызовов и структура записи активации различаются даже в разных реализациях одного и того же языка. Коды в последовательностях вызова и возврата зачастую разделяются между вызывающей и вызываемой функциями. Не существует однозначного разграничения задач между вызывающей и вызываемой функциями – исходный язык, целевая машина и операционная система налагают свои требования, в силу которых может оказаться предпочтительным то или иное решение.

Поскольку каждый вызов имеет собственные фактические параметры, вызывающая функция обычно вычисляет фактические параметры и передает их в запись активации вызываемой функции. В стеке времени исполнения запись активации вызывающей функции находится непосредственно под записью активации вызываемой функции, как показано на рис. 8.13. Размещение полей параметров и возможного возвращаемого значения вслед за записью активации вызывающей функции дает некоторый выигрыш за счет того что вызывающая функция может получить доступ к таким полям, используя только смещения от конца собственной записи активации. Для этого не нужна полная информация о компоновке записи активации вызываемой функции. В частности, при трансляции вызывающей функции, как правило, ничего не известно о локальных переменных вызываемой функции.

Несмотря на то что размер поля для временных значений в конечном счете фиксируется во время компиляции, он может быть неизвестен вплоть до окончания этапов оптимизации и первой фазы генерации объектного кода. На этих этапах количество временных переменных для хранения промежуточных значений может быть уменьшено, а следовательно, на этапе семантического анализа размер этого поля может быть неизвестен. Поэтому поле временных переменных лучше размещать после поля локальных данных, поскольку изменения его размера не будут влиять на смещение прочих локальных объектов.

Если вызов функции встречается в тексте программы n раз, то часть последовательности вызова в вызывающих функциях будет построена транслятором n раз, в то время как часть последовательности вызова в вызываемой функции генерируется лишь единожды. Следовательно, для экономии памяти желательно разместить как можно большую часть последовательности вызова в коде вызываемой функции. Однако, очевидно, не всю работу по организации вызова можно переложить на вызываемую функцию.

Для выполнения всей работы необходимы два регистра процессора: указатель стека и указатель текущей записи активации.

Указатель стека (УС) в любой момент времени разделяет занятое и свободное пространства в стеке и используется всегда для доступа к единственному объекту, находящемуся на верхушке стека. Он не может быть использован для доступа к любым другим объектам в силу того, что значение этого указателя постоянно изменяется в результате выполнения различных команд. Поэтому для хранения базового адреса записи активации, как правило, выделяется еще один регистр процессора – указатель записи активации (УЗ). С использованием двух регистров типичная последовательность действий по вызову функции выглядит так.

Перед началом процесса вызова регистр УЗ указывает на конец поля состояния машины в записи активации.

1. Вызывающая функция вычисляет фактические параметры и заносит их в стек, модифицируя УС.
2. Вызывающая функция сохраняет адрес возврата и старое значение УЗ в записи активации вызываемой функции, после чего устанавливает значение УЗ равным значению УС, как показано на рис. 8.13. Таким образом, УЗ перемещается за локальные и временные данные вызывающей функции, параметры и поле состояния машины вызываемой функции.
3. Управление передается вызываемой функции, которая сохраняет в стеке значения регистров и другую информацию о состоянии машины и модифицирует УС таким образом, чтобы зарезервировать место под свои локальные данные и временные значения.
4. Вызываемая функция инициализирует локальные данные и переходит к исполнению своего тела.

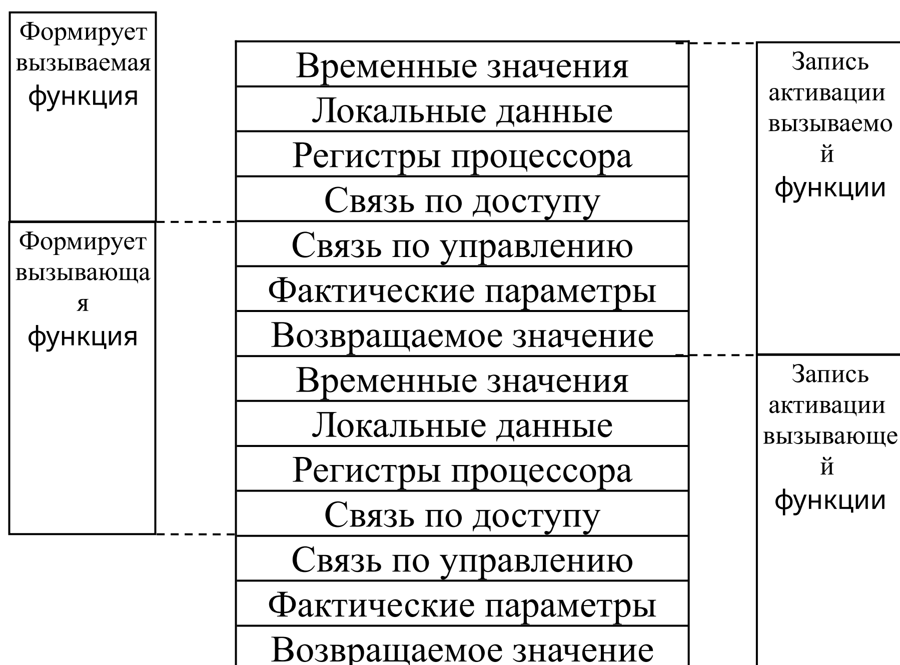


Рис. 8.13. Разделение задач между вызывающей и вызываемой функциями

Возможная последовательность возврата из функции выглядит следующим образом.

1. Вызываемая функция размещает возвращаемое значение после записи активации вызывающей программы. Используя информацию в поле состояния машины, вызываемая функция восстанавливает УЗ и другие регистры и передает управление по адресу возврата в коде вызывающей функции.
2. Хотя значение УЗ было уменьшено, вызывающая функция может скопировать возвращаемое значение в собственную запись активации и использовать его в вычислениях.

Приведенные выше последовательности вызова позволяют реализовывать функции с переменным числом аргументов. Транслятор, обрабатывая вызывающую функцию, может подсчитать количество фактических параметров. Следовательно, при формировании кода вызывающей функции ему известен размер поля параметров, который может быть неявно передан тоже в виде параметра. Теперь остается только таким образом формировать код вызываемой функции, чтобы он правильно обрабатывал неизвестное при трансляции, но получаемое в момент вызова количество параметров. Другой возможный вариант способа реализации переменного количества параметров – функции типа *printf* в языке C. Первый параметр такой функции содержит перечень и типы остальных параметров в виде текстовой строки. Поэтому, как только код тела функции *printf* выбирает первый параметр, он может по нему определить способ доступа к произвольному количеству остальных параметров.

3.10. Доступ к нелокальным объектам

Правила области видимости языка определяют способ работы со ссылками на нелокальные объекты – такие, объявления которых находятся вне текста функции, в которой объект используется. Существует общее правило, именуемое правилом текстуальной (или статической) области видимости, согласно которому видимость любого объекта определяется исключительно расположением его объявления в тексте программы.

В языках Pascal, C и Ada и многих других используется текстуальная область видимости с добавлением правила «ближайшее вложенное».

В некоторых языках программирования используется другое правило динамической области видимости, при котором ассоциация наименования любого объекта с его объявлением определяется не текстом программы, а совокупностью имеющихся на данный момент активаций функций. Согласно этому правилу доступ к значению нелокального объекта осуществляется путем поиска его имени в таблицах, ассоциированных с активациями функций. Просмотр активаций функций должен осуществляться в заранее обусловленной последовательности. Правило динамической области видимости используется в таких языках, как Lisp, APL и Snobol.

Исторически понятие нелокальных объектов появилось в языке Algol и связано с впервые введенной в этом языке конструкцией блоков. Блок представляет собой последовательность инструкций, содержащую объявления собственных локальных данных. Основная отличительная

особенность понятия блока заключается во вложенности структуры. Блоки либо следуют друг за другом, либо один блок является полностью вложенным в другой. Невозможно такое перекрытие блоков B_1 и B_2, при котором первым по тексту начинается блок B_1, после чего начинается блок B_2, затем заканчивается блок B_1, а затем – B_2.

Начало и конец блока оформляются специальными ограничителями (в С для этой цели используются фигурные скобки *{* и *}*; в Algol и Pascal – ключевые слова *begin* и *end*).

3.11. Блочные области видимости

Область видимости любого объекта в языке с блочной структурой определяется правилом ближайшего вложенного, суть которого состоит в следующем:

1. Область видимости объявления в блоке *B* включает весь блок *B*.
2. Если объект *x* не объявлен в блоке *B*, то все появления имени *x* в *B* находятся в области видимости объявления *x* во внешнем окружающем блоке B_o, таком, что:
   - B_o содержит объявление объекта с именем *x*;
   - B_o является ближайшим окружающим *B* блоком среди всех, содержащих объявление *x*.

Для пояснения того, как действует правило ближайшего вложенного, приведем следующий фрагмент программы на языке С (рис. 8.14). В нем каждое объявление инициализирует объявляемый объект числом, равным номеру блока, в котором оно появляется. Обратим внимание на то, что область видимости целой переменной *b*, объявленной в блоке B_0, не включает блока B_1, поскольку в блоке B_1 объявлена другая переменная с тем же именем *b*. Такой разрыв называется «дырой» в области видимости объявления.

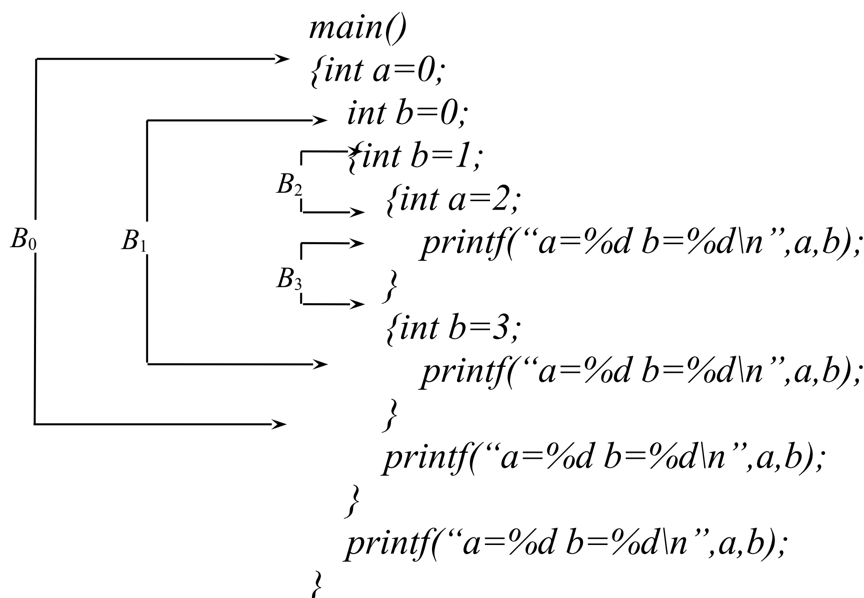


Рис. 8.14. Блоки и видимость переменных в С-программе

Всего в этом тексте объявлено пять разных объектов, области видимости которых приведены в табл. 8.2:

Таблица 8.2

Объявление	Область видимости
int a = 0;	B_0, B_2
int b = 0;	B_0, B_1
int b = 1;	B_1, B_3
int a = 2;	B_2
int b = 3;	B_3

Результаты работы этой программы иллюстрируют правило «ближайшего вложенного». При ее исполнении управление передается внутрь блока из точки, находящейся непосредственно перед ним, а затем из блока возвращается в точку, следующую прямо за ним в исходном тексте. Инструкции печати выполняются в последовательности B_2, B_3, B_1 и B_0 (в порядке, в котором завершается исполнение блоков). Значения переменных *a* и *b* в этих блоках показаны в табл. 8.3.

Таблица 8.3

Блок	*a*	*b*
B_2	2	1
B_3	0	3
B_1	0	1
B_0	0	0

Блочная структура программы может быть реализована с использованием стекового распределения памяти. Поскольку область видимости объявления не выходит за пределы того блока, в котором оно записано, память для объявленного объекта может быть выделена при входе в блок и освобождена при выходе из него. Это эквивалентно использованию блока как «процедуры/функции без параметров», вызываемой только из одной точки непосредственно перед блоком и возвращающей управление в точку непосредственно за блоком.

Нелокальная среда ссылок для блока может поддерживаться с использованием технологий для функций, которые будут рассмотрены в этой главе ниже. Заметим, что с точки зрения разработчика транслятора реализация блоков несколько проще, чем реализация функций, поскольку блокам не передаются параметры, а поток управления строго следует статическому тексту программы (не осуществляются передачи управления).

Другая возможная реализация программ с блочной структурой состоит в однократном выделении памяти для всех локальных объектов функции в тот момент, когда ей передается управление. При наличии блоков в функции размер поля локальных данных ее записи активации рассчитывается транслятором с учетом всех объявлений внутри блоков.

Для переменных программы, приведенной на рис. 8.14, можно выделить память так, как показано на рис. 8.15. Индексы у локальных переменных *a* и *b* на этом рисунке соответствуют блоку, в котором они объявлены.

...	a_0	b_0	b_1	a_2, b_3	...

Рис. 8.15. *Локальные объекты в записи активации*

Заметим, что для объектов a_2 и b_3 может быть отведена одна и та же область памяти, поскольку блоки B_2 и B_3 не пересекаются.

При отсутствии динамических массивов (и других данных переменной длины) максимальное количество памяти, необходимой для выполнения блока, может быть определено в процессе компиляции (с данными переменной длины можно работать, используя указатели). Определение

этого размера транслятор должен выполнять исходя из предположения, что при исполнении программы будут пройдены все возможные пути управления по тексту функции.

3.12. Текстуальная область видимости без вложенных функций

Для изучения того, каким образом реализуется видимость нелокальных объектов в языке, не допускающем текстуально вложенных функций, рассмотрим программу сортировки элементов массива на языке C (рис. 8.16).

Эта программа предназначена для демонстрационных целей, ее не следует (как и приведенную ниже программу на языке Pascal) рассматривать в качестве образца программирования или реализации методов сортировки. Правила текстуальной области видимости в языке C проще, чем в языке Pascal (для языка Pascal они обсуждаются позже) вследствие того, что определение функции не может находиться внутри другой функции.

Если в некоторой функции используется не объявленный в ней объект *X*, то предполагается, что его объявление находится вне любой функции. Область видимости любого такого объявления состоит из тел функций, следующих за ним (объявлением) по тексту файла (с дырами, если внутри функций имеются объявления объектов с точно такими же именами).

```
(1)  int arr[   ];
(2)  void ReadArray(void) { ... arr ... }
(3)  int PartExch(int iBeg,int iEnd) {
(4)      int i, j, k, mVal;
(5)      i=iBeg; j=iEnd; mVal=arr[i+(j-i)/2];
(6)      while(i<j) {
(7)        for( ;arr[i]<mVal;i+=1);
(8)        for( ;arr[j]>mVal;j-=1);
(9)        if(i<j){k=arr[i];arr[i]=arr[j];arr[j]=k;}
(10)     }//end while
(11)     return i;
(12) }  //end PartExch
(13) void QuickSort(int iBeg,int iEnd) {
(14)     int iMid;
(15)     if(iBeg<iEnd) {
(16)       iMid=PartExch(iBeg,iEnd);
(17)       if(iBeg<iMid-1) QuickSort(iBeg,iMid-1);
(18)       if(iMid+1<iEnd) QuickSort(iMid+1,iEnd);
(19)     }
(20) }  //end QuickSort
(21)  void main() {
(22)     ReadArray();QuickSort(0,(sizeof(arr)/sizeof(int))-1);
(23) }  //end main
```

Рис. 8.16. C-программа с нелокальным массивом arr

При отсутствии вложенных функций для языка с текстуальными областями видимости можно использовать стратегию стекового распределения памяти для локальных объектов. Память для объектов, объявленных вне функций, может быть выделена статически. Положение этой памяти известно в процессе компиляции, так что даже если некоторый объект нелокален в теле данной функции, транслятор будет использовать статически определенный адрес объекта для построения команд, оперирующих с его значением.

Важное преимущество статического выделения памяти для нелокальных объектов заключается в том, что любые функции могут свободно передаваться в качестве параметра другим функциям и возвращаться как результат вызова (в языке С функция передается как параметр путем передачи указателя на нее). При использовании текстуальной области видимости и отсутствии вложенных функций любой нелокальный для одной функции объект является нелокальным и для всех остальных функций, а следовательно, его статический адрес может использоваться всеми функциями одинаково и независимо от того, каким образом они активируются. Это справедливо для любой функции, в том числе и для функций, которые вызываются по вычисляемым ссылкам.

Доступ к другим элементам среды ссылок следует изучить по [6-9].

4) Порядок выполнения работы (рекомендуется использовать в качестве примера систему правил Samples/Sample8):
   4.1) Расширить систему действий синтаксического анализатора, построенного при выполнении лабораторных работ 3 – 7, действиями для исполнения последовательности тетрад (триад/пентад согласно варианту задания на курсовую работу);
   4.2) Обеспечить выполнение некоторых семантических проверок для выявления ошибок времени исполнения при интерпретации;
   4.3) Построить интерпретатор (виртуальную машину) для учебного языка или его подмножества, убедиться в его работоспособности.
   4.4) Подготовить, сдать и защитить отчет к лабораторной работе.
5) Требования к содержанию отчета.

   Отчет должен содержать:
   - 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.
6) Контрольные вопросы
   6.1) Что такое побочный эффект вызова процедуры/функции?
   6.2) Что такое глобальный, локальный и нелокальный объект?

6.3) Что такое текстуальная область видимости без вложенных функций?
6.4) Что такое передача аргумента по имени? Как она реализуется?
6.5) Перечислите способы передачи аргументов в функцию.
6.6) Что такое жизненный цикл наименования объекта?
6.7) Что такое кодированное представление типов данных? В чем состоят его достоинства и недостатки?
6.8) Что такое вложенность процедур/функций?
6.9) Что такое нелокальная среда ссылок?
6.10) Расшифруйте словами тип данных языка С:
(int[](unsigned))*.
6.11) Что такое именное представление типов данных? В чем состоят его достоинства и недостатки?
6.12) Что такое передача аргументов методом копирование-восстановление?
6.13) Что такое запись активации процедуры/функции?
6.14) Как делится ответственность за формирование записи активации между вызывающей и вызываемой функциями?
6.15) Что такое дерево активации функций?
6.16) Перечислите минимально необходимые поля записи активации функции.
6.17) Что такое блочная область видимости переменной?
6.18) Что такое вызывающая последовательность?
6.19) 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 М 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 М 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 М 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.